DYNAMOS: Dynamically Adaptive Microservice-based OS

A Middleware for Data Exchange Systems

Jorrit Stutterheim
jorrit.stutterheim@gmail.com

July 16, 2023, 77 pages

Academic supervisor: Ana Oprescu, A.M.Oprescu@uva.nl
Second reader: Thomas van Binsbergen, l.t.vanbinsbergen@uva.nl
Research group: UvA: Complex Cyber-infrastructure group. https://cci-research.nl/

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

Data sharing has become increasingly important in modern business and research. Consequently, many initiatives are being developed worldwide to facilitate open data exchange in a secure distributed environment. Within this context, each party maintains control over their data and can implement access through legal, programmable policy.

Research on this topic has pinpointed common data-sharing patterns, known as archetypes, and has provided suggested architectures to support these platforms. The choice of an optimal archetype or architecture, however, may change over time, as policy evolves or the runtime environment receives more requests.

In addressing these dynamic requirements, deploying a static, monolithic application might not be the most efficient approach. In contrast, the microservice paradigm, which has gained widespread acceptance, offers interesting opportunities. Microservices can be dynamically created to optimize resource consumption and dynamically orchestrated to handle diverse data-sharing requests, archetypes, and architectures. Furthermore, by allowing the creation of data-exchange microservices, new functionalities can be quickly introduced without changing the core system.

This thesis introduces DYNAMOS: 'Dynamically Adaptive Microservice-based Operating System'. Inspired by the way 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. This is done by simulating the distributed nature of a data-sharing platform and recreating a real-life data-sharing use case, called the 'Research Data Exchange' that focuses on enabling data exchange between universities for research purposes.

The current implementation shows how architectures, microservices, and data-sharing patterns can be selected dynamically per request. The communication and transmission of data between microservices are seamlessly abstracted, allowing them to focus on their primary functions. For this thesis, we delve into the design decisions made when developing DYNAMOS, the disadvantages, and advantages of different microservice architectures, and this approach to tackling data-exchange problems.

One key insight drawn from our study is that using a microservice architecture in data-exchange scenarios may increase latency due to the additional data transfers from microservice to microservice. Secondly, flexibility is key, we can't foresee and judge exact requirements on the trade-offs of security, architectures, latency, and evolving policy. Thus DYNAMOS is designed to be self-adaptive, utilizing extendable algorithms to choose among architectural or archetype patterns, influenced by policy, user input, or system events.

Contents

Chapter 1

Introduction

Data sharing has become increasingly important in modern business. The ability to access larger datasets can significantly improve the performance of machine learning models, while in fields like medical science, the availability of more patient data holds immense potential. However, for data sharing to be successful, it must align with the desired level of privacy and security of each participating party [1].

To create a platform for this, various initiatives around the world work on establishing a Digital Data-sharing Marketplaces (DDM)[2]. One such initiative is the Amsterdam Data Exchange Fieldlab (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 policies.

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.

This thesis introduces DYNAMOS: 'Dynamically Adaptive Microservice-based OS'. Inspired by the way 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. 'Decentralized' in DDM is actually an important keyword here, which makes DYNAMOS a *distributed* OS.

In this thesis, we introduce the generic term *'microservice chains'* to describe various microservice patterns. A chain is a 'directed acyclic graph' of microservices that form a valid data exchange scenario and adheres to policy agreement. These chains need to be executed, for this, we use the generic term 'jobs'. A user request flows through this chain job deployed to the parties set out by the agreements.

Through this approach, we aim to achieve the following advantages. Firstly, we strive to establish a self-adaptable system capable of seamlessly incorporating changes in agreements without the need for developer intervention. Secondly, by introducing lightweight data exchange microservices, we eliminate the necessity to redeploy monolithic applications, resulting in a more agile and efficient environment. Lastly, the 'operating system' will allow for optimizing extra-functional properties, enabling cost and energy savings based on the system's environment.

1.1 Problem statement

To create a system like we described in the previous paragraph a list of problems needs to be solved. First, we need a system or language to link microservices to both an archetype and a type of data-sharing request.

Secondly, we need adaptability. Per user, or per request of the same user, different architectures or data-exchange patterns (archetypes) could be selected to run. This requires the routing of data and requests to be highly dynamic. This behavior should be able to adapt based on evolving agreements, user-input or extra-functional properties.

A common approach to achieve self-adaptability is the use of a MAPE-K control loop, which stands for (M)onitor, (A)nalyze, (P)lan, (E)xecute, over a shared (K)nowledge base [4]. Part of this thesis is finding out what information (Knowledge) DYNAMOS needs to generate microservice chains and optimize for extra-functional properties, and how this can be practically used for implementing self-adaptability.

Finally, whatever architectures or archetypes are selected, should be invisible to the user. In other words, where data requests are computed or sent does not change the result or interface from a user's perspective.

1.1.1 Research questions

Building this dynamic system will require choices in technology and design, each with trade-offs and advantages. Thus we derived the following research questions, focusing on what knowledge is required to make a system adaptable and what the consequences of choices in the design is.

RQ1 What are the advantages and disadvantages of different approaches to automatically re-compose a running microservices architecture in a distributed system?

RQ2 What attributes/properties should a 'knowledge-base' contain to compose microservices in a DDM environment?

RQ3 What extra attributes/properties should a 'knowledge-base' contain to make decisions on extra-functional properties, like energy consumption and performance?

1.1.2 Research method

DYNAMOS is developed using **action research** [5]. The main focus is implementing the UNL use-case from the AMDEX project. In the UNL case, a data analyst performs analyses on wage information of different universities across the Netherlands.

The results of this research will be an analysis of the choices made in developing DYNAMOS. The choices made in technology, and the methods of supporting DYNAMOS's core functionality, e.g. creating microservice chains, jobs, implementing self-adaptability, and dynamic routing.

The literature study is focused on the existing architecture proposals for DDM and methods for attaining self-adaptability.

Finally, a set of quantitative experiments will be performed on DYNAMOS to establish a performance baseline. In these experiments, we look at how DYNAMOS handles data transferring and identify bottlenecks of the implementation.

1.2 Contributions

Our research makes the following contributions:

1.2.0.1 Core research contributions

CR1 Combining data-sharing requests and DDM archetypes into microservice chains
CR2 Specific microservice requirements for generic microservices to perform the AMDEX UNL use case scenarios
CR3 Analysis of how baseline extra-functional properties (costs, energy, etc.) can be taken into account in the system.
CR4 Analysis of different approaches to making DYNAMOS
CR5 Architecture to bridge the gap from a formal policy evaluation result, to a web environment where jobs are executed.

1.2.0.2 Core technical contributions

CT1 Generic method to create single-use, self-cleaning microservice jobs
CT2 Generic communication method of transferring data, up to 4 MB, through microservice chains
CT3 Exact types, code, and templates to generate microservice chains
CT4 Decoupled messaging solution, with exact message definitions for communication in DYNAMOS
CT5 An open-source system that allows experimentation with microservice chains in a distributed environment

1.3 Outline

In Chapter 2 we describe the background of this thesis. Chapter 3 describes the conceptual design of DYNAMOS, and Chapter 4 shows the technical implementation. Chapter 5 shows the quantitative

experiments and their results. While the thesis results are shown in Chapter **??** and discussed in Chapter 6. Chapter 7, contains the work related to this thesis. Finally, we present our concluding remarks in Chapter 8 together with future work.

Chapter 2

Background

This chapter provides the necessary background information for this thesis. First, we list the properties of an OS and of a distributed OS. Subsequently, the AMDEX Fieldlab proposition, the UNL use-care, and related data-sharing terminology are introduced. Finally, best practices in the context of distributed systems, microservices, and microservice orchestration are described.

2.1 Operating systems

An operating system is software that manages a computer's hardware, provides a basis for application programs, and acts as an intermediary between the computer user and the computer hardware in a wide variety of computing environments. We can also view a computer system as consisting of hardware, software, and data. The operating system provides the means for the proper use of these resources in the operation of the computer system. It performs no useful function by itself but provides an environment within which other programs can do useful work. As described by Silberschatz *et al.*, 2018[6].

In other words, the OS abstracts away the complexities of the system to allow the users to program or use applications that do something useful. An OS contains the following key attributes *et al.*, 2018[6]:

- Process Management
- Memory Management
- File system Management
- Device Management
- User Interface
- Security
- Networking
- Error Handling
- Multi-programming
- Logging

2.1.1 Distributed Operating systems

"A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicate with one another through various communication lines, such as high-speed buses and the Internet."[6]

In a distributed operating system the properties of a 'regular' OS, listed above are inherited. An extra set of properties attributed to distributed operating systems is listed in table 2.1.

Thus, DYNAMOS should, to a certain extent, be able to provide these OS attributes to the user. How DYNAMOS, adheres to these policies will be further explored in chapter 3.

Property	Summary
Transparency	The use of multiple processors should be invisible (transparent) to the user [7]
Concurrency	Computations can be done concurrently to improve throughput and response time [8]
Reliability	If a part goes down, other parts of the system still work, or take over the work [7]
Resource management	Load can be transferred to idle parts of the system [7]
Scalability	Handle increased load and adding nodes without degrading performance [6]
Flexibility	The most appropriate target for a type of computation can be selected [8]
Security	As a regular OS processes should not interfere with each other, and unauthorized access is prevented [6]

Table 2.1: Properties of a distributed OS

2.2 Distributed system management

"Large-scale distributed applications require different forms of coordination. Configuration is one of the most basic forms of coordination." [9] ZooKeeper, is one first generic applications that allow developers to write their own coordination primitives for distributed applications. To manage the complexities of distributed systems, other coordination tools have evolved.

These tools centralize services for configuration management, providing a consistent repository, and ensuring that each node in the system is in sync with the latest configurations. Beyond configuration, these tools keep a cohesive view of the system's state, be it active or failed nodes. This overview facilitates tasks such as leader election and distributed locking, optimizing resource management. Furthermore, features like "watches" alert nodes about specific system changes.

Among the tools available, prominent ones like ZooKeeper[1], etcd[2], Consul[3], and Apache Mesos[4] stand out, each catering to the specific needs of distributed systems.

2.3 Amsterdam Data Exchange Fieldlab

The following text has been distilled from the AMDEX funding proposal in which the Fieldlab goals are outlined [2]. The 'Amsterdam Data Exchange Fieldlab' is done by a consortium of the following entities: Amsterdam Internet Exchange B.V, Dexes B.V., Stichting Amsterdam Economic Board, SURFsara B.V., and the University of Amsterdam. The goal of the consortium is to develop a neutral, generic, independent infrastructure that ensures sovereignty in data exchange initiatives with the working title *Amsterdam Data Exchange* (AMDEX). With AMDEX it should be possible to:

- On the supply side, offerr interested parties reliable and fair data markets and contracts
- Empower companies to let third parties use their data without giving a copy of the data
- Use this FieldLab to demonstrate a few of the possible workflows
- Facilitate data on AMDEX infrastructure
- Use this lab to perform tests

Studies by the Dutch economic ministry and the European Commission show that the current infrastructure for safely sharing data is not adequate. Even worse, there are a number of foreign large tech companies that gather and own huge amounts of data, of which the consumers giving the data have no control[2]. The AMDEX project is a part in a larger project to tackle these types of problems. A concrete example is medical data; hospitals and patients are open to sharing patient data with researchers but can't due to a lack of legal agreements and infrastructure.

At the request of the Amsterdam Economic Board, a few proof of concepts (POC's) have been done. A few important problems that resulted from these POC's are listed below.

- For data suppliers there is a high technical challenge and legal red tape which makes it hard

[1] `https://zookeeper.apache.org/`
[2] `https://etcd.io/`
[3] `https://www.consul.io/`
[4] `https://mesos.apache.org/`

- The first AMDEX use cases haven't been designed for scalability
- Data suppliers are now usually limited to market parties that try to create a monopoly (loose control over your data)
- Sharing over borders presents a lot of legal issues

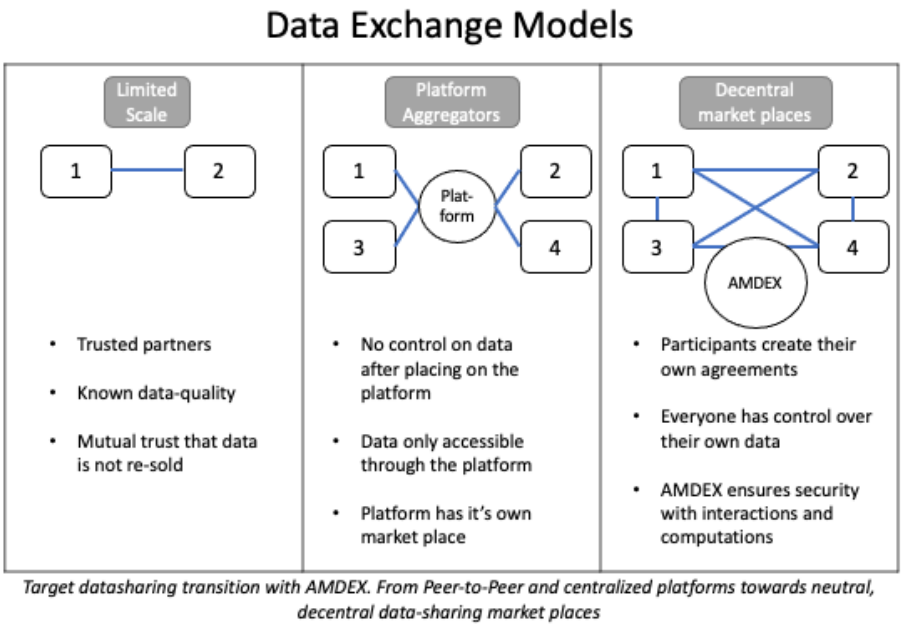

Figure 2.1: Data sharing architectures [2]

The third column of figure 2.1 shows the target of AMDEX, an *internet of data*, where multiple parties exchange data and algorithms in a secure, open, federated way.

2.4 Data-sharing terminology

- **Data steward**: entity within the DDM that owns datasets that can be exchanged
- **Data analyst**: person or institution that has entered into an agreement with a data steward to interact with a dataset
- **Data pod**: entity in the data stewards domain that contains datasets in some form

Research on DDM has developed a set of typical patterns in which data can be exchanged, these patterns are called archetypes [1, 3]. Whether a party is allowed to access data, and which archetypes will be chosen to facilitate the data exchange, is encoded in policies. Thus, an agreement in a policy leads to a chosen archetype, which leads to data exchange.

2.4.0.1 Archetypes

In general, the participating organizations in a DDM can share two kinds of resources: software and data [1]. 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.

To more easily reason about the ways these data-transactions occur the term 'archetypes' has been coined [3][1]. Figure 2.2 shows an overview of a set of archetypes. The UNL use-case from this thesis focuses on the 'Data through a trusted third party' and 'Compute to data' archetypes.

2.4.0.2 Policy

Within the AMDEX project, the part responsible for enforcing agreements is called a policy enforcer. There is research in progress to make these agreements programmable through eFLINT [10], which would

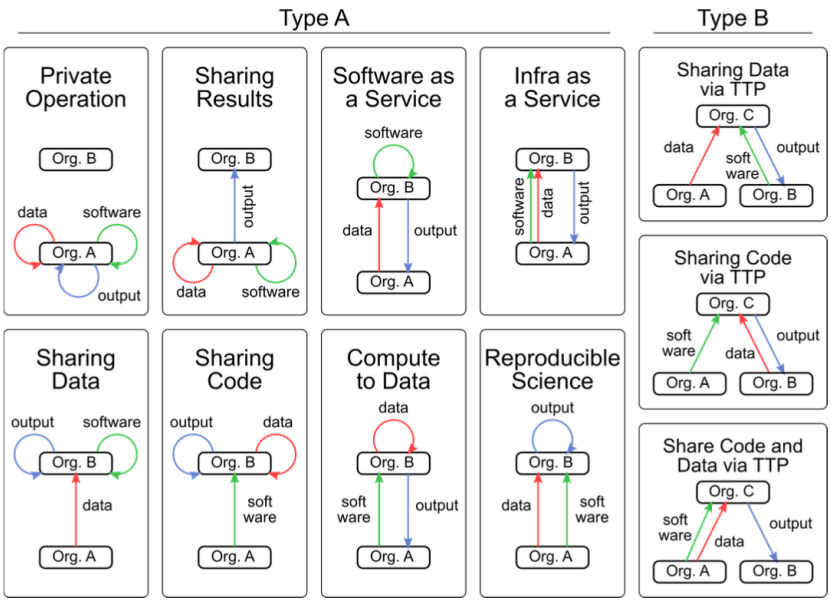

Figure 2.2: Data-sharing archetypes [1]

enable an automatic policy enforcement framework. In this thesis, we mock this policy enforcer to mimic request validation and have a source of archetype information.

2.5 Microservices

The concept of microservices was initially introduced in 2014 by Lewis and Fowler [11], and since then, it has gained considerable popularity. Prominent organizations such as Netflix [12] and Zalando [13], among others, have adopted microservices as the foundation of their architectural frameworks. While there exist various approaches for implementing microservices, the utilization of lightweight Docker containers stands out as the most widely recognized method. Nevertheless, specific language frameworks, such as Erlang, Elixir, and Spring Boot for Java, have also emerged as proficient options for microservice development. For the purpose of this thesis, however, Docker will be the primary focus.

Supposedly, there are many advantages to using microservices. These include flexibility in dealing with uncertainties related to operation, maintenance, and evolution [14]. It also allows for greater software development agility and improved scalability of applications that are deployed. Additionally, it offers cheaper deployment costs by only scaling up selected Microservices that experience high-load [12]. However, microservices do add a lot of complexity to the system, especially in design, testing, and monitoring [15]. According to the 2022 study Kubernetes benchmark [16], 53% of companies underestimate this complexity.

Within this section, we delve into various orchestration tools for managing microservices, briefly explore communication patterns, and discuss microservice best practices that will serve as points of reference and utilization throughout this thesis.

2.5.1 Microservice orchestration

The most used Docker orchestration tools are Kubernetes, Docker Swarm, Apache Mesos, and RedHat OpenShift which is actually built on Kubernetes. Although it is difficult to get exact numbers, most online articles and studies agree that Kubernetes is the most-used platform and gaining popularity every year [17–19].

Kubernetes is generally perceived as more complex, with a steeper learning curve, when compared to Docker Swarm. However, it does offer more services and customization options. For example, allowing custom resources definitions (CRDs) [20], operators [21], Kubernetes jobs [22], and make use of service meshes [23].

2.5.2 Microservice communication

Microservice communicate with each other using lightweight mechanisms, often an HTTP resource API [11], which is a synchronous protocol, an asynchronous alternative is AMQP. For primary operations such as data fetching or storing, HTTP REST is the best overall option thanks to its simplicity. If, on the contrary, the service needs to dispatch lots of messages to other services and trigger heavy processes that don't need to be awaited, AMQP is a better option [24].

2.5.3 Microservice best practices

Wang *et al.* [12] performed an exploratory study on microservice best practices, conducting in-depth interviews with microservice experts. The below paragraphs depend for a large part on that study, combined with online sources.

2.5.3.1 Logging

Logging is a challenge in microservice architectures. If something happens, it would be near impossible to check individual components for logs and correlate these to actual events [25]. Furthermore, setting up logging and monitoring early is crucial to the success of a microservice platform [12].

Within the Kubernetes environment, there are several off-the-shelf solutions available. In essence, Kubernetes will catch all output from a pod that is written to STDERR and STDOUT. A stack of existing services can capture that output enhance it, store it centrally, and make it visual and able to query.

Common solutions are the ELK stack, (Elastic search, Logstash, Kibana) or the combination of Promtail, Loki, and Grafana. However, because the application is unaware of the logging solution, all solutions are easily replaceable and can be switched if requirements demand.

2.5.3.2 Distributed tracing and Service Mesh

Metrics and traces are two more key components that enhance the monitorability of both the platform and microservice application. Where logs only show what went wrong according to the application. A trace would actually show the entire path, transit time, and time spent in services, of all requests. Experts express that metrics and tracing must be set up as early in the process as possible, for a successful microservice platform [12].

First, the distributed tracing ecosystem is very complex. It includes a dizzying array of projects such as Zipkin, Jaeger, OpenTracing, OpenCensus, OpenTelemetry, and many many others, each with partially overlapping sets of functionality. There are complex matrices of which projects can interoperate with which other projects and in which ways. [26]

These metrics, allow for identifying bottlenecks and spotting errors that at first seemed unrelated. Furthermore, a trace can be visualized which can help both with debugging and even showing parts of the application architecture.

Again, there are several solutions for metric collection and tracing. Prometheus is the de facto standard for metrics in Kubernetes and Zipkin and Jaeger are well-known tracing solutions.

This is where a service mesh comes in as a best practice. A service mesh is a dedicated infrastructure layer built right into a platform [27]. A service mesh can handle metric collection, distributed tracing, ingress encrypted traffic, load balancing, canary deployments, and more. All by injecting a sidecar proxy into every microservice and taking over (HTTP) traffic. The most well-known service meshes for Kubernetes are Istio and Linkerd, and other options include NGINX Service Mesh, Open Service Mesh, and HashiCorp Consul.

2.5.4 Sidecar pattern

The sidecar pattern moves common functionalities into a separate container located close to the primary application. The primary application will have one unified way to talk to the sidecar. Shared libraries, logging, messaging, and configuration are common use cases for the sidecar pattern, which is good according to the single responsibility principle [28]. The sidecar pattern also allows the sidecar and primary application to have separate release cycles to allow development from different teams [28].

2.5.5 gRPC

gRPC is a modern open-source high-performance Remote Procedure Call (RPC) framework that can run in any environment. It can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking, and authentication. It is also applicable in the last mile of distributed computing to connect devices, mobile applications, and browsers to backend services. [29]

For communicating with a sidecar a lightweight protocol is preferred. An HTTP-based REST method is common, but in most cases, gRPC will outperform REST [30, 31]. In short, gRPC is a binary protocol that allows efficient data packing and transfer. For example, REST transfer text (JSON or XML), so the message $\{\text{“}id\text{”} : 42\}$ will be approximately 9 bytes. While the same message in gRPC binary would be two bytes: “$0x08\ 0x2a$” [30]. Furthermore, gRPC supports unary and bi-directional streaming, which allows the transfer of large amounts of data.

Remote gRPC functions are declared in ‘protocol buffers’, which can then be generated to eleven common programming languages [32]. This allows language-agnostic interfacing. Supported language at the time of writing are:

- C# (or .NET)
- C++
- Dart
- Go
- Java
- Kotlin
- Node
- Objective-C
- PHP
- Python
- Ruby

2.6 Workflow tools

A workflow consists of an orchestrated and repeatable pattern of activity, enabled by the systematic organization of resources into processes that transform materials, provide services, or process information [33]. There are several off-the-shelf solutions for creating microservice workflows.

2.6.0.1 Argo

Argo Workflows is an open-source container-native workflow engine for orchestrating parallel jobs on Kubernetes. Argo Workflows is implemented as a Kubernetes CRD (Custom Resource Definition) [34].

Argo allows the output of one container as input to the next container of the workflow. Simple data like an integer or string can be transferred by outputting it to STDOUT. Complex data needs to be transferred through an artifact, however. This can be as simple as writing to a local file towards storing artifacts in external repositories like Amazon S3 storage.

2.6.0.2 Apache Airflow

Apache Airflow is a Python-based workflow scheduler, which can be used to schedule Kubernetes pods. Communication between different pods would require external data transfer methods like databases, AMQ messaging, or shared file systems. Airflow offers a high level of abstraction, its configuration in code, or through the Web-UI makes it less dynamic, however.

2.6.0.3 Kubernetes Jobs

A Job creates one or more Pods and will continue to retry execution of the Pods until a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (ie, Job) is complete. [22] The jobs can be configured to auto-delete after successful completion.

2.7 Data pods

A data pod is an abstract concept in which an entity has a secure container to store data. Users might have a data pod provided by their workplace, one from a commercial provider, and maybe even one they host themselves. The Solid foundation [35] is a project that lets people store their data securely in decentralized data stores called Pods, controlling which people and applications can access it.

Chapter 3

DYNAMOS: A microservice-based Operating System

In this chapter, we present the conceptual design of DYNAMOS. We describe the UNL use case [36] in depth to derive the DYNAMOS requirements and architecture. Subsequently, we show how microservice chains are generated and how these are used to dynamically create different microservice architectures. Several types of possible microservice architectures are described, and how these can be dynamically chosen.

Throughout this chapter, we reference back to the properties of a distributed OS, from table 2.1. The following list summarizes how DYNAMOS adheres to these properties, these will be referenced throughout the chapter to accentuate a feature.

OS1 **Transparency:** The user is agnostic to which archetypes or microservices are used within DYNAMOS
OS2 **Concurrency:** If there are multiple data stewards all will process concurrently, results are only awaited if aggregation is required
OS3 **Reliability:** DYNAMOS will handle failed jobs by re-scheduling, and restarting DYNAMOS components on failure
OS4 **Resource management:** Archetypes can be dynamically picked to distribute the load over the system
OS5 **Scalability:** DYNAMOS components can be dynamically scaled based on system load
OS6 **Flexibility:** Specific archetypes, microservice architectures, or target systems can be dynamically picked to handle certain request types
OS7 **Security:** A full authorization flow is an integral part of the system

3.1 UNL scenario

The scenario chosen to test DYNAMOS with is the existing AMDEX UNL use case[1]. In this section we discuss the use case, present a set of microservices that can be used to implement it, and finally list a set of initial requirements to cover the case.

3.1.1 Actors

Figure 3.1 shows an abstract representation of the actors in DYNAMOS involved in the UNL use case. A data analyst, through a frontend, requests an access token from the 'orchestrator'. After gaining access, the data analyst can query the data stewards. The following actors are part of the system.

- Frontend → Data analyst logs in, select a dataset, run 'practice' queries a synthesized dataset, and can start different request types.
- Policy enforcer → Not part of DYNAMOS, checks policy between parties and provides access tokens

[1] `https://amdex.eu/news/sharing-research-data-under-ones-own-conditions/`

- Orchestrator → Orchestrate requests to the system over the available computing space based on data from the policy enforcer.
- Distributed agents → Agent running on a data steward's system, accepts, checks, and regulates incoming requests and deploys the required microservice chains
- Data store → Distributed data store accessible by the agents and the orchestrator. Stores system state, and objects related to the distributed system e.g. archetypes, microservice metadata, agent configurations etc.
- Data pod → Some type of secure data store containing datasets

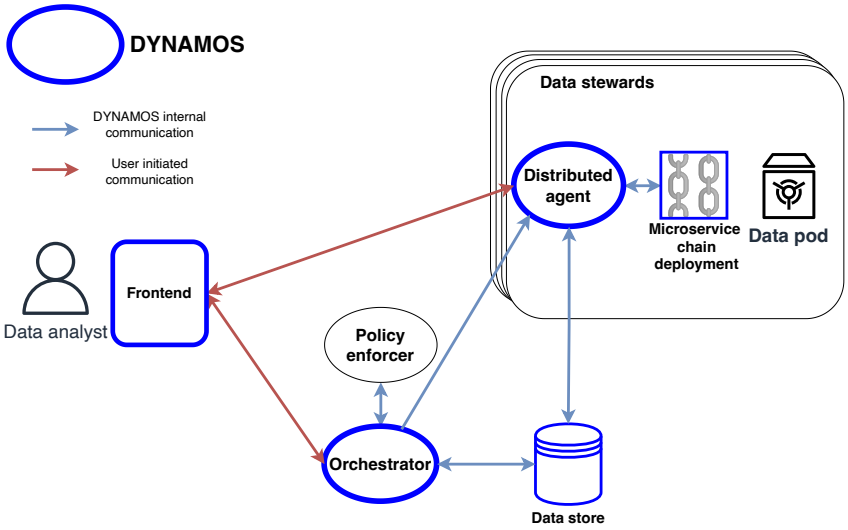

Figure 3.1: Abstract representation of DYNAMOS

3.1.2 Scenarios

For the UNL use case, we define two scenarios that work with two different archetypes, namely the 'compute to data' and the 'data through trusted third-party' archetypes, henceforward called *computeToData* and *dataThroughTtp*. The goal of the data analyst in the scenario is to analyze the wage information of two universities. Figure 3.2 shows the first scenario with the following steps.

3.1.2.1 Scenario one

1. Request approval → User requests approval from the 'orchestrator', listing the intended target data stewards
2. Validation Request → The orchestrator simply passes on the request. The 'policy enforcer' checks existing agreements and returns whether the requests are allowed. Returns an access token for the data analyst.
3. Composition request → The orchestrator chooses an archetype, generates a job ID, and sends this information including their role in the archetype to all distributed agents
4. Accepted request → The orchestrator checks whether all targets are online, and returns their endpoints, job ID and the access token to the data analyst
5. Data request → The data analyst can send a data request with queries and algorithms to the given endpoint(s), and wait for results

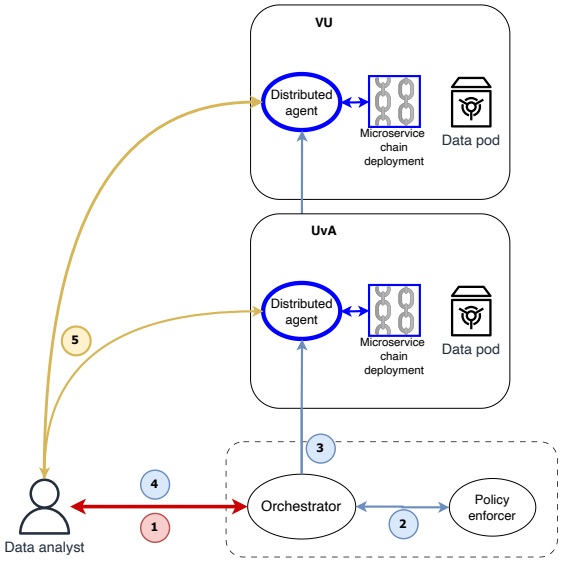

Figure 3.2: ComputeToData scenario

3.1.2.2 Scenario two

Figure 3.3 shows the second scenario with the following added steps.

6. Forward request → Data request is forwarded to all data providers
7. Data transfer → Requested data of the data providers are returned to the trusted third party
8. Process data → The data is further processed by the third-party
9. Return results → Final results are returned to the data analyst

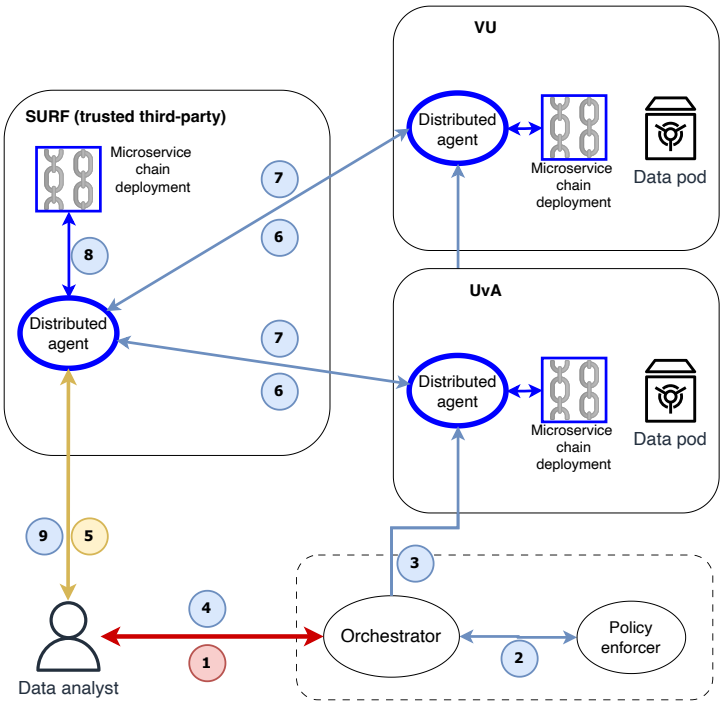

Figure 3.3: DataThroughTtp scenario

3.1.3 Microservices

For both scenarios, two basic microservices are required. A **'query service'** that performs a SQL request on the dataset and an **'algorithm service'** that can process the data e.g. perform a wage gap analysis.

However, to make the chains larger and to show the power of DYNAMOS's dynamic architecture we add three more microservices. These microservices could be written in different programming languages. DYNAMOS facilitated this by being language agnostic towards microservice chains.

3.1.3.1 Anonymize service

The data steward has added a system rule that certain columns of the dataset need to be anonymized before leaving the system. This service could also be part of a policy agreement, forced by the policy enforcer.

3.1.3.2 Graph service

The data analyst in this scenario has the option of adding a flag in the request to ask for a graph to be generated. This adds the graph service that will convert the results into a graph.

3.1.3.3 Aggregate service

For the dataThroughTtp scenario, a **'aggregation'** service is added to combine the results of all data stewards before further processing.

3.2 Requirements

Based on the scenarios we can create the initial requirements for DYNAMOS and its components.

3.2.1 DYNAMOS in general

DYN1 Authorization flow that allows every component to make sure the user has access and that allows access revocation
DYN2 Communication protocol for inter-component messaging
DYN3 Distributed data store for configuration management, system state, and concurrency management (concurrency OS2)
DYN4 Components should be replaceable as much as possible to prevent technology lock-in
DYN5 Basic security features such as RBAC, user authentication, and mTLS are supported (security OS7)

3.2.2 Orchestrator

ORCH1 Determine the availability of distributed agents
ORCH2 Query and interpret decisions of the policy enforcer
ORCH3 Make decisions on which archetypes should be used based on different variables (flexibility OS6)
ORCH4 Dynamically change an archetype when appropriate (transparency OS1)
ORCH5 Inform the data analyst of approved data providers and how to reach them
ORCH6 Inform distributed agents on approved data requests

3.2.3 Distributed agent

AGENT1 Create and manage data request execution for a user
AGENT2 Ensure proper authorization and security
AGENT3 Create microservice chains
AGENT4 Ensure resources are not consumed by removing services that are not actively in use (resource management OS4)
AGENT5 Determine action on results of a local microservice chain, send it to the user or a trusted-third-party
AGENT6 Ensure data requests can be retried after unexpected failures (reliability OS3)

3.2.4 Microservice chain jobs

JOB1 A form of job that can process different chains on every request (flexibility OS6)
JOB2 Effectively pass on results of a computation from a data-exchange microservice, to the next microservice in the chain. Or if the chain is finished to the distributed agent for further processing
JOB3 Be generalized enough to work with different datasets and perform transformations on the data
JOB4 Language agnosticism. Any mix of programming languages can facilitate data exchange jobs
JOB5 Based on the archetype, microservices could run on different parts of the system

3.2.5 Frontend

FE1 Login functionality to an identity server
FE2 Load a target dataset with synthesized data to 'practice' the algorithm
FE3 Fill out a form to perform a SQL data request, authorization is seamlessly provided

3.3 System components

In this section, based on the requirements from chapter 3.2, choices in technology to implement the UNL scenarios are evaluated and selected.

3.3.1 Communication protocols

DYNAMOS is designed to handle external communication from data analysts solely via HTTP through the frontend. However, for internal communication between DYNAMOS components, different protocols can be used. AMQP stands out as a common asynchronous alternative to HTTP for this purpose, especially when services dispatch messages frequently that don't require immediate responses [24].

The orchestrator plays a pivotal role by relaying messages to both the policy enforcer and the distributed agents without necessarily expecting responses (requirements: ORCH5, ORCH6). Similarly, agents must communicate with each other under specific archetype scenarios (AGENT5). Given these communication needs, AMQP is a more fitting choice. Adopting AMQP also offers benefits like guaranteed message delivery and retry mechanisms. Figure 3.4 provides a visual representation of how communication is structured in DYNAMOS.

DYNAMOS employs RabbitMQ for asynchronous message passing. While there are numerous alternatives available, RabbitMQ stands out due to its ease of implementation, comprehensive feature set, and broad compatibility with various programming languages. Kafka is another prominent contender, and it's been employed in other POCs within the AMDEX project. However, Kafka is primarily tailored for high-throughput event streaming, rather than traditional queuing systems. Additionally, Kafka has the best support in the Java/Kotlin ecosystem, which DYNAMOS does not use, making RabbitMQ a more fitting choice.

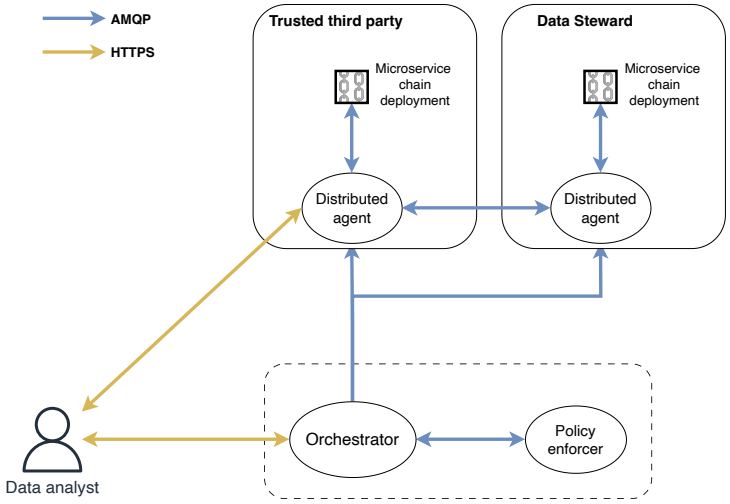

Figure 3.4: DYNAMOS communication protocols

3.3.2 Sidecar pattern

DYNAMOS employs a sidecar container to encapsulate its RabbitMQ implementation. Instead of interfacing directly with RabbitMQ, services communicate with this sidecar using gRPC. DYNAMOS defines a standard set of function definitions and messages in protocol buffers, to invoke the sidecar functions. This sidecar approach accomplishes several objectives:

- It establishes an internal communication protocol, as indicated in DYN2.
- As per DYN4, it mitigates the risk of technology lock-in. Notably, the individual services within DYNAMOS remain agnostic to the underlying communication system. While RabbitMQ serves as the current backbone, it can be seamlessly replaced, by for example Kafka, without requiring changes to the application code.
- With reference to JOB4, this design promotes language agnosticism. Irrespective of the programming language, all services utilize the same communication library. Thus, if a new data exchange service uses a different language, there's no need to integrate with a new library for backend messaging connectivity.

3.3.3 Container platform

Kubernetes is the chosen container platform for running DYNAMOS. As described in chapter 2.5, the amount of features in Kubernetes makes it a better choice than Docker Swarm. Furthermore, Kubernetes is the most used container platform, which makes it a good candidate for running an open-source platform to experiment with microservice chains. Finally, Kubernetes works with existing service meshes, which can be useful tools in self-adaptation scenarios [23, 37].

Kubernetes 'namespaces' will be used to simulate distributed environments, in which DYNAMOS has its own namespaces, and each 'data steward' will have a namespace. In reality, this is not a hard segregation without implementing specific network protections on a namespace, but it does come close to simulating a distributed system. Furthermore, it allows for easy local testing.

RedHat OpenShift is an enterprise solution, that can be installed locally as well. It builds on Kubernetes and thus has its own ecosystem, which makes the learning curve steeper. For this reason, OpenShift was not selected.

3.3.3.1 Service mesh

Service meshes offer a range of benefits, including monitoring capabilities, enhanced security (as referenced in DYN5), and potential assistance in self-adaptation scenarios [23, 37].

For its operations, DYNAMOS has chosen Linkerd[2]. This decision is primarily rooted in Linkerd's reputation for having a lightweight traffic proxy, in contrast to Istio, which tends to have greater overhead.

DYNAMOS's reliance on RabbitMQ for internal messaging, however, means that some of the built-in security features of a service mesh are bypassed. As a result, the primary advantage currently being used from the service mesh is the distributed tracing capability, facilitated by the Jaeger add-on.

3.3.3.2 Logging

DYNAMOS uses the trio of Promtail, Loki, and Grafana as its logging solution, primarily due to its lightweight nature. Unlike the ELK stack, which indexes each log message individually, Loki indexes the application logs based on the metadata from Kubernetes pods/environments.

3.3.3.3 Metrics

For metrics collection, 'Prometheus' is used, and the metrics are forwarded to Grafana for dashboarding. Since Prometheus is the de-facto standard in Kubernetes, no serious alternatives were considered.

3.3.3.4 Ingress

Nginx is installed as an ingress controller. Within the Linkerd service mesh environment some other common ingress controllers, like Traefik, required extra configuration. Nginx was easy to use and fits well within the service mesh.

[2] `https://linkerd.io/`

3.3.3.5 Distributed tracing

As previously mentioned, Jaeger is integrated into the Linkerd deployment to facilitate distributed tracing.

3.3.4 Programming language

DYNAMOS is written in the Go programming language [38] for several practical reasons.

First, much of the Kubernetes ecosystem is built using Go. As a result, the Go SDK is a mature option for interacting with Kubernetes APIs. Should there be a need to develop a custom Kubernetes scheduler or Custom Resource Definition (CRD) in the future, using Go would simplify the task.

Furthermore, Go inherently supports asynchronicity, which aligns seamlessly with DYNAMOS's distributed architecture, enabling efficient asynchronous message processing.

Additionally, Go's robust compatibility with gRPC serves as an added advantage, making it the language of choice for DYNAMOS.

3.3.5 Configuration management

Building upon the context set in chapter 2.2, DYNAMOS needs a distributed configuration tool to fulfill requirements ORCH1, ORCH4, DYN3, and AGENT3.

The choice for DYNAMOS was etcd, a coordination tool and key-value store. Acting as the central knowledge hub for the MAPE-K control loop mechanism [4], etcd uses a Go-based architecture, is used by Kubernetes for service naming and other functions, which makes ensures it has a mature and robust Go SDK, which fits DYNAMOS well. It is utilized for various use cases, such as:

- **Distributed agent availability.** Upon startup, an agent registers itself, allowing the system to discover available agents, their contact methods, and relevant metadata (ORCH1).
- **Microservice metadata.** Details on available microservices further discussed in chapter 4.3 (AGENT3)
- **Archetype configuration**, further discussed in chapter 4.3 (AGENT3)
- **Request types**, further discussed in chapter 4.3 (AGENT3)
- **Job information**, further discussed in chapter 4.3 (AGENT6)

ZooKeeper, arguably a better coordination system, leans more towards optimization for Java and C environments and isn't fundamentally designed as a key-value store like etcd. On the other hand, platforms like Apache Mesos and Consul represent comprehensive ecosystems. While powerful, their steep learning curve is excessive for the current DYNAMOS implementation.

3.4 Microservice chains

So far we have defined our scenario and identified the major components of our system. We have discussed how microservice chains are an integral part of this system. In this section, we define the language that we use to reason about these chains and look at what information is needed for an algorithm to create a chain.

3.4.1 Definition

A microservice chain is a directed acyclic graph (DAG) of microservices that together process a data-sharing request. A microservice chain has a minimum amount of *required* services to be able to process a request. A chain might have *optional* microservices that add extra functionality. Figure 3.5 shows a valid microservice chain for the SQL data request from our scenario.

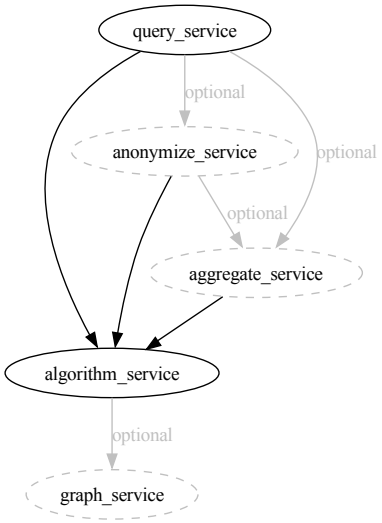

Figure 3.5: SQL-like data request microservice chain

3.4.2 Chain generation

When generating a microservice chain, requirement JOB5 states that depending on the archetype, microservices will run in different parts of the system. By analyzing figure 2.2 from chapter 2, we observe the following:

- The entity providing the software consistently sends it to the same destination as the data. Depicted by the green and red arrows in the figure.
- In DYNAMOS terminology, the destination of these arrows is referred to as the **'computeProvider'**. Subsequently, the origin of the red arrow, or the party sending the data, is named the **'dataProvider'**.
- The recipient of the results might differ per archetype. So this party will be denoted as the **'resultRecipient'** in DYNAMOS. For our scenario, however, this is always the requesting data analyst. Thus this field is ignored from now on.

To make this distinction clear in the DAG, microservices will be labeled with their intended functionality, this can be seen in the updated microservice DAG shown in figure 3.6.

Continuing, each data steward, or trusted third party, will then have a **'Role'** assigned, based on the archetype to determine the service it needs to run:

- **all:** creates all services labeled computeProvider and dataProvider
- **computeProvider:** only creates services labeled computeProvider
- **dataProvider:** only creates services labeled dataProvider

As an example, the below list shows possible outputs for generating a microservice chain for the *'sqlDataRequest'* type using two archetypes.

3.4.2.1 computeToData

The data steward has the role **'all'**

- Generated chain with two optional services:
  - query
  - anonymize
  - algorithm
  - graph

3.4.2.2 dataThroughTtp

The data steward has the role **'dataProvider'**
The third party has the role **'computeProvider'**

- Generated chain by the DataProvider:
  - query
  - anonymize
- Generated chain by the ComputeProvider:
  - aggregate
  - algorithm
  - graph

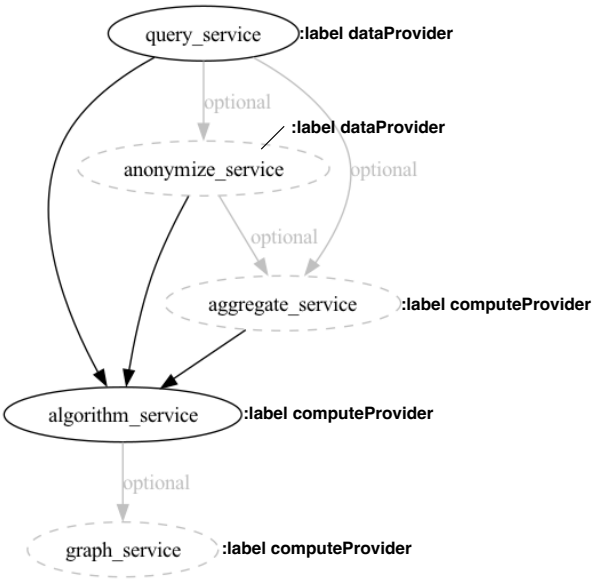

Figure 3.6: SQL-like data request labeled microservice chain

3.5 Chain job architectures

The next step is translating microservice chains into executable microservice jobs. Two main approaches for executing microservice chains have been identified. The first approach is having long-running, **persistent microservices** on the data steward nodes. The second approach is using a dedicated workflow tool to spawn **single-use, ephemeral jobs** for individual data-sharing requests. Both approaches are viable and would work depending on a data steward's or user's requirements.

Another consideration is how data moves between the microservices, especially with the potential for large transfers in DDM scenarios. Within the data stewards, only the distributed agent communicates directly with DYNAMOS via RabbitMQ, which leads to two main data transfer methods:

- The agent has to send the data request to the first microservice in the chain. The result needs to be received by the agent for further processing, to other agents, or back to the requester
- The microservices need to forward data to further services along the chain in a language-agnostic manner

DYNAMOS is flexible enough to handle both the ephemeral and persistent architectures which will be described in section 3.6. In this section, we discuss the communication methods and data transfer methods for both options, although, in the current implementation, only ephemeral jobs have been implemented.

Note, for the current DYNAMOS implementation, the same RabbitMQ instance is used by the data stewards, microservices, and the orchestrator. As just mentioned, in an actual scenario the distributed

agent would be the only entity fully connected to the general RabbitMQ server. The internal communication on the data steward might look different, for example:

- A local instance of RabbitMQ
- Connected to the central RabbitMQ, as a user with less access
- Different service bus or queueing system
- HTTP only

Extra modules to handle different formats would need to be created if this is a system requirement. For the rest of the thesis we assume there is a RabbitMQ system that can reach the local distributed agent.

3.5.1 Ephemeral jobs

The first microservice architecture involves ephemeral jobs. These jobs are created on demand, executed, and subsequently removed. The following advantages and disadvantages have been identified for this architecture.

3.5.1.1 Advantages

- Resource efficiency, resources are utilized only when a task is being executed.
- Security:
  - *Limited exposure time*, ephemeral jobs run for a short duration, which reduces the window of opportunity for attackers to exploit any potential vulnerabilities.
  - *Reduced attack surface*, as these jobs run for a specific task and then terminate, they don't keep unnecessary services or open ports, thus reducing the available attack surface.
  - *Fresh state*, with each execution of an ephemeral job, a clean state is provided. This ensures that any potential malware or rogue processes from a previous run don't carry over.
  - *Isolation*, ephemeral jobs are isolated from the host and other jobs. This can prevent the spread of a potential security breach from one job to another or to the host system.

3.5.1.2 Disadvantages

- *Startup overhead*, each job incurs an initialization overhead which can be costly for frequent, short-lived tasks.
- *Resource spikes*, concurrent initiation of many single-use jobs can lead to sudden, high demands on resources.
- *Complexity*, workflow tools introduce an additional layer of technology and potential failure points

3.5.1.3 Existing workflow tools

For executing microservice chains as ephemeral jobs in DYNAMOS, *Kubernetes jobs* are used. Primarily due to their compatibility with Go, which DYNAMOS is written in. Kubernetes jobs natively support failure handling and resource cleanup, making them not only efficient but also resilient.

Argo has built-in solutions for passing the output of a service to the next service, however, these are 'heavy' solutions requiring file I/O and external artifact repositories. A 'hello world' sample of Argo [39] with two services, takes around 20 seconds to run, see appendix B.1 for Argo output. This delay can be attributed to Argo's sequential execution model where it waits for one service to finish before launching the next, introducing a startup delay for each service in the chain. Argo does support parallel processing, but then the output of a previous service can not be used.

Apache Airflow has to be programmed in Python. This conflicts with DYNAMOS's Go language base, so was not seriously considered as an alternative.

3.5.1.4 Transferring data

For ephemeral jobs, transferring data through RabbitMQ is not viable. It would entail creating and deleting multiple queues per request, which adds overhead, and another easy point of failure. DYNAMOS already uses gRPC to transfer data to the sidecar, which brings us to one of the core ideas of this thesis

on how to transfer data. 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 (JOB4)
- Data stays in memory, it never goes 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 wire-protocol is more efficient in (un)marshalling objects compared to text-based methods (JSON or XML), resulting in less overhead
- 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

Finding 1: Transferring data within a Kubernetes job can be made theoretically efficient using a generic gRPC implementation

Propagation throughout DYNAMOS would require the results to be packed into an AMQ message, by the existing sidecar. See figure 3.7 for a schematic representation of this concept.

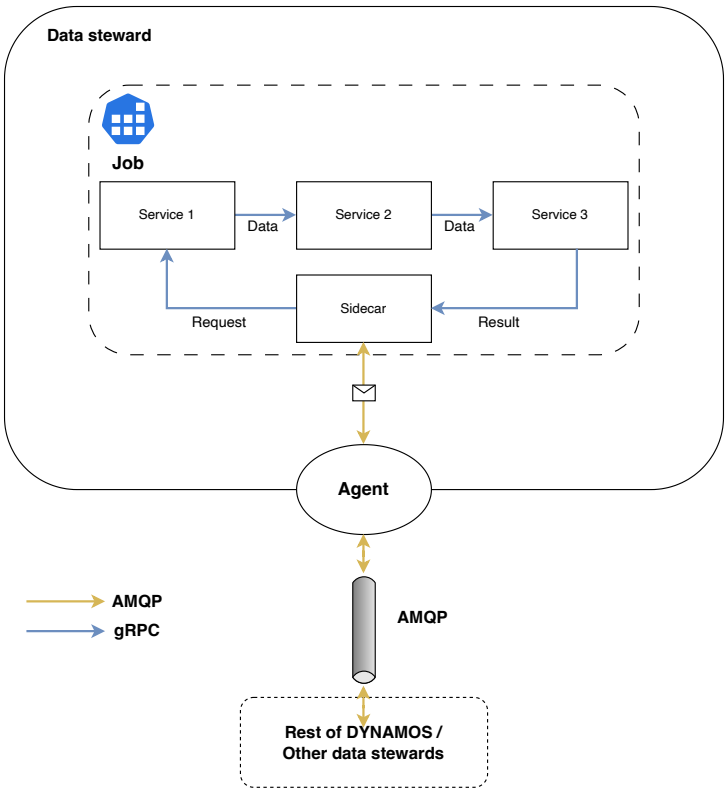

Figure 3.7: Ephemeral job I/O

3.5.2 Persistent microservices

In the second microservice architecture, microservices run persistently within the cluster. Each service listens to a queue for incoming messages. Messages will need to contain routing information to find their way to the correct microservices. A timer will be set to remove services that have been disused for a longer period of time.

3.5.2.1 Advantages

- *Readiness*, always on and ready to handle requests, minimizing cold start delays.

- *Consistent performance*, no need to bootstrap the environment for each request.

3.5.2.2 Disadvantages

- *Resource consumption*, services consume system resources during idle times
- *Potential security risks*: prolonged attack exposure, especially if misconfigured.
- *Lack of isolation*, users will share the same services

3.5.2.3 Transferring data

RabbitMQ can handle data transfers between microservices. Routing can be managed by including the microservice chain data in the message. But as noted in chapter 3.3.2, DYNAMOS uses a sidecar container to generalize AMQ communication. So, to connect with RabbitMQ, microservices need a sidecar, which adds an extra gRPC data transfer step. Figure 3.8 shows this setup.

Directly transferring data to the next service is an option but has drawbacks. Using RabbitMQ directly means every microservice needs a RabbitMQ client, adding complexity to data-exchange microservices. Sending data using gRPC, as it does to the sidecar pattern, is another option. But this misses out on RabbitMQ's asynchronous processing, making it harder to horizontally scale services and possibly creating bottlenecks.

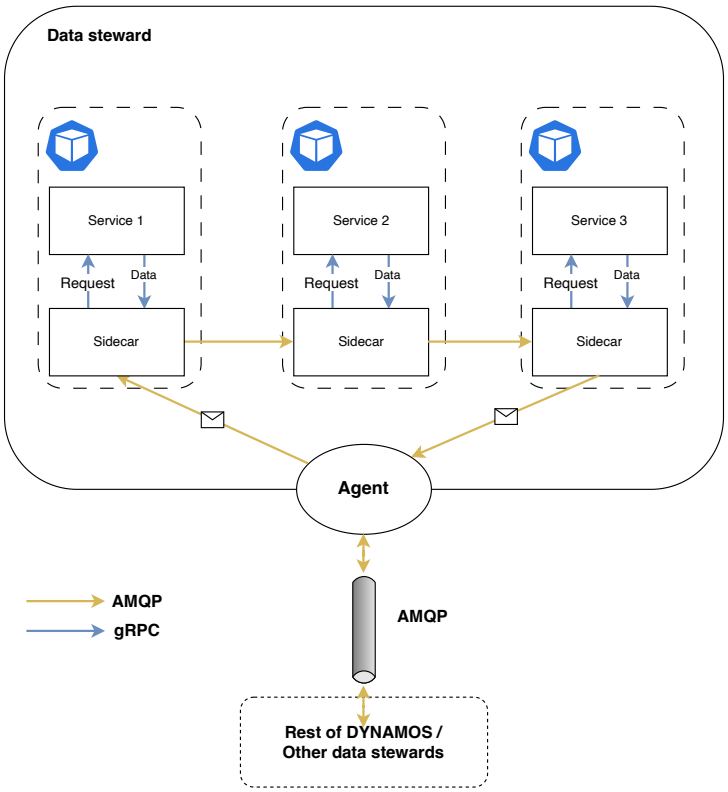

Figure 3.8: Persistent job I/O

3.6 Dynamic architectures

As outlined in Section 3.4, microservice chains are generated based on the selected archetype. Then in Section 3.5, we discussed two distinct microservice execution architectures: ephemeral and persistent. In this section, we'll delve into how to dynamically select archetypes and how distributed agents determine which microservice architecture to use. Crucially, both the microservice architecture and the archetype can vary per request, even coming from the same user.

3.6.1 Dynamic microservice architectures

The distributed agent is responsible for generating the microservice chain and either creating an ephemeral job or persistent microservice architecture, by default an ephemeral job will be created for a request. At the moment there are three sources of information that would allow the agent to create a persistent architecture instead.

- Enforced by the policy enforcer
- Configured by the data steward in the distributed data store
- Configured in the type of request, for certain request types specific architectures could be optimal

This allows DYNAMOS to be flexible enough to create different architectures, including architectures that have not been discussed so far. Furthermore, the distributed agents can function in a lightweight MAPE-K [4] loop to remove microservices after a certain period of disuse or upgrade microservices when new versions become available.

3.6.2 Dynamic archetypes

In our scenario, we are working with two archetypes, computeToData, and dataTroughTtp. In this section, we discuss how per request, the backend archetype could change, and DYNAMOS handles this. There exists an algorithm to map a type of data request to a best-fit archetype [3], however, for now, we assume this to be done when creating an agreement, thus the policy enforcer contains a set of best-fit archetypes for a request.

The following events are a few examples of an event that triggers a change in the chosen archetype.

- The agreement has been changed, a different archetype is allowed by the policy enforcer
- A flag has been added to the data request to force data aggregation in an 'aggregate' service
- One of the systems is under a high load, offload data processing to other parties

The first occurrence is event-based, and easily implemented based on the configuration set in the distributed data store, further explored in Chapter 4.5.

The second occurrence is actually not trivial, it requires extra information that that service is only allowed to run on a third party. At that point, if the correct archetype is not in place the orchestrator and policy enforcer will be involved to check if running that archetype on a trusted third party is allowed. The implication of this is that there needs to be an extra middleware, or gateway between the frontend and the data stewards, this is further discussed in Chapter 4.5.

The last occurrence requires the orchestrator to implement a MAPE-K loop on the system's extra-functional properties. In the future, we imagine an archetype selection algorithm that would pick a fitting archetype based on these and other possible events, the next paragraphs describe two such algorithms.

3.6.2.1 Extra-functional properties

The term extra-functional properties is broad and can mean anything from safety and security requirements to performance requirements, user experience, and more. Assuming that scalability, security, and resilience are all basic requirements of DYNAMOS, in this thesis we take a quick look at two properties, namely energy consumption, and user experience. In which user experience is seen as the time it takes to complete a request.

It is impossible to say anything about power consumption profiles without specific hardware-, software architectures, and testing. Literature however suggests that energy efficiency does not scale linearly with CPU usage [40] and in most cases, an optimum CPU load for energy consumption is around 80%. This means that if a data steward is under high load, a different archetype can be dynamically selected, if allowed by the policy enforcer, to adhere to the resource management property OS4.

Listing 3.1 shows a simple algorithm that DYNAMOS could use to select an archetype. This algorithm is based on the assumption that distributing the load more evenly trumps the cost of data transfer. The algorithm is not definitive and should be adapted after testing energy consumption on hardware that will be used in a DDM.

```
if dataProvider CPU and memory < 80 %
  if user does not need aggregation:
    computeToData > dataThroughTtp
  else:
    if computeProvider  CPU and memory < 80 %:
```

```
      dataThroughTtp > computeToData
    else:
      computeToData > dataThroughTtp
  goto end

if dataProvider CPU and memory > 80 %:
  if computeProvider  CPU and memory < 80 %:
    dataThroughTtp > computeToData
  else:
    computeToData > dataThroughTtp
  goto end

end:
distribute roles
```

Listing 3.1: Archetype selection algorithm

The user experience is more straightforward. The archetype that has the fewest data transfers, will have the least latency. In our scenario, the computeToData archetype will always outperform the dataThroughTtp archetype. By adding a *weight* to the archetype configuration, an archetype selection algorithm will include this in the final archetype selection.

Chapter 4

DYNAMOS: Implementation

As described in chapter 3.3 DYNAMOS runs on a Kubernetes platform, and the individual components are written in Go. The project's codebase is available on *GitHub*, and can be used as a reference with this chapter.

DYNAMOS is deployed in several layers depicted in figure 4.1, the **core** deploys all centralized components required for messaging, logging, tracing, metrics, and the distributed datastore.

The **orchestrator** layer is deployed after the core has been deployed, at which point DYNAMOS is ready to receive requests. How requests are processed is dependent on the data stewards that are online with a **distributed agent**. The **frontend** layer is used for login and sending requests to DYNAMOS, at the time of writing this part is not yet fully operational. The component deployment is managed through Helm 'the Kubernetes package manager'. The DYNAMOS images are built locally with the help of Makefiles. Exact installation and build instructions can be found in Appendix A.

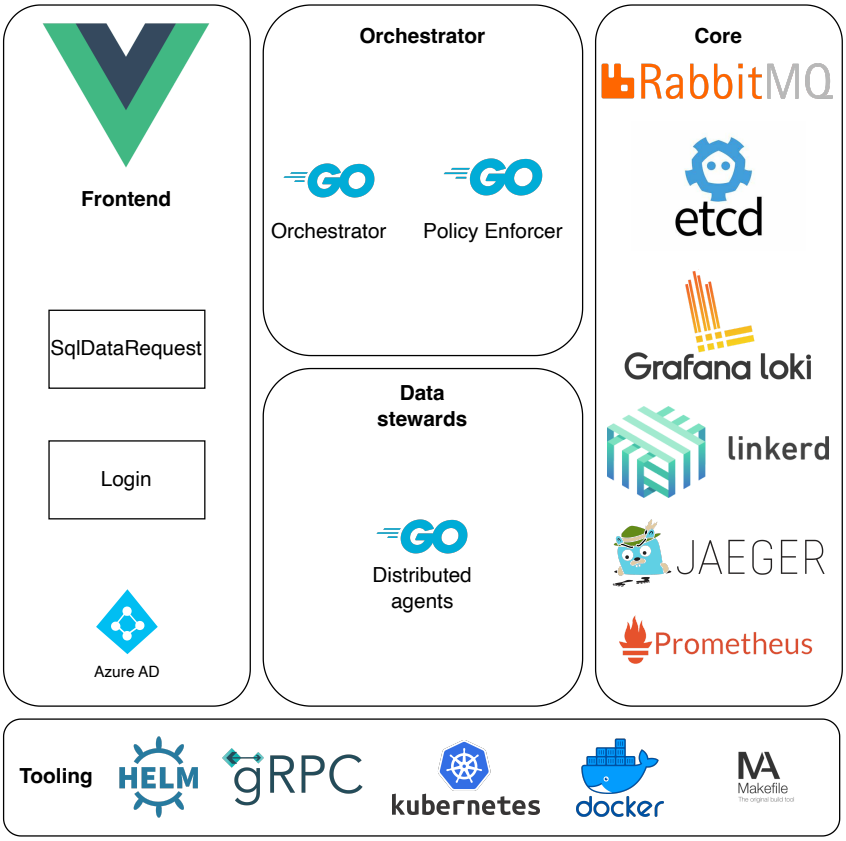

Figure 4.1: Overview DYNAMOS components

We start this chapter with a generic architecture overview, then go into more depth about how etcd is used within DYNAMOS. Subsequently, how to distributed agents generate microservice chains, and how

persistent jobs and ephemeral jobs are created. Finally, the chapter touches upon the gRPC messaging framework, distributed tracing, and the DYNAMOS API.

4.1 Architecture

DYNAMOS follows the client-server architectural pattern. In which the client, the data analyst, requests access to data resources, and sends data-sharing requests to these resources. In Chapter 3, several figures have already shown this design in several levels of abstraction. Figure 4.2, shows a similar design, now including the message definitions.

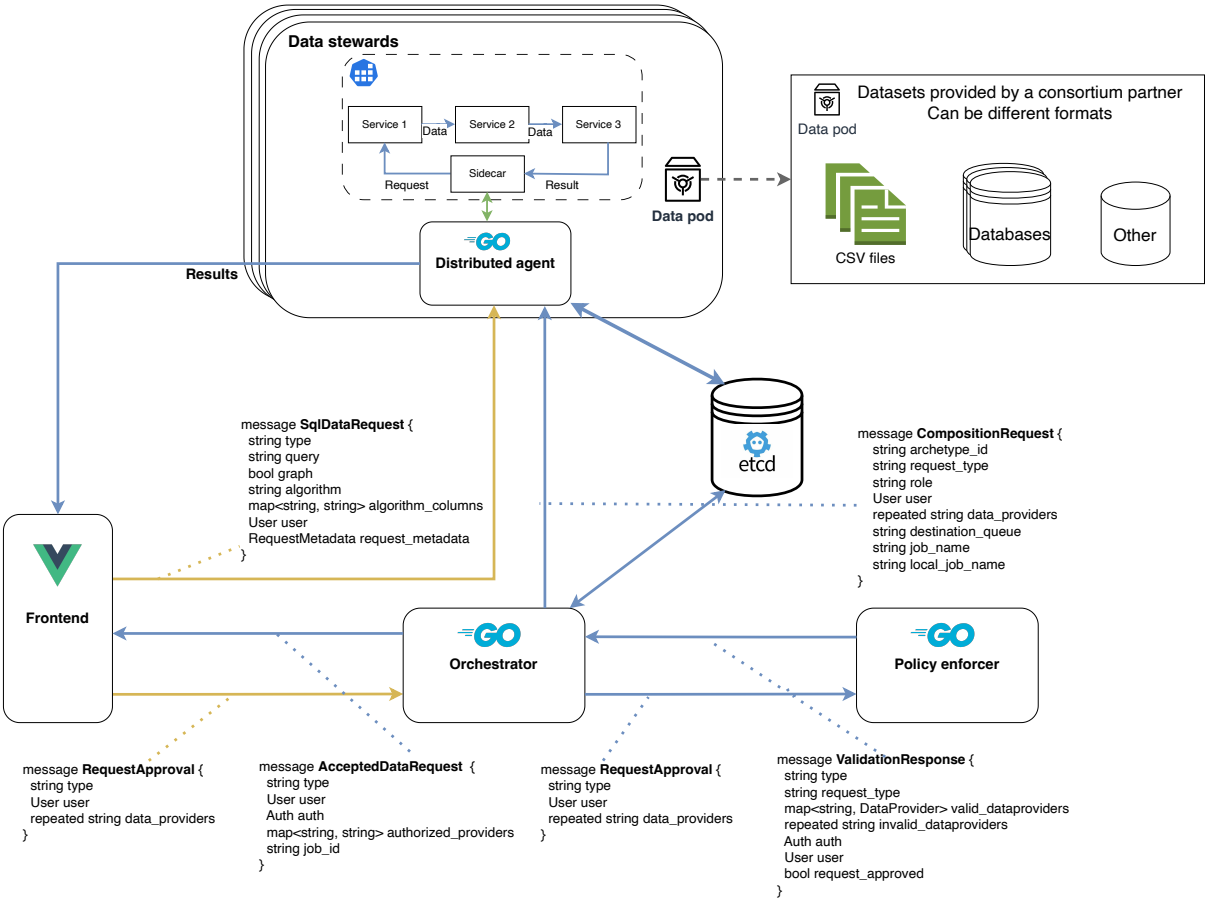

Figure 4.2: DYNAMOS architecture

4.2 Configuration management

As briefly discussed in chapter 3.3.5, DYNAMOS makes use of 'etcd' as a distributed datastore. Etcd is used for static configuration, reactive configuration management, and health checks.

4.2.1 Static configuration store

To simulate the work of the policy enforcer the configuration/etcd_launch_files directory contain JSON files that represent static configuration, loaded by the orchestrator on startup. The use of this static configuration is explained in chapter 4.3.

File	Goal
agreements.json	Simulated agreements between consortium partners and a data analyst
archetype.json	Archetype configuration
datasets.json	Metadata regarding existing datasets
microservices.json	Microservice metadata, label and allowed outputs
requesttype.json	Configuration regarding existing request types (sqlDataRequest)

Table 4.1: Static configuration files

4.2.2 Reactive configuration

When a user requests a transaction, a job is created with a single queue to process messages for this job. For as long as the user has valid access, the same queue will be re-used for future jobs, to save on the overhead of constantly creating and re-creating queues. At creation information of the queue will be stored in etcd, with a timeout of 10 minutes. Every time the user sends a request, the lease on the etcd key will be renewed for another 10 minutes. After the timer expires, however, the user has been inactive too long and the queue and related job information will be deleted, requiring the user to get new approval. In the future, this would work differently based on the expiry of user access tokens.

A similar system can be implemented for persistent microservice architectures, to remove services that are inactive for a longer period of time.

4.2.3 Health checks

To determine distributed agent availability, an agent registers itself on startup with the metadata from the below table. When a user request comes in, the request is checked against the existing policy to determine allowed data stewards. After this, it is matched again against actual available agents. If an agent goes offline for some reason the etcd key is removed, this could be extended in the future to send out alerts when this happens.

Metadata	Note
Name	Agent name
RoutingKey	Used by the messaging solution to identify this agent
DNS	HTTP endpoint of this agent

Table 4.2: Agent information

4.2.4 Etcd keys

Key path	Contains
$/agents/ < Agentname >$	Agent configuration
$/agents/jobs/ < Agent\ name > / < User\ name > / < Job\ name >$	Composition request data
$/agents/jobs/ < Agent\ name > /queueInfo/ < Job\ name >$	Queue attached to a job
$/archetypes/ < Archetype\ name >$	Archetype configuration
$/datasets/ < dataset\ name >$	Dataset metadata
$/microservices/ < Microservice\ name > /chainMetadata$	Microservice metadata
$/policyEnforcer/agreements/ < data\ steward\ name >$	Agreements for data-sharing
$/requestTypes/ < request\ type\ name >$	Request type information

Table 4.3: Etcd keys

4.3 Microservice chains

We discussed the concept of creating microservice chains in chapter 3.4. In this section, we look at the exact configuration and data required to actually create the chains. all data provided below is stored in etcd.

4.3.1 Defining request types

Listing 4.1 shows the type of request that is used in our scenario, "sqlDataRequest". Which will be used for querying a SQL-like data store, and allowing an analysis of the queried data. In the future other types can be introduced to allow different data-sharing scenarios, for example, "federatedLearningRequest".

As discussed in the scenario chapter 3.1, a request type can have required services and optional services.

```
{
    "type": "sqlDataRequest",
    "requiredServices": ["query", "algorithm"],
    "optionalServices": ["aggregate", "anonymize", "graph"]
}
```

Listing 4.1: Request Types

4.3.2 Defining microservices

Listing 4.2 shows a list of all available microservices. The 'allowedOutputs' field denotes valid output microservices for each service, in other words, it lists the services that are equipped to handle the output of this service.

The 'label' field is used to indicate the *role* of the microservice in a DDM archetype, as discussed in chapter 3.4.

```
[
    {
        "name": "query",
        "label": "dataProvider",
        "allowedOutputs": [
            "algorithm",
            "anonymize"
        ]
    },
    {
        "name": "anonymize",
        "label": "dataProvider",
        "allowedOutputs": [
            "algorithm"
        ]
    },
    {
        "name": "algorithm",
        "label": "computeProvider",
        "allowedOutputs": [
            "graph"
        ]
    },
    {
        "name": "graph",
        "label": "computeProvider",
        "allowedOutputs": []
    },
    {
        "name": "aggregate",
        "label": "thirdParty",
        "allowedOutputs": ["algorithm"]
    }
]
```

Listing 4.2: Microservice metadata

4.3.3 Defining archetypes

The only relevant information required is who the 'computeProvider' in an archetype is. Suppose that one is the same as the dataProvider (computeToData). This means all services can be scheduled with the same data steward. If there is another party, it can be assumed that the policy enforcer declares who that third party is. Listing 4.3 shows the simple configuration required as to what an archetype is.

The 'weight' property, is not currently implemented. As discussed in Chapter 3.6, it would be used to pick one archetype above the other, when all other archetype selection algorithms don't have a specific preference, in which a lower weight constitutes higher preference. The reason for computeToData to be picked above dataThroughTtp, is that its response is faster due to having fewer data transfer points.

```
[
    {
        "name": "computeToData",
        "computeProvider": "dataProvider",
        "resultRecipient": "requestor",
        "weight": 100
    },
    {
        "name": "dataThroughTtp",
        "computeProvider": "other",
        "resultRecipient": "requestor",
        "weight": 200
    }
]
```

Listing 4.3: Archetype config

4.3.4 Generating the chain

Based on the archetype, the orchestrator creates one of three roles for all involved parties and sends this, together with relevant user and job information to the distributed agents, this is called a **'compositionRequest'**. On arrival of a compositionRequest, the distributed agent saves all information regarding the job, role, request, and user in etcd.

4.3.4.1 Ephemeral jobs

The distributed agent, having a role, will at the moment the request comes in check which required and optional microservices are needed, and generate a DAG corresponding to the request type by using a 'topological sort' algorithm [41].

It is important to note that a topological sort (and DYNAMOS) can not handle cyclical references. This means that a microservice chain developer using DYNAMOS will need to make definitive choices on a valid order of services. This will be assisted by validation functions when uploading or updating microservice metadata and offering a graph generation service (like figure 3.5) for visually validating microservice chains.

Appendix B, shows how a Kubernetes job is created in Go.

4.3.4.2 Persistent jobs

The moment a composition request comes in, the distributed agent checks if the microservice exists. If not, the microservices of this request type are created. Timeouts on each service will be set to promote clean-up if services are not used for a longer period of time.

4.4 Ephemeral job data transfers

In section 3.5, it was discussed how for persistent jobs, routing information could be generated upfront and passed with each request. How the generic gRPC implementation for ephemeral jobs works is discussed in this section. As an example, we take the dataThroughTtp archetype, in which the data steward with the 'dataProvider' role will deploy a microservice chain containing the 'query' and 'anonymize' services.

The microservice chain, always ordered in line with its DAG, is passed into the deployment function. In our example that is first 'query' and then 'anonymize' service. Each microservice will get a designation

'first', *'last'*, or *'both'*, and the address of the sidecar. The first service will initialize the sidecar and wait for an incoming message, process it, and send it to the next link. The last service will connect back to the sidecar to send back the results. The default gRPC port of the sidecar container is set at 50051, so the communication flow of this chain will look like figure 4.3. The algorithm is shown in listing 4.4.

The demo microservices are designed to exit after successfully processing a single message. This could be different for more complex scenarios, for example, the 'aggregation' service would need to process several messages. Messages contain a unique ID for every separate job, it would be possible to check if a service is receiving the correct messages and quit after a specified number.

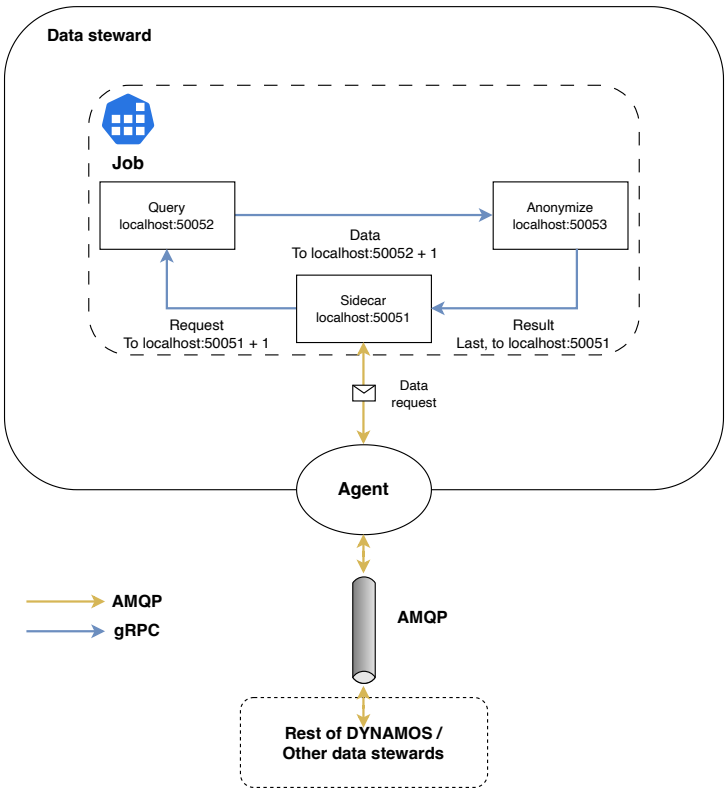

Figure 4.3: Example communication flow

```
firstPort = SidecarPort + 1
first = true
last = false
numberOfMicroservice = length(microserviceChain)

for i, microservice in microserviceChain {
    designatedGrpcPort = firstPort + i

    if i == numberOfMicroservice-1 {
        lastService = true
    }

    container := v1.Container{
        Name:            microservice.Name,
        Image:           microservice.Name + :latest
        Env: []v1.EnvVar{
            {Name: "DESIGNATED_GRPC_PORT", Value: designatedGrpcPort},
            {Name: "FIRST", Value: first},
            {Name: "LAST", Value: last},
            {Name: "JOB_NAME", Value: jobName},
            {Name: "SIDECAR_PORT", Value: SidecarPort},
            {Name: "OC_AGENT_HOST", Value: tracingHost},
        },
```

```
24      }
25      job.Spec.Template.Spec.Containers = append(job.Spec.Template.Spec.
            Containers, container)
26      firstService = false
27  }
28  job.Spec.Template.Spec.Containers = append(job.Spec.Template.Spec.Containers,
        addSidecar())
```

Listing 4.4: Adding microservices to the job

4.4.1 gRPC message size

The default max size of a gRPC method is around 4MB, making the max query size to be around the 30.000 records used in the experiments of Chapter 5. The current implementation of gRPC is a 'unary RPC', the client sends a single request to the server and gets a single response back. Unary RPCs set to unlimited size, but can be inefficient and potentially problematic for very large datasets because they require enough memory to hold the entire message on both the client and the server. Streaming data is a possible solution.

4.4.1.1 Streaming messaging

In server-side streaming, the client sends a request to the server and gets a stream to read a sequence of messages back. The client reads from the returned stream until there are no more messages. This pattern is actually used withing DYNAMOS for a application to receive RabbitMQ message from its sidecar.

In client-side streaming, the client writes a sequence of messages and sends them to the server, again using a provided stream. Once the client has finished writing the messages, it waits for the server to read them all and return its response. There's also bidirectional streaming where both sides send messages using a read-write stream. Streaming RPCs are more efficient for large datasets because they break the data up into multiple smaller messages. This however has not been tested yet with regards to passing on SQL data.

4.4.2 Microservice chain libraries

To interface with this gRPC flow, a small library would need to be written for each language that is used in the data exchange microservices. DYNAMOS has a library in *Go* and in *Python*, the one in Go is more developed at this moment. This library should do the following:

- Create gRPC connections based on the information given in listing 4.4
- Start a gRPC server to receive messages from the previous link in the chain
- Exit the service gracefully after certain conditions are met
- Send messages to the next link in the microservice chain or the sidecar

Listing 4.5 shows in pseudocode how that would look.

```
1   func main() {
2       // Initialize some logger package that writes to STDOUT
3       logger = lib.InitLogger()
4
5       // Sets up the microservice so that it knows how to receive and send
            messages
6       configuration = lib.InitMicroservice()
7
8       // Start consuming messages
9       lib.startConsume(configuration, messageHandler)
10  }
11
12  func messageHandler(ctx context.Context, grpcMsg *pb.
        MicroserviceCommunication) {
13
14      switch grpcMsg.Type {
```

```
15
16          case "generic":
17                  // Handle generic requests
18          case "sqlDataRequest":
19                  // Handle sqlDataRequest
20              }
21      }
```

Listing 4.5: Template microservices need to implement

4.4.3 Detailed ephemeral job examples

In the previous sections, we discussed how ephemeral jobs are created and how the data flows through the services. In this section, we go through a step-by-step sample of the actions an agent takes for each archetype.

As long as the data analyst has valid access tokens the same job can be executed indefinitely. For each new job, a new microservice chain will be generated and deployed, which will re-use the created input queue for that job. After a time-out of disuse, the queue will also be deleted.

4.4.3.1 ComputeToData scenario

In this scenario, the data steward handles every part of the microservice chain. Figure 4.4 shows numbered events and colored arrows of messages arriving at the agent, the numbers correlate to the following steps:

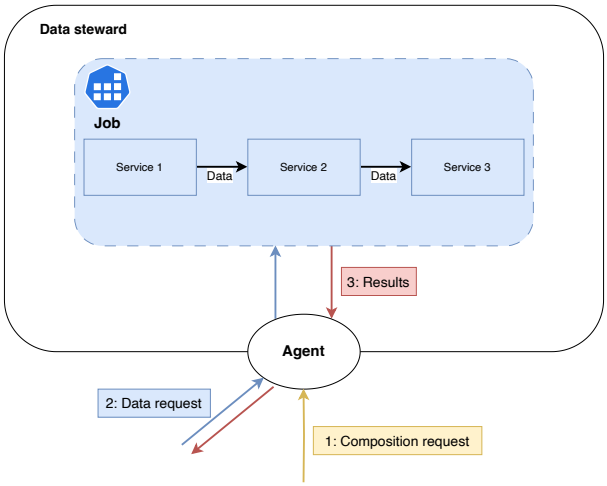

Figure 4.4: Compute to data, agent actions

1: composition request received, the role for this job is 'all'.

- Generate a unique name based on the given job name
- Register composition request for later use
- Create a queue with that unique name, as input for the jobs
- Wait for incoming HTTP data requests from the data analyst

2: Data request received.

- Look up the matching job for this user
- Generate and deploy microservice chain, possibly including optional services from the HTTP request
- Put the request into the queue for this job

3: Results received.

- Return results to the data analyst

4.4.3.2 DataThroughTtp

In this scenario, the trusted third party receives the data request and forwards this to the related data stewards. The colours in figure 4.5 show based on what request the job is created. The below tables show which actions each agent takes on receiving a request.

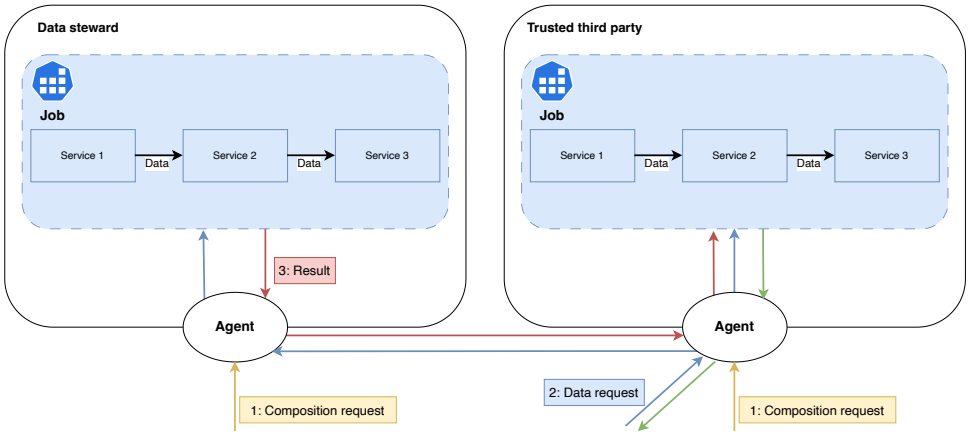

Figure 4.5: Data through TTP, agent actions

	Data steward (DataProvider)	Third party (ComputeProvider)
1: Composition request		
Generate a unique name based on the given job name	X	X
Register composition request for later use	X	X
Create a queue with that unique name, as input for the jobs	X	X
Wait for incoming HTTP data requests from the data analyst		X
Wait for incoming requests from third party	X	

	Data steward (DataProvider)	Third party (ComputeProvider)
2: Data Request		
Generate and deploy microservice chain, including optional services from the HTTP request	X	X
Forward request to other data stewards		X
Wait for results		X
Put the request into the queue for this job	X	

	Data steward (DataProvider)	Third party (ComputeProvider)
3: Results		
Send result to ComputeProvider	X	
Put the request into the queue for this job		X
After processing, return results to data analyst		X

4.5 Dynamic Archetypes

In Chapter 3.6, we discussed three examples of when an archetype could change.

- The agreement has been changed, a different archetype is allowed by the policy enforcer
- A flag has been added to the data request to force data aggregation in an 'aggregate' service
- One of the systems is under a high load, offload data processing to other parties

In this section, we first have a look at how an agreement is processed in DYNAMOS to come to a valid archetype. Then we discuss how each change event can be handled, and show an implementation of the first.

4.5.1 Processing Policy

As discussed in Chapter 2.4, for DYNAMOS we have stubbed a 'policy enforcer'. Listing 4.6 how this is configured. There are three universities, of which two have an agreement with a data analyst. Each university lists:

- Its name
- The compute providers (third parties) it trusts
- The data-sharing archetypes it allows relations to work with
- Each relations with this university show:
  - Name
  - ID
  - The request types allowed for this user
  - The datasets allowed for this user
  - Archetypes allowed for this user
  - Compute providers (third parties), trusted by this user

```
[
    {
        "name": "VU",
        "relations": {
            "jorrit.stutterheim@cloudnation.nl" : {
                "ID" : "GUID",
                "requestTypes" : ["sqlDataRequest"],
                "dataSets" : ["wageGap"],
                "allowedArchetypes" : ["dataThroughTtp", "computeToData"],
                "allowedComputeProviders" : ["SURF"]
            }
        },
        "computeProviders" : ["SURF", "otherCompany"],
        "archetypes" : ["computeToData", "dataThroughTtp",  "reproducableScience"]
    },
    {
        "name": "UVA",
        "relations": {
            "jorrit.stutterheim@cloudnation.nl" : {
                "ID" : "GUID",
                "requestTypes" : ["sqlDataRequest"],
                "dataSets" : ["wageGap"],
                "allowedArchetypes" : ["dataThroughTtp"],
                "allowedComputeProviders" : ["SURF"]
            }
        },
        "computeProviders" : ["SURF", "otherCompany"],
        "archetypes" : ["computeToData", "dataThroughTtp",  "reproducableScience"]
    },
    {
        "name": "RUG",
        "relations": {
        }
    }
]
```

Listing 4.6: Agreement config

When a data analyst requests approval, the policy enforcer checks which universities the data analyst has agreements with. And returns all valid data providers, with a list of valid archetypes and trusted third parties.

The orchestrator then checks which distributed agents are actually online, to make a final choice of archetype and trusted third party. At this moment, the orchestrator will simply pick the first matching archetype, and the trusted third party matched in all the agreements. In listing 4.6 this would mean it picks 'dataThroughTtp' with 'Surf' as compute provider.

In future iterations of DYNAMOS, this would be improved with the archetype selection algorithms discussed in Chapter 3.6. Furthermore, it would also allow different archetypes on different data stewards, in the above example it could be fine to have the VU operate as computeToData and the UVA as dataThroughTtp and if the data doesn't need to aggregate, different third parties could be used as well.

4.5.2 Evolving Agreement

Agreements can be changed through the API (4.9). The workflow that is triggered, checks if 'relations' in the changed agreement have active jobs. Active jobs are stored in etcd as a compositionRequest:

```
// Key: /agents/jobs/SURF/jorrit.stutterheim@cloudnation.nl/jorrit-stutterheim-43ea82da
{
    "archetype_id": "dataThroughTtp",
    "request_type": "sqlDataRequest",
    "role": "computeProvider",
    "user": {
        "id": "12324",
        "user_name": "jorrit.stutterheim@cloudnation.nl"
    },
    "data_providers": [
        "UVA"
    ],
    "destination_queue": "SURF-in",
    "job_name": "jorrit-stutterheim-43ea82da",
    "local_job_name": "jorrit-stutterheim-43ea82dasurf1"
}
```

If there are active jobs for a user in the changed agreement, an algorithm starts to determine if the current archetype is still allowed, and the best choice. If the archetype changes, the fields 'archetype_id' and 'role' from listing 4.5.2 will be adjusted to reflect the new archetype. Schematically this process is shown in figure 4.6.

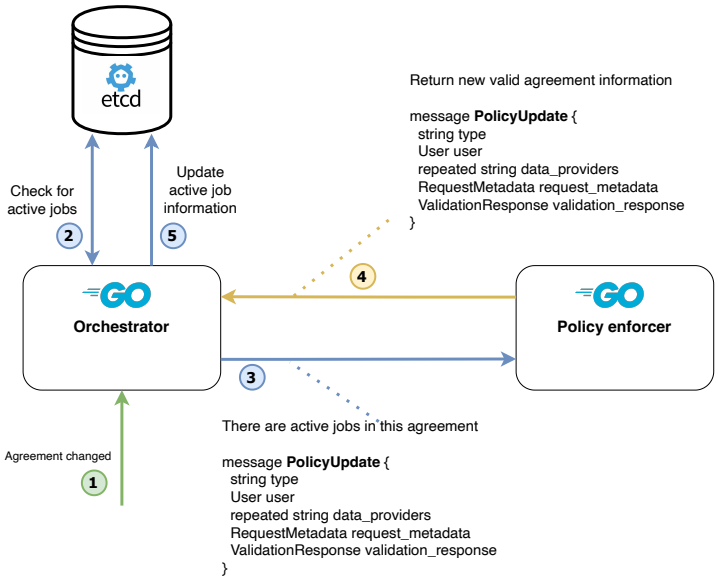

Figure 4.6: Workflow on policy change

4.5.3 Aggregate flag

As mentioned in Chapter 3.6, allowing an aggregate flag is not trivial in the current implementation of DYNAMOS. This is due to the fact that at this moment the URL is different per archetype. If the computeToData archetype is active, the user is required to send a request to each data steward. If the dataThroughTtp archetype is active, the request should be sent to the trusted third party.

For the previous use case, this architecture could still be workable as long as DYNAMOS can stream events to active users, and the frontend client can pick the appropriate backend targets based on these events. However, in this case, where the user actually forces an archetype change, a different approach is required.

An obvious solution is introducing an API gateway, which could be off-the-shelf or custom, depending on the exact requirements. In this architecture, our ingress NGINX, would be configured to forward all traffic to this gateway service and be used only for SSL termination, rate limiting, and other functionalities it excels at.

Depending on the requirements, the gateway can, aggregate responses, modify requests or responses, handle authentication, and dynamically choose backends. Or in this case, ensure the archetype is checked, modified, and the request is routed correctly.

In this thesis, we don't provide a full design of this gateway, but accept that this would be required for future implementations of DYNAMOS. Even if user-inspired archetype changes will not be allowed, an API gateway still provides a lot of useful features that can be exploited.

4.5.4 System load

We assume that the distributed agent can query the local data steward environment for metrics. The data steward should configure the agent with a *threshold*, saying that if CPU and memory are above a certain percentage, compute operation need to be offloaded. The agent stores this threshold and metrics in etcd, the orchestrator sets watches on all the metrics to take action when a threshold is passed[1].

As the selection algorithm in Chapter 3.6 describes, new jobs will now choose archetypes that have the least impact on that agent. However, as described in the previous section and similar to the flow in figure 4.6, the orchestrator could also start adjusting active jobs.

The main takeaway here should be that DYNAMOS can change both new and active jobs, and thus act on metrics to change a server load by adhering to different archetypes. This type of load balancing on metrics, however, is quite complex and can be a study in itself. Questions raised for example when thinking about this are:

- How many active jobs should be changed, and for how long?
- Are there too many users, or are there a few users with very heavy jobs?
- If the policy does not allow new users to switch archetypes, do we start rate limiting or something else?
- What do we do if the third party is overloaded as well?

In this thesis, we don't intend to solve this problem but show the capabilities of how DYNAMOS could be extended to implement these algorithms.

4.6 Messaging

As discussed in chapter 3.3, DYNAMOS abstracts away its messaging infrastructure through the use of a sidecar and gRPC. The messages and functions to be used by gRPC are defined in '.proto' files. Exact message definitions can be found in the GitHub repository or in appendix C.2. Two generic message definitions stand out and need some extra context, namely 'SideCarMessage' and 'MicroserviceCommunication'.

[1] In practice, it would be advised to create a separate service, orchestrator-metrics, for example, to manage this specific task

4.6.1 Generic messages

4.6.1.1 Sidecar message

The sidecar of each DYNAMOS component has a constant connection with the 'main' container to stream incoming messages from the messaging system into the container. The SideCarMessage is the standard message type that is used throughout DYNAMOS to communicate down that stream. The type of the SideCarMessage body is 'google.protobuf.Any'. An 'Any' type can hold any valid protocol message, the advantage being that no separate function definitions need to be made for each message that streams for the sidecar into the system. Listing 4.7 illustrates this.

```
//Current implementation
service Sidecar {
  rpc Consume(ConsumeRequest) returns (stream SideCarMessage) {}
}
message SideCarMessage {
  string type = 1;
  google.protobuf.Any body = 2;
  map<string, bytes> traces = 3;
}

...

// Sidecar packs an incoming message:
func (s *server) SendRequestApproval(ctx context.Context, in *pb.
    RequestApproval) (*emptypb.Empty, error) {
    message := amqp.Publishing{
        Body: proto.Marshal(in),
        Type: "requestApproval",
    }
        stream.send(message)
    ...
}

// Main container unpacks this message:

func handleIncomingMessages(ctx context.Context, grpcMsg *pb.SideCarMessage)
    error {
    switch grpcMsg.Type {
    case "requestApproval":
        var requestApproval pb.RequestApproval
        grpcMsg.Body.UnmarshalTo(&requestApproval)
            // Handle requestApproval message
        ...
        }
}

// If 'Any' was not used like this then:
// Separate streaming functions for each message type would need to be
    implemented
service Sidecar {
  rpc ConsumeValidation(ConsumeRequest) returns (stream ValidationMessage) {}
  rpc ConsumeRequestApproval(ConsumeRequest) returns (stream RequestApproval)
      {}
  ...
}
```

Listing 4.7: Streaming side car messages

4.6.1.2 Microservice communication

A second often-used message is 'microServiceCommunication'. Since all microservices in a microservice chain communicate data through the use of gRPC, a default message that is passed to, through, and

from the chain makes this a uniform message flow. The data field is of type 'google.protobuf.Struct' this is basically a flexible map where the keys are strings and the values can be of any type, in a way this mimics a JSON-like data structure. Furthermore, it can contain metadata about the data, the original users' request, and routing data.

```
service Microservice {
  rpc SendData(MicroserviceCommunication) returns (google.protobuf.Empty) {}
  rpc SendMicroserviceComm(MicroserviceCommunication) returns (google.protobuf
      .Empty) {}
}

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;
  RequestMetada request_metada = 6;
  map<string, bytes> traces = 7;
  bytes result = 8;
  repeated string routing_data = 9; // To be used for persistent jobs
}
```

Listing 4.8: Microservice Communication messages

4.7 Transferring generic data

In the previous section, we highlighted that DYNAMOS messages utilize google.protobuf.Struct within the microServiceCommunication message to transfer data. An alternate method could involve encoding this data into other structured formats such as JSON. This section delves into the advantages and disadvantages of both methods, providing insights into how subsequent microservices can process data. This challenge, however, is not the main focus of this thesis, so apart from generic observations it will not delve into too much detail.

The benefits and limitations of the Struct type are as follows:

Advantages

- Items retains type information
- Efficient marshalling and unmarshalling operations

Disadvantages

- Due to type information retention the messages increase in size
- I requires custom logic to pack a message and retain its type information as close as possible

On the other hand, using JSON offers the following:

Advantages

- Universal and familiar, making it easier to integrate and work within various platforms.
- Typically results in smaller message sizes, especially when representing complex data structures, leading to potentially faster transmissions.

Disadvantages

- Offers less precision in type information retention, which may require additional validation or interpretation logic

Both methods would work in the current DYNAMOS implementation. More research focusing on this specific issue would be needed to make a definitive decision on ways to transfer generic SQL data, this would heavily influence how data-sharing microservices are designed. We stuck with the Struct message type for its type safety properties. In the next paragraphs, we will quickly show why message sizes differ between JSON and Struct, and how processing would work.

4.7.0.1 Message size

Listing 4.9 showcases the reason for the size difference.

```
// A single field of the queried data is represented on the ‘google.protobuf.
    Struct’ like this:
"Aadnstvb": {
  "listValue": {
    "values": [
      {
        "stringValue": "V"
      }
    ]
  }
}

// In JSON this can be represented like this:
"Aadnstvb": ["V"]
```

Listing 4.9: Struct vs. JSON encoding

4.7.0.2 Processing

The code in listing 4.10 illustrates the process of extracting string values from two distinct data formats: a Protocol Buffers Struct and a JSON string. The example mainly shows that passing data as JSON would involve a lot more type checking and error handling.

```
func convertAllDataFromStruct(data *structpb.Struct) {

//-----Struct:
// Initialize variables to store keys, values and maximum length
keys := make([]string, 0)
allValues := make([][]string, 0)
maxLength := 0

// Loop over fields in the structpb
for key, value := range data.GetFields() {
    stringValues := value.GetListValue().GetValues()
    // Check if there are any values to process
    if len(stringValues) > 0 {
        keys = append(keys, key)
        rowValues := make([]string, len(stringValues))
        // Extract string values
        for i, v := range stringValues {
            rowValues[i] = v.GetStringValue()
        }
        allValues = append(allValues, rowValues)
        // Update maxLength if current row is longer
        if len(rowValues) > maxLength {
            maxLength = len(rowValues)
        }
    }
}
..

}

func convertAllDataFromJSON(data []byte) {

//-----JSON:
// Unmarshall JSON:
jsonData := make(map[string]interface{})
err := json.Unmarshal(data, &jsonData)
if err != nil {
    fmt.Printf("Error while unmarshalling JSON: %v\n", err)
    return
}
```

```
41
42 // Initialize variables
43 keys := make([]string, 0)
44 allValues := make([][]string, 0)
45 maxLength := 0
46
47 // Loop over key-value pairs in the JSON data
48 for key, rawValue := range jsonData {
49     values, ok := rawValue.([]interface{})
50     if !ok {
51         fmt.Printf("Expected a list for key %s but got a different type\n", key
               )
52         continue
53     }
54     if len(values) > 0 {
55         keys = append(keys, key)
56         rowValues := make([]string, len(values))
57         for i, v := range values {
58             // Convert the interface value to string
59             strValue, ok := v.(string)
60             if !ok {
61                 fmt.Printf("Expected a string value in list for key %s but got
                       a different type\n", key)
62                 continue
63             }
64             rowValues[i] = strValue
65         }
66         allValues = append(allValues, rowValues)
67         if len(rowValues) > maxLength {
68             maxLength = len(rowValues)
69         }
70     }
71 }
72 ..
73 }
```

Listing 4.10: Struct vs. JSON unpacking

4.8 Distributed tracing

As discussed in chapter 2.5.3.2, distributing tracing is extremely important for debugging a distributed application. Furthermore, it is used to get exact figures on how long transferring data takes within DYNAMOS. However, it is also complex and requires the application code to properly initiate, propagate and handle trace context. To try and approach tracing more as an 'out-of-the-box' feature, DYNAMOS leverages the Linkerd (service mesh) Jaeger extension. This extension adds the following three services to the cluster.

- cr.l5d.io/linkerd/jaeger-webhook → Webhook for auto-forwarding Linkerd tracers
- otel/opentelemetry-collector → OpenTelemetry collect for collecting, processing, and forwarding trace data
- jaegertracing/all-in-one → Jaeger tracing backend and front-end UI

With these services running, applications can be instrumented as shown in listing 4.11, to do this properly already can be a time-consuming task. However, instrumenting microservices of an ephemeral job revealed additional challenges.

4.8.1 Conflicting libraries

Linkerd, recommends the use of the 'OpenCensus' tracing library [42]. In the future, however, this library is deprecated in favor of the 'OpenTelemetry' library. In DYNAMOS Go services this is no issue, OpenCensus works fine and traces are propagated in a binary format. In our Python microservices the

OpenCensus library does not seem to work. Pythons' OpenTelemetry library works, but can't decode the binary propagation format being used in DYNAMOS. So, DYNAMOS was adapted to also propagate text-based trace information, which the Python services can use to manually construct new traces.

4.8.2 Timing issues

In tracing, if an application exits before all spans are forwarded to the trace collector, some tracing information might be lost. The microservice chains are designed to exit after completing message processing. And are susceptible to exiting before properly emitting traces. Thus, the microservice chain libraries are/need to be designed to exit gracefully.

```
func main(){
    // Initialize a tracer instance
    _, err := lib.InitTracer(serviceName)
    if err != nil {
        logger.Sugar().Fatalf("Failed to create ocagent-exporter: %v", err)
    }
    ..
}
// Start a span for this function
func requestApprovalHandler(c pb.SideCarClient) http.HandlerFunc {
    ctx, span := trace.StartSpan(ctx, "requestApprovalHandler")
    // Defer will close the span on function exit
    defer span.End()

    // pass the context to the next function to create a 'child span' of the
        current span
    otherfunction(ctx)
    // Do work
    return
}

func otherfunction(ctx) {
    ctx, span := trace.StartSpan(ctx, "otherfunction")
    defer span.End()
    ..
    return
}
```

Listing 4.11: DYNAMOS tracing example

4.9 API

Both the orchestrator and the distributed agents expose an API endpoint. Apart from the two main methods needed to get approval and perform a data request, every etcd static configuration resource can be dynamically updated. The body for each request can be found in the protocol buffer files, or in the etcd configuration files.

Method	**Target**	**Path**	**Info**
POST	orchestrator	$/api/v1/requestapproval$	Get data request approval
POST	agent	$/agent/v1/sqlDataRequest/ < agentname >$	Send data request
GET	orchestrator	$/api/v1/archetypes$	Get all archetypes
GET	orchestrator	$/api/v1/archetypes/ < archetype\ name >$	Get specific archetype
PUT	orchestrator	$/api/v1/archetypes$	Add new/update archetypes
GET	orchestrator	$/api/v1/policyEnforcer$	Get all agreements
GET	orchestrator	$/api/v1/policyEnforcer/ < data\ steward\ name >$	Get specific agreement
PUT	orchestrator	$/api/v1/policyEnforcer$	Add new/update agreements
GET	orchestrator	$/api/v1/requestTypes$	Get all request types
GET	orchestrator	$/api/v1/requestTypes/ < requestType\ name >$	Get specific request type
PUT	orchestrator	$/api/v1/requestTypes$	Add new/update request types
GET	orchestrator	$/api/v1/microservices$	Get all microservices
GET	orchestrator	$/api/v1/microservices/ < microservice\ name >$	Get specific microservice
PUT	orchestrator	$/api/v1/microservices$	Add new/update microservices

Chapter 5

Experiment design

In this section, we evaluate the performance and bottlenecks of DYNAMOS. We want to get a feel for data transfer speeds between microservices. Then we look at the bottleneck of dynamically creating jobs on user request arrival.

5.1 Experiment setup

5.1.1 Hardware

All experiments are run on a personal MacBook Pro, CPU: 2,6 GHz 6-Core Intel Core i7, Memory: 16 GB 2667 MHz DDR4. Kubernetes runs on 'Docker Desktop', and is given 6 CPU cores, 10GB of memory, and 2GB of swap memory.

5.1.2 Subject

For the experiment, we use two microservices, namely a 'query' service and an 'algorithm' service. The query service is written in Python, it will do a database query on 2 CSV files, transform the data to a protobuf readable format, and forward it to the next service. The two CSV files contain synthetic salary information about university personnel. However, to have very specific control over the size of the data queried a simple join will be done without an 'ON' clause. This will result in the Cartesian product of the two tables, with the 'LIMIT' clause to cap the data to a certain size.

```
SELECT * FROM Personen p JOIN Aanstellingen s LIMIT 10000
```

The algorithm service, written in Go, will transform the data back to a list of comma-separated rows, basically the text output of the SQL query. Only the first two rows are returned, however. This is to shorten the HTTP transfer time and simulate a more realistic outcome of a data request, where only a specific result would be returned. Both the computeToData, and dataThroughtTtp archetypes will be tested.

5.1.3 Trace measuring points

Figures 5.1 and 5.2, show full traces of a SQL query of 30000 rows, through both the computeToData and dataThroughTtp archetypes. In the traces, we identified three measuring points (MP) for the computeToData archetype, and six MP for the dataThroughTtp archetype. Each MP constitutes a type of data transfer, starting from the query service. The MP is either:

- A transfer from a data-sharing microservice to another microservice
- From a data-sharing microservice to a sidecar
- From the sidecar to the agent or a data-sharing microservice
- Agent to agent

Tables 5.1 and 5.2 show where the transfer happens, what communication methods are used, and which points in the trace are used for that MP.

Measuring transfer speeds in these traces is not completely straightforward, however. As seen in the *'query SendData'* entry in figure 5.1, the trace is as long as the entire handling of the next microservice.

This is because the 'SendData' function is executed on the container of the next service but then handles the entire message flow before returning an answer to the gRPC method. In other words, the moment the 'SendData' function starts execution is the moment all data has been transferred to the next service and is the point that needs to be measured. See listing 5.1 on how this approximately works in pseudocode.

Archetype			**Calculation**	
Compute to Data			StartOf(endpoint) - StartOf(starting point)	
Ref	**Transfer**	**Method**	**Starting point**	**Endpoint**
MP1	Query → Algorithm	gRPC	query /proto.Microservice/SendData	algorithmService sidecar SendData/func:
MP2	Algorithm → Sidecar	gRPC	sidecar /proto.Microservice/SendData	sidecar SendData/func:
MP3	Sidecar → UVA	gRPC + RabbitMQ	sidecar SendDataThroughAMQ/func	UVA/func: handleIncomingMessages

Table 5.1: Compute to data measuring points

Archetype			**Calculation**	
Data through ttp			StartOf(endpoint) - StartOf(starting point)	
Ref	**Transfer**	**Method**	**Starting point**	**Endpoint**
MP4	Query → Sidecar	gRPC	query /proto.Microservice/SendData	sidecar SendData/func:
MP5	Sidecar → UVA	gRPC + RabbitMQ	sidecar SendData/func:	UVA/func: handleIncomingMessages
MP6	UVA → SURF	RabbitMQ	(UVA) proto.Sidecar. SendMicroservice Communication[1]	SURF/func: handleIncomingMessages
MP7	SURF → Sidecar	gRPC	(SURF) proto.Sidecar. SendMicroservice Communication[2]	
MP8	Algorithm → Sidecar	gRPC	sidecar /proto.Microservice/SendData	sidecar SendData/func:
MP9	Sidecar → SURF	gRPC + RabbitMQ	sidecar SendDataThroughAMQ/func	SURF/func: handleIncomingMessages

Table 5.2: Data through ttp measuring points

```
func (s *SharedServer) SendData(ctx context.Context, data *pb.MicroserviceCommunication)
    (*emptypb.Empty, error) {
  // Enter function, now all the data is available in the next microservice, at the *pb.
     MicroserviceCommunication pointer.
       // So start a manual trace
       startManualTrace(ctx,data)

       // Execute whatever this container needs to do:
       handleDataRequest(ctx, data)

       // Return after executing, thus closing the sendData span
  return &emptypb.Empty{}, nil
}
```

Listing 5.1: Usage SendData explained

[1] Measure times between the end of the first and the start of the second function
[2] length of this function can be taken as is

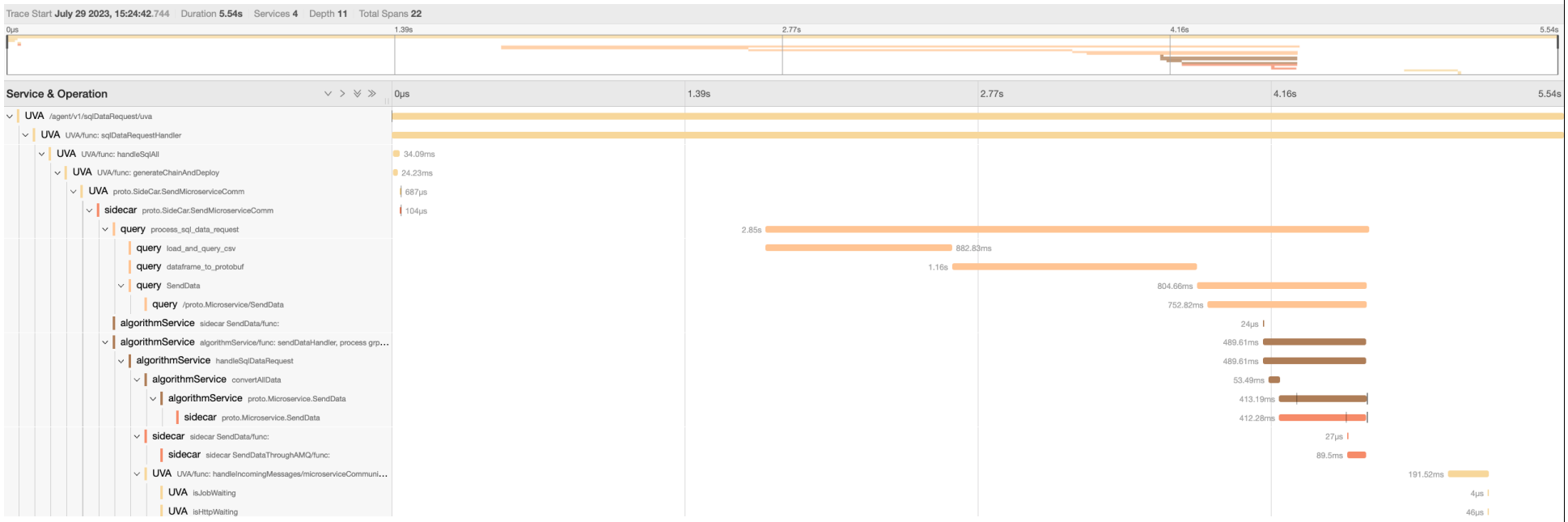

Figure 5.1: Trace of a 30000 line SQL request, compute to data archetype

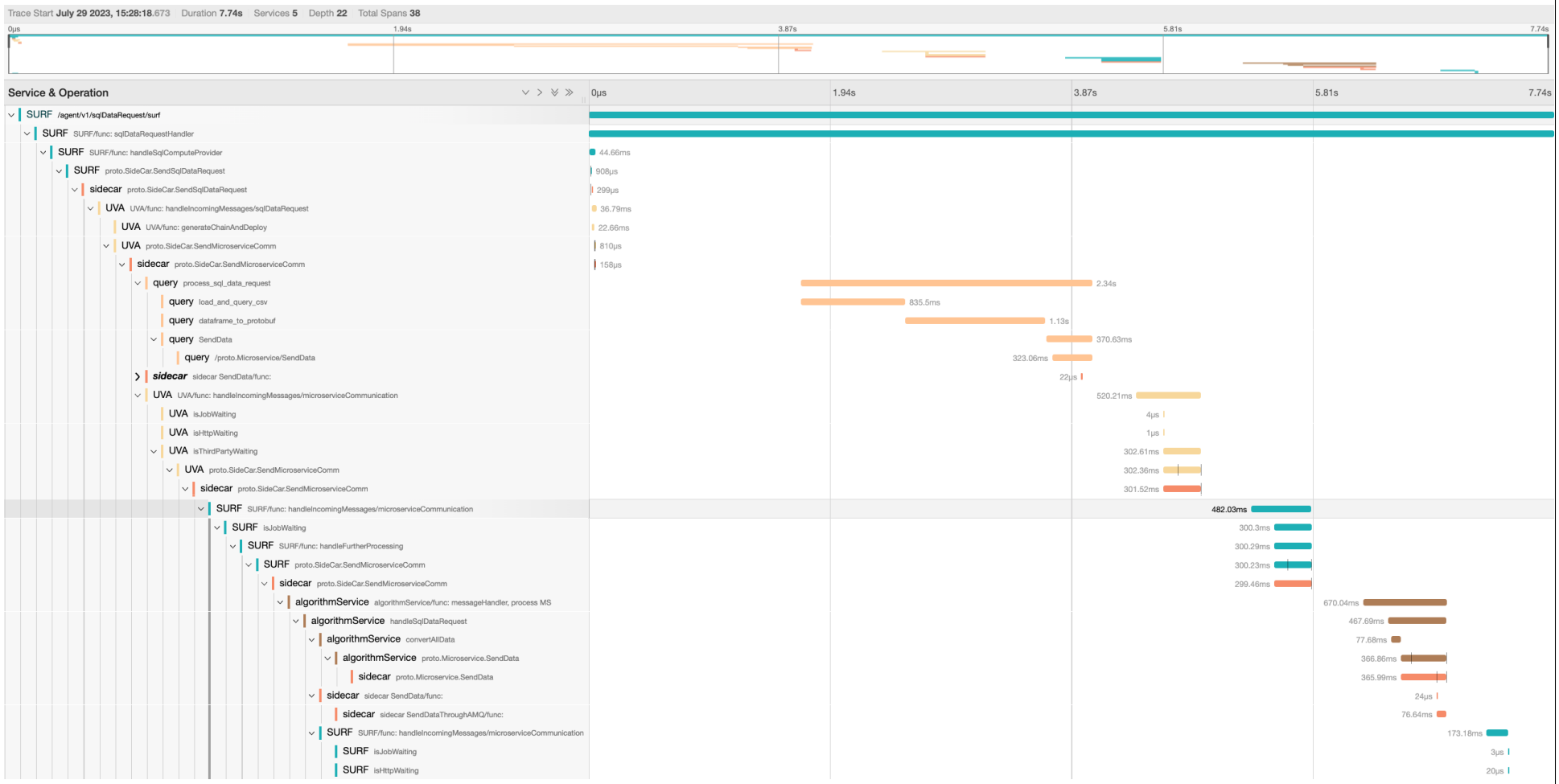

Figure 5.2: Trace of a 30000 line SQL request, data through ttp archetype

5.2 Experiment description

5.2.1 Experiment 1

The goal of this experiment is to get numbers on data transfer speeds and the size of data in different formats. The distributed traces of each 'sqlDataRequest' have been enhanced to record the following attributes of the complete message after processing:

- Size of the message in bytes as currently formatted (gRPC proto-wire format)
- Size of the message in bytes serialized to JSON

The queries will be done in three sizes 10000, 20000, and 30000 respectively. Five queries will be done for each size, with some spacing in between to make sure the system is always at rest. Afterward, the traces can be analyzed to get average transfer times and message sizes.

5.2.2 Experiment 2

For this experiment, a new set of traces will be created and analyzed to get more insights into the major bottlenecks of the system. The following measures are taken for experiment two.

- Startup time → Start of the 'query' service - the start time of 'SendMicroserviceCommunicatoin'
- Total processing time → all processing functions in the query and algorithm services, excluding transfer times
- Transfer time → $TransferTime = Latency - (StartupTime + TotalProcessingTime)$

Furthermore, for a comparison Figure 5.3, shows the trace, another proof-of-concept implementation, made at the University of Amsterdam by Marten Steketee [36]. The architecture of of this application is very different, it is a monolithic application, so changes to services need to be programmed in, and it allows a single database scheme at the moment. Furthermore, the processing is also more optimal due to the use of an effective SQL database.

Due to the monolithic nature, and lack of a sidecar, there are fewer data transfers happening. The only data transfer in the trace is directly from one data provider to the trusted third party. This transfer is done through RabbitMQ, in the message the data is encoded as a string, which makes the size larger than DYNAMOS transfers.

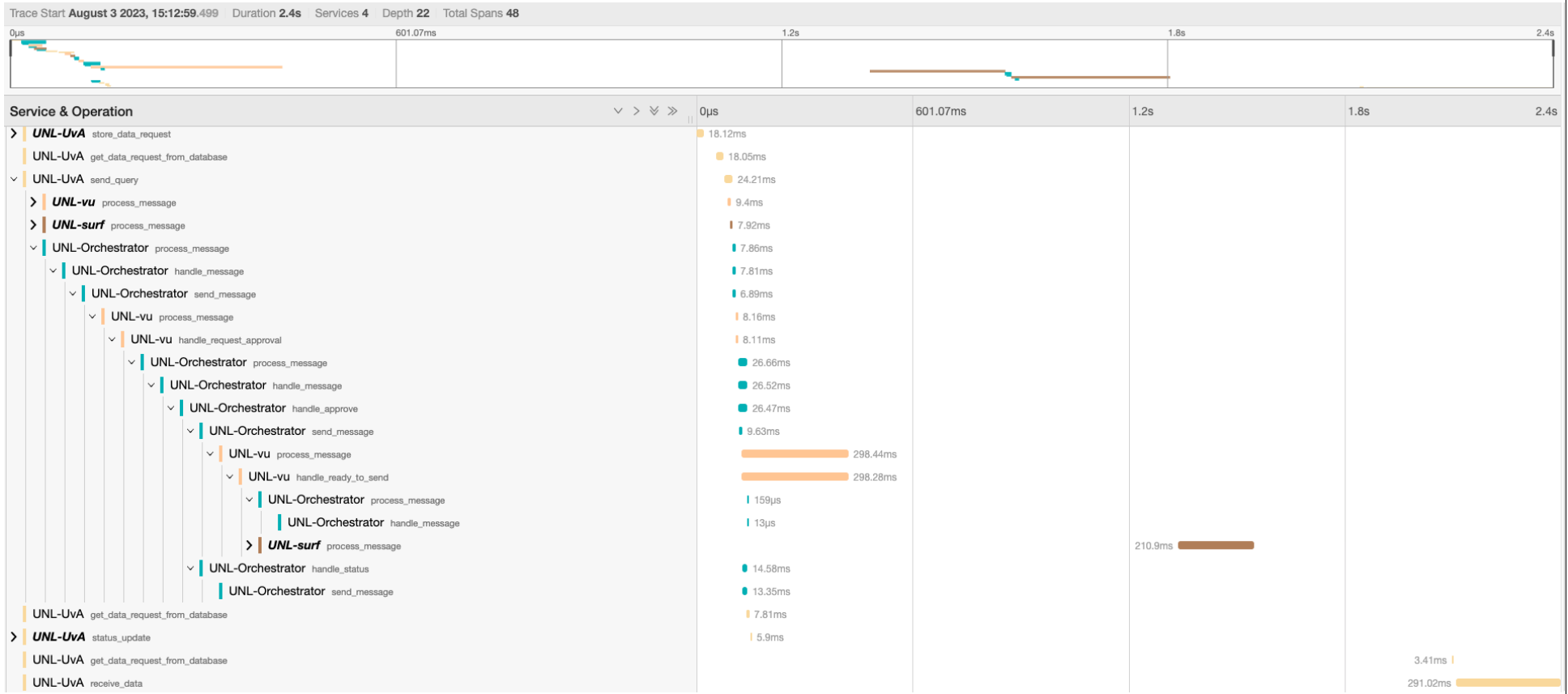

Figure 5.3: Trace of 4141 line SQL request, compute to data archetype, different program

5.3 Experiment results and discussion

5.3.1 Experiment 1

Tables 5.3, 5.5, 5.4, 5.6 show the results and statistics from the first experiment.

5.3.1.1 Message sizes

Limit	**proto-wire format** (in bytes)	**JSON encoded** (in bytes)	**On disk** (in bytes)	**Avg. gRPC transfer speed**	**Transfer speed MB/s**
10000	1.344.899	1.180.220	983.455	102ms	13,25
20000	2.693.365	2.364.465	2.463.677	200ms	13,43
30000	4.036.759	3.542.934	2.955.314	296ms	13,63

Table 5.3: Message sizes in bytes

As discussed in chapter 4.6, the message size of a proto-wire encoded message is larger than its JSON-encoded equivalent.

5.3.1.2 Data transfer speeds

For the largest (4MB) message the gRPC data transfer averages out to around 296 ms, which means an average transfer speed of 13.6 MB/s.

Limit	**Latency**	**MP1** gRPC	**MP2** gRPC	**MP3** gRPC + RabbitMQ
10000	3.31s	115	99	157
10000	3.43s	96	110	160
10000	3.31s	112	98	144
10000	3.51s	97	108	199
10000	4.12s	113	119	175
20000	4.3s	186	205	347
20000	4.74s	167	256	326
20000	5.08s	340	342	339
20000	4.85s	181	239	349
20000	4.69s	191	281	339
30000	6.47s	318	676	707
30000	6.01s	253	658	500
30000	6.01s	250	498	586
30000	5.59s	269	669	621
30000	5.89s	232	315	476

Table 5.4: Data transfer times computeToData archetype, measurements in milliseconds,

Limit	**Latency**	**MP4** gRPC	**MP5** gRPC + RabbitMQ	**MP6** RabbitMQ	**MP7** gRPC	**MP8** gRPC	**MP9** gRPC + RabbitMQ
10000	4.49s	84	150	98	98	124	130
10000	4.41s	84	163	128	96	97	137
10000	4.9s	100	175	139	95	88	117
10000	4.58s	87	152	131	111	108	124
10000	4.5s	92	150	118	97	101	126
20000	5.45s	191	323	296	273	205	264
20000	6.77s	227	300	284	211	181	259
20000	6.09s	155	291	258	211	173	239
20000	5.95s	157	281	226	190	190	226
20000	6.01s	165	312	251	202	185	232
30000	7.82s	247	424	337	320	312	396
30000	8.08s	370	460	307	278	266	357
30000	8.06s	401	475	306	287	263	351
30000	8.22s	253	468	318	309	431	391
30000	8.01s	240	486	288	283	278	364

Table 5.5: Data transfer times for dataThroughTtp archetype, measurement in milliseconds,

5.3.2 Experiment 2

Table 5.8 and 5.9 show latency, startup time, total processing time, and transfer times all derived from the distributed traces. Table 5.7 shows some basic statistics on those data points.

From these results, two main bottlenecks can be observed. Namely, ephemeral job startup time and data transfer time. For the ephemeral job startup time, we discuss alternative architectures and decisions that can be taken to minimize this. Processing time could most likely be improved a bit as well, but that is ignored for now since that is mainly an engineering issue.

For the data transfer time, we discuss removing the sidecar pattern and the proof-of-concept (POC) implementation of Marten Steketee [36] mentioned in section 5.2.

5.3.2.1 Job startup time

A 1.7-second average startup loss is long in the modern web application world. However, in this thesis, we cannot qualify whether this would be unacceptable for a digital data-sharing marketplace solution.

If future user requirements would see this as unacceptable, there are ways to mitigate this. The persistent job architecture could be used to avoid cold starts. The trade-off is unnecessary resource consumption and possible security risks (see section 3.5). Otherwise, for ephemeral jobs, a solution could be to not allow the user to add extra options dynamically (like the 'graph' flag). This would allow the agents to always have a job ready to reduce startup times, at the cost of user experience.

5.3.2.2 Data transfer time

With the largest query (30000 lines, 4MB message), around 50% of the time is spent in data transfers in dataThroughTtp versus 23% in the computeToData archetype. Correcting for the startup time of the container, this percentage rises to 62% and 32% respectively. The obvious difference is in the three, respectively six data transfers happening.

In the alternative POC, the message consists of 4141 lines and is around 8 MB. Two transfers happen, the first from the data provider to the third party by RabbitMQ, which costs 916ms. The second from the third party to the user, 540ms, for a total of 1.46 seconds. The total latency is 2.4 seconds, making

Measuring points	**Avg**	**Min**	**Max**	**Stdev**
MP1, measuring gRPC transfers				
10000	107	96	115	9
20000	213	167	340	72
30000	248	219	267	20
MP2, MP4, MP8, measuring gRPC transfers				
10000	101	84	124	13
20000	191	155	250	25
30000	312	240	464	70
MP3, MP5, MP9, measuring gRPC + RabbitMQ				
10000	151	117	199	22
20000	295	226	349	43
30000	471	351	707	102
MP6, measuring RabbitMQ message transfer				
10000	123	98	139	16
20000	263	226	296	28
30000	311	288	337	18
MP7, measuring gRPC transfers				
10000	99	99	99	99
20000	217	190	273	32
30000	295	278	320	18

Table 5.6: Statistics data transfer times, in milliseconds

data transfers to be approximately 60% of the processing time.

This shows two facts. First, encoding and transferring data is done more efficiently in DYNAMOS, which could be easily improved in the other POC. Second, the number of transfers done in DYNAMOS highly impacts its latency.

Due to the microservice architecture, there will always be more data transfers than in a monolithic application. In DYNAMOS, the sidecar pattern, adds to that number of transfers. Completely removing the sidecar will reduce the dataThroughTtp transfers from 6 to 4, and the computeToData archetype from 3 to 2.

Removing the sidecar pattern by compiling a messaging (RabbitMQ) library into each data-sharing microservices has its trade-offs. There would be tighter coupling with the messaging platform, increasing difficulty to switch to other solutions if so required. Furthermore, microservices written in different programming languages will use different libraries connecting to the same RabbitMQ instance, this could lead to a maintenance overhead.

	Average	**MIN**	**MAX**	**STD**
StartupTime:	1,70	1,53	2,42	0,21
Processing Time:				
10000	1,21	1,08	1,40	0,11
20000	1,90	1,61	2,43	0,26
30000	2,42	2,18	2,85	0,20
ComputToData TransferTime:				
10000	0,53	0,47	1,33	0,31
20000	1,00	0,88	1,14	0,11
30000	1,24	1,17	1,33	0,06
DataThroughTtp TransfertIme:				
10000	1,45	1,28	1,58	0,14
20000	2,66	2,46	2,85	0,17
30000	3,92	3,65	4,18	0,21

Table 5.7: Bottleneck analysis statistics

Limit	**Latency**	**Startup Time**	**Total Processing Time**	**Transfer Times**	**Startup %**	**Processing Time %**	**Transfer Time %**	**Transfer Time %** Excl. Startup
10000	3,22	1,55	1,15	0,52	48,23%	35,71%	16,06%	31,01%
10000	3,41	1,55	1,36	0,50	45,51%	39,88%	14,60%	26,80%
10000	3,17	1,57	1,14	0,47	49,50%	35,84%	14,67%	29,04%
10000	4,14	2,11	1,40	0,64	50,97%	33,70%	15,34%	31,28%
10000	3,73	1,97	1,22	0,54	52,73%	32,71%	14,56%	30,80%
20000	4,31	1,58	1,83	0,90	36,68%	42,41%	20,90%	33,02%
20000	5,63	2,16	2,43	1,05	38,37%	43,07%	18,56%	30,12%
20000	4,47	1,61	1,84	1,02	36,04%	41,19%	22,77%	35,61%
20000	4,42	1,60	1,95	0,88	36,11%	44,05%	19,84%	31,06%
20000	5,11	1,69	2,28	1,14	33,15%	44,56%	22,29%	33,34%
30000	5,24	1,54	2,37	1,33	29,41%	45,17%	25,42%	36,01%
30000	5,76	1,65	2,85	1,27	28,56%	49,45%	21,99%	30,78%
30000	5,34	1,70	2,42	1,22	31,84%	45,37%	22,79%	33,43%
30000	5,2	1,62	2,41	1,17	31,23%	46,31%	22,46%	32,66%
30000	5,41	1,57	2,62	1,22	28,96%	48,43%	22,61%	31,82%

Table 5.8: Bottleneck analysis, computeToData archetype

Limit	**Latency**	**Startup Time**	**Total Processing Time**	**Transfer Times**	**Startup %**	**Processing Time %**	**Transfer Time %**	**Transfer Time %** Excl. Startup
10000	4,13	1,73	1,076	1,33	41,84%	26,05%	32,11%	55,20%
10000	4,41	1,63	1,211	1,57	37,01%	27,46%	35,53%	56,41%
10000	4,49	1,61	1,299	1,58	35,88%	28,93%	35,19%	54,88%
10000	4,3	1,69	1,137	1,47	39,33%	26,44%	34,23%	56,42%
10000	3,99	1,62	1,084	1,28	40,65%	27,17%	32,18%	54,22%
20000	6,16	1,669	1,828	2,66	27,09%	29,68%	43,23%	59,30%
20000	6,41	1,785	1,771	2,85	27,85%	27,63%	44,52%	61,71%
20000	6,11	1,729	1,845	2,54	28,30%	30,20%	41,51%	57,89%
20000	5,65	1,589	1,606	2,46	28,12%	28,42%	43,45%	60,45%
20000	6,84	2,417	1,607	2,82	35,34%	23,49%	41,17%	63,67%
30000	7,66	1,616	2,271	3,77	21,10%	29,65%	49,26%	62,43%
30000	7,59	1,563	2,377	3,65	20,59%	31,32%	48,09%	60,56%
30000	7,7	1,525	2,183	3,99	19,81%	28,35%	51,84%	64,65%
30000	8,31	1,604	2,527	4,18	19,30%	30,41%	50,29%	62,32%
30000	7,99	1,758	2,218	4,01	22,00%	27,76%	50,24%	64,41%

Table 5.9: Bottleneck analysis, dataThroughTtp archetype

Chapter 6

Discussion

Link back to OS attributes?

In Chapter 5 we saw the results and discussion of the specific experiments. In this chapter, we look more broadly at DYNAMOS as a whole. To do this we start by discussing each research question in turn.

6.1 Research questions

6.1.1 RQ 1

What are the advantages and disadvantages of different approaches to automatically re-compose a running microservices architecture in a distributed system?

Throughout the thesis, we looked at many different aspects of designing DYNAMOS. Each chapter already listed the advantages and disadvantages of the part of DYNAMOS it describes, so we don't repeat ourselves by re-iterating every decision. A few findings however stand-out that we will discuss here.

> **Finding 2:** Using a microservice architecture in data-exchange scenarios increases latency due to the increased amount of data transfers from microservice to microservice, the utilized sidecar pattern magnifies this issue

As described in subsection 5.3.2, six data transfers are identified with only two microservices in the dataThroughTtp archetype. With the larger 4 MB data transfer latency went up to around 8 seconds (including container startup time). We can't validate whether this latency is too much for this scenario.

The advantages of being able to dynamically add data-exchange services and having a loosely coupled communication solution are valid reasons to accept this extra latency.

Re-composing the microservices has been effectively solved by implementing the microservice chains as directed acyclic graphs. The way chains are generated, and how etcd functions as a central data store and configuration tool, allows DYNAMOS to implement algorithms to dynamically choose architectures or archetypes per request.

Furthermore, a generic way has been created to pass data between the services and distributed agents of the system, using gRPC and the sidecar pattern. The data exchange microservices have to include a small DYNAMOS gRPC library to interface with this sidecar or other data-exchange microservices.

6.1.1.1 Disadvantages

As we saw in Chapter 5, there are a few bottlenecks in DYNAMOS. Namely the startup times of containers in ephemeral jobs, and extra data transfer time. Due to having docker microservices, which always require extra transfers compared to a monolith. This effect is further magnified by the sidecar pattern.

The DYNAMOS gRPC library for interfacing with the sidecar requires extra care for developers that create date-exchange microservices. For a DYNAMOS developer, the gRPC setup has a learning curve, it is complex.

6.1.1.2 Advantages

The sidecar pattern makes all communication loosely coupled from all DYNAMOS components and the data-exchange microservices. Furthermore, it is a single communication pane from services written in different programming languages.

The microservice chains promote a distributed programming environment where data-exchange services can be developed by people not related to DYNAMOS. This is made possible by abstracting all communication and microservice architectures away from the user and making the latter a choice for the platform.

6.1.2 RQ 2

What attributes/properties should a 'knowledge-base' contain to compose microservices in a DDM environment?

One part of this question could be stated in another way, what information is needed to generate microservice chains? The answer to that question can be taken from Chapter 4.3.

We solved this by mapping optional and required microservices to a 'requestType', giving these microservices a specific order and a label, and abstracting an archetype to only look at who the 'computeProvider' is so the data stewards can be assigned a role. In DYNAMOS this is set as JSON in the configuration management store.

Apart from the properties required to generate the microservice chain, a few more practical properties are required. Namely, queue names of microservices (jobs), and endpoints of the distributed agents. In DYNAMOS, this information is handled by the distributed agents and, again, stored in the configuration management store.

6.1.3 RQ 3

What extra attributes/properties should a 'knowledge-base' contain to make decisions on extra-functional properties, like energy consumption and performance?

As discussed in Chapter 3.6, we defined extra-functional properties as user experience and energy consumption. We also showed how past a certain point of system load (80%), it can make sense to change archetypes and distribute the load more evenly over the system.

In Chapter 4.3, we show how DYNAMOS can dynamically adapt archetypes based on a 'weight' property. In Chapter 4.5 we demonstrate an architecture on how DYNAMOS can make adaptive choices based on CPU and memory load, although we don't actually implement the algorithms.

The answer to this research question ends up not being an exact list of properties. But the assurance, and showcase, of how DYNAMOS can actively adapt both active, and new jobs. The exact algorithms are for a future iteration of DYNAMOS.

6.2 Threads to validity

6.2.1 Specific sample use case

In this thesis we took two archetypes and the UNL use case as basis for the DYNAMOS implementation. I believe the implementation of the language to generate microservice chains, including the way archetypes are defined, are 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.

6.2.2 Experiment statistics

The measurements and statistics shown in Chapter 5 are by no means statistically relevant. They were taken on a single laptop, not a distributed environment, and lacking automation of trace analysis the sample size was deliberately small.

The purpose of the experiments though was to get an initial feel for data transfer speed, bottlenecks, and the effectiveness of the system. This objective was attained, showing quite clearly the advantages and disadvantages of the different microservice architectures.

6.2.3 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. The requirements for the microservices are generally listed in C, so for this use case we believe it will work. However, we can not show, or prove, that this method of passing data is user friendly, or generic enough to scale up towards more complex data-sharing scenarios.

6.2.4 Authorization and data pods

The generic architecture hinges on the idea that we can implement webtokens, that can be invalidated, to access data pods on data stewards. This has not been tested, or researched much for DYNAMOS, although we believe that the techniques and technology does exist to make this happen, as discussed in Chapter 2.

6.2.5 Message size

As noted in 4.3, 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 thesis.

6.2.6 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.

Chapter 7

Related work

7.1 Digital Data-sharing Marketplaces

The paper by Shakeri *et al.* [1] is a major inspiration for the design of this thesis. The paper proposes a high-level architecture for a container-based DDM, clearly defining how a DDM would work, and archetypes relevant for DDMs as seen in figure 2.2 (chapter 2. This architecture in figure 7.1 defines many roles also used in this thesis like the 'Policy matcher', 'Request Handler', and request-based container setup. The focus of Shakeri *et al.* however is evaluating and comparing different container overlay networking methods, which is not what we are trying to achieve in this thesis.

The definitions of data-sharing archetypes were defined before Shakeri *et al.* by Zhang *et al.* [3]. The authors model type of customer requests to these 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?'. My hope is that the result and platform of this thesis can be an experimental ground to answer questions such as that.

7.2 Self-adaptive Microservice systems

A lot of the research on self-adaptive microservices tends to be focused on architecture and modeling [4][43][44][45]. Mendonça *et al.* acknowledge the limited scope of current works addressing the challenges of developing self-adaptive microservice applications [23][37][46]. They strive to bridge the gap between state-of-the-art research and practice, with a particular emphasis on 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 [23, 37] based on the self-adaptability patterns of Rainbow [47].

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 [43]. Many papers on self-adaptive (microservice) systems predominantly focus on Quality of Service (QoS) 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 [44]. Additionally, a proposal for a multilevel self-adaptation architecture and domain-specific language (DSL) addresses seamless service upgrades [45].

The MAPE-K approach is widely used in self-adaptive systems, and Arcaini *et al.* serve as a valuable inspiration. Their work formalizes the modeling of MAPE-K systems in distributed systems, discussing aspects of validation and correctness [4]. Boyapati *et al.* [48] implement a lightweight MAPE-K loop to control dynamic rate limiting in a Docker compose setup.

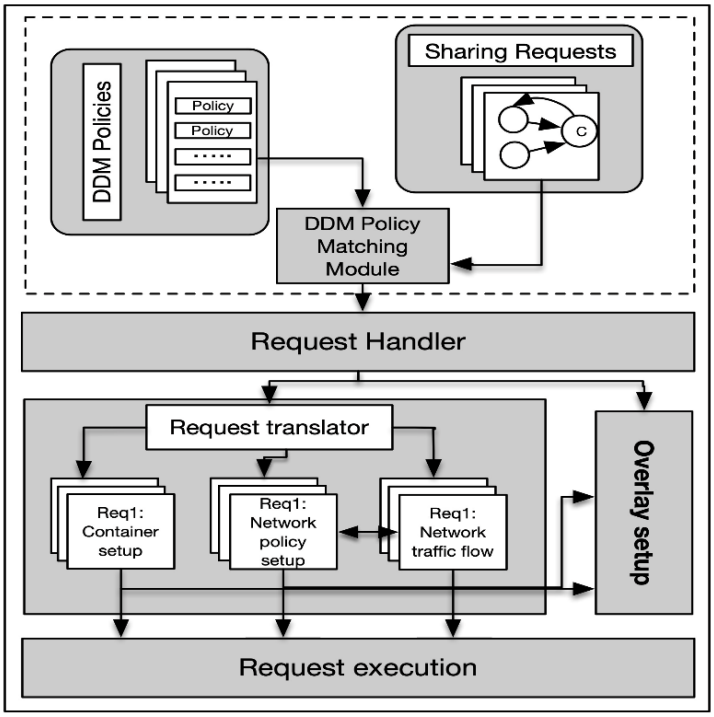

Figure 7.1: Container-based DDM [1]

Chapter 8

Conclusion

Creating a distributed system with a microservice architecture is not a trivial task, the 8000+ lines of code that constitute DYNAMOS might be a testament to that fact. However, we feel confident that DYNAMOS is a useful simulator for experimenting with microservice patterns and microservice architectures in a DDM environment. We defined microservice chains as a directed acyclic graph, designed two microservice architectures, and incorporated two data-sharing archetypes. The power of DYNAMOS is that it can handle both archetypes and architectures and can be extended to support more. Finally, we performed initial experiments to gain insight into bottlenecks and the advantages and disadvantages of different microservice architectures.

8.1 Future work

There is a whole lot of future research we can think of to continue the work on DYNAMOS. A few spring to mind however as more essential to show that the general architecture of DYNAMOS is relevant.
Data pods and authorization, in this thesis, we have passingly mentioned the use of data pods and web tokens as authentication and authorization methods. 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 DDM, whether a microservice or monolithical architecture is used.
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 a federated learning model, where results of learning algorithms on different parties aggregate. And archetypes that are more flexible in who gets the results.
Archetype selection algorithms, DYNAMOS 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.

Acknowledgements

If so inclined, thank people.

Bibliography

[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] A. I. Exchange, "Amdex fieldlab funding request," 2020.

[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] 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.

[5] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting empirical methods for software engineering research," *Guide to advanced empirical software engineering*, pp. 285–311, 2008.

[6] A. Silberschatz, P. Galvin, and G. Gagne, *Operating System Concepts, 10/E.* 2018.

[7] A. S. Tanenbaum and R. Van Renesse, "Distributed operating systems," *ACM Computing Surveys (CSUR)*, vol. 17, no. 4, pp. 419–470, 1985.

[8] P. K. Sinha, *Distributed operating systems: concepts and design*. PHI Learning Pvt. Ltd., 1998.

[9] 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.

[10] 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.

[11] J. Lewis and M. Fowler. "Microservices: A definition of this new architectural term." (2014).

[12] Y. Wang, H. Kadiyala, and J. S. Rubin, "Promises and challenges of microservices: An exploratory study," *Empirical Software Engineering*, vol. 26, 2021.

[13] S. Cadieux and M. Heyn. "The journey to an agile organization at zalando." (2018), [Online]. Available: https://www.mckinsey.com/capabilities/people-and-organizational-performance/our-insights/the-journey-to-an-agile-organization-at-zalando.

[14] S. Hassan and R. Bahsoon, "Microservices and their design trade-offs: A self-adaptive roadmap," in *2016 IEEE International Conference on Services Computing (SCC)*, 2016, pp. 813–818. DOI: 10.1109/SCC.2016.113.

[15] M. Waseem, P. Liang, M. Shahin, A. D. Salle, and G. M'arquez, "Design, monitoring, and testing of microservices systems: The practitioners' perspective," *J. Syst. Softw.*, vol. 182, p. 111 061, 2021.

[16] humanitec. "Kubernetes benchmarking study." (2022), [Online]. Available: https://humanitec.com/whitepapers/kubernetes-benchmarking-study-2022.

[17] C. Rosen. "Docker swarm vs. kubernetes: A comparison." (2022), [Online]. Available: https://www.ibm.com/cloud/blog/docker-swarm-vs-kubernetes-a-comparison.

[18] K. Casey. "Kubernetes by the numbers, in 2022: 11 stats to see." (2022), [Online]. Available: https://enterprisersproject.com/article/2022/10/kubernetes-statistics-2022.

[19] M. Matherson. "26 kubernetes statistics to reference." (2021), [Online]. Available: https://www.airplane.dev/blog/kubernetes-statistics.

[20] Kubernetes. "Custom resources." (), [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/.

[21] Kubernetes. "Operator pattern." (), [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/.

[22] Kubernetes. "Kubernetes jobs." (), [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/job/.

[23] 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.

[24] J. P. S. Cardoso, "A guide for microservices in greenfield projects," Ph.D. dissertation, Instituto Politecnico do Porto (Portugal), 2021.

[25] J. Kanjilal. "Logging microservices: The challenges and solutions." (), [Online]. Available: https://www.developer.com/design/logging-microservices/.

[26] A. Leong. "A guide to distributed tracing with linkerd." (), [Online]. Available: https://linkerd.io/2019/10/07/a-guide-to-distributed-tracing-with-linkerd/.

[27] RedHat. "What's a service mesh?" (), [Online]. Available: https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh.

[28] Q. Coltof, "Exploratory case study: Com- paring design patterns for green- field microservices architecture development on the thesis fair platform," Ph.D. dissertation, University of Amsterdam, 2022.

[29] gRPC Authors. "A high performance, open source universal rpc framework." (), [Online]. Available: https://grpc.io/.

[30] R. Inanc. "Benchmarking rest vs. grpc." (), [Online]. Available: https://medium.com/sahibinden-technology/benchmarking-rest-vs-grpc-5d4b34360911.

[31] L. Kami'nski, M. Kozlowski, D. Sporysz, K. Wolska, P. Zaniewski, and R. Roszczyk, "Comparative review of selected internet communication protocols," *Foundations of Computing and Decision Sciences*, vol. 48, pp. 39–56, 2022.

[32] gRPC Authors. "Supported languages." (), [Online]. Available: https://grpc.io/docs/languages/.

[33] E. architecture Centre of Excellence. "Business process management (bpm)center of excellence (coe) glossary." (2009), [Online]. Available: https://web.archive.org/web/20170131011831/https://www.ftb.ca.gov/aboutFTB/Projects/ITSP/BPM_Glossary.pdf.

[34] A. Proj. "Argo workflows." (), [Online]. Available: https://argoproj.github.io/argo-workflows/.

[35] "Solid project." (), [Online]. Available: https://solidproject.org/.

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

[37] 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.

[38] "The go programming language." (), [Online]. Available: https://go.dev/.

[39] A. Proj. "Argo artifacts." (), [Online]. Available: https://argoproj.github.io/argo-workflows/walk-through/artifacts/.

[40] J. v. Kistowski, H. Block, J. Beckett, K.-D. Lange, J. A. Arnold, and S. Kounev, "Analysis of the influences on server power consumption and energy efficiency for cpu-intensive workloads," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '15, Austin, Texas, USA: Association for Computing Machinery, 2015, pp. 223–234, ISBN: 9781450332484. DOI: 10.1145/2668930.2688057. [Online]. Available: https://doi.org/10.1145/2668930.2688057.

[41] "Topological sorting." (), [Online]. Available: https://en.wikipedia.org/wiki/Topological_sorting.

[42] Linkerd. "Distributed tracing with linkerd." (), [Online]. Available: https://linkerd.io/2.13/tasks/distributed-tracing/.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

Acronyms

AMDEX Amsterdam Data Exchange Fieldlab. 5, 6
DAG directed acyclic graph. 21, 33
DDM Digital Data-sharing Marketplaces. 5, 6, 10, 23, 61, 63
QoS Quality of Service. 61

Appendix A

Installation instructions

A.1 Installing DYNAMOS

Before deploying the DYNAMOS components Linkerd needs to be installed and configuration needs to be prepared for rabbitMQ. Listing A.1 and A.2 show exactly how to prepare the Kubernetes cluster and deploy the components.

A.1.0.1 Linkerd

```
#Linkerd
linkerd install --crds | kubectl apply -f -
linkerd install --set proxyInit.runAsRoot=true | kubectl apply -f -
linkerd check


linkerd jaeger install | kubectl apply -f -
# Possibly: linkerd viz install | kubectl apply -f -
```

Listing A.1: Preparing cluster

A.1.0.2 RabbitMQ

```
# Create namespaces by deploying the core.
corePath=<path to chart directory>
coreValues=$corePath/values.yaml
helm upgrade -i -f "${coreValues}" core ${corePath}

# Uninstall core (it will fail anyway)
helm uninstall core

# Create a password for a rabbit user
pw=$(openssl rand -base64 12)

# Add password to all namespaces
kubectl create secret generic rabbit --from-literal=password=${pw} -n orchestrator
kubectl create secret generic rabbit --from-literal=password=${pw} -n uva
kubectl create secret generic rabbit --from-literal=password=${pw} -n vu
kubectl create secret generic rabbit --from-literal=password=${pw} -n surf

# Hash password
docker run --rm  rabbitmq:3-management rabbitmqctl hash_password $pw

# configuration/k8s_service_files/definitions_example.json to definitions.json
# Add hashed password to RabbitMQ definitions.json


#Install prometheus
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo update
helm upgrade -i -f "${coreChart}/prometheus-values.yaml" prometheus prometheus-community
    /prometheus
```

```
#Install Nginx
helm install -f "${coreChart}/ingress-values.yaml" nginx oci://ghcr.io/nginxinc/charts/
    nginx-ingress -n ingress --version 0.18.0

# Install core
helm upgrade -i -f "${coreChart}" core ${corePath}
```

Listing A.2: Preparing rabbitMQ

A.1.0.3 Deploying components

After preparing the cluster the following Helm commands can be used to deploy the individual components at will. Add these commands to a function in a */.zhsrc* file or something equivalent for easy deployment.

```
# Deploy orchestrator layer
orchestratorPath=<path to chart directory>
orchestratorValues=$orchestratorPath/values.yaml
helm upgrade -i -f "${orchestratorValues}" orchestrator ${corePath}

# Deploy agents layer
agentsPath=<path to chart directory>
agentValues=$agentsPath/values.yaml
helm upgrade -i -f "${agentValues}" agents ${agentsPath}

# Deploy thirdparty layer
thirdpartyPath=<path to chart directory>
thirdpartyValues=$thirdpartyPath/values.yaml
helm upgrade -i -f "${thirdpartyValues}" surf ${thirdpartyPath}
```

Listing A.3: Deploying other components

A.1.1 Building DYNAMOS components

Building a service can be done by calling the 'Makefile' from the respective folder, see listing A.4 for an example. The proto-files directory contains all '.proto' files, which define all DYNAMOS messages.

```
cd go
make agent

# make all will build everything
# make proto will just generate the gRPC files.

# Re-deploy by uninstalling the agent
helm uninstall agent
helm upgrade -i -f "${agentValues}" agents ${agentsPath}

# Or just restart the pod:
kubectl rollout restart deployment agent -n agent
```

Listing A.4: Preparing rabbitMQ

A.1.2 Running DYNAMOS

Apart from deploying the application to the cluster, DYNAMOS can also be run locally. Each service has both a 'config_local.go' and 'config_prod.go' file with global variables that allow it to run in, or out of the cluster. Running locally is a bit more basic and might not support every scenario. To run a service with the local variables include the 'tags' flag: *"go run -tags local ."*.

Appendix B

Jobs

B.1 Argo output

```
Name:                artifact-passingxjd42
Namespace:           argo
ServiceAccount:      unset (will run with the default ServiceAccount)
Status:              Succeeded
Conditions:
 PodRunning          False
 Completed           True
Created:             Wed Aug 02 14:38:58 +0200 (20 seconds ago)
Started:             Wed Aug 02 14:38:58 +0200 (20 seconds ago)
Finished:            Wed Aug 02 14:39:18 +0200 (now)
Duration:            20 seconds
Progress:            2/2
ResourcesDuration:   9s*(1 cpu),9s*(100Mi memory)

STEP                          TEMPLATE        PODNAME                                 DURATION  MESSAGE
  artifact-passingxjd42   artifact-example
 |- generate-artifact   whalesay          artifact-passingxjd42-whalesay-4165260985    5s
 |- consume-artifact    print-message     artifact-passingxjd42-print-message-3486089  5s
```

Listing B.1: DYNAMOS at rest

B.2 Defining a job

A Kubernetes job is straightforward to create with the Go SDK, as seen in listing B.2. The job specification features that DYNAMOS uses.

- ActiveDeadlineSeconds → Job will be terminated after this deadline is exceeded, currently set at 10 minutes.
- TTLSecondsAfterFinished → Job will be removed after completing successfully, currently set at 30 seconds.
- BackoffLimit → On failure, how many times will the job retry to create the pod before counting the job as a failure, defaults to 6.

In the current implementation recovering from failures is not handled perfectly yet. If a container has some startup trouble, and recovers, it might all run to completion. However, when a data request is already consumed from the queue it is gone. Meaning that if an application crashes or the pod is evicted for some reason, the request needs to be re-submitted.

There are myriad ways of handling these failure scenarios, which need more research to determine the best-fit scenario for DYNAMOS.

```
1 job := &batchv1.Job{
2     ObjectMeta: metav1.ObjectMeta{
3         Name:      jobName,
4         Namespace: dataStewardName,
```

```
        Labels:    map[string]string{"app": dataStewardName, "jobName": jobName
            }, // for logging purpose
    },
    Spec: batchv1.JobSpec{
        ActiveDeadlineSeconds:   &activeDeadlineSeconds,
        TTLSecondsAfterFinished: &ttl, // Clean up job TTL after it finishes
        BackoffLimit:            &backoffLimit,
        Template: v1.PodTemplateSpec{
            ObjectMeta: metav1.ObjectMeta{
                Labels: map[string]string{"app": dataStewardName},
            },
            Spec: v1.PodSpec{
                Containers:    []v1.Container{},
                RestartPolicy: v1.RestartPolicyOnFailure,
            },
        },
    },
}
```

Listing B.2: Job definition

Appendix C

Requirements

C.1 Microservice requirements

C.1.0.1 General requirements

Requirement	Specification
Receive messages from an input queue	Make use of side-car pattern for RabbitMQ connectivity
Implement proper logging, forwarded to a dashboard	Make use of side-car pattern for logging
Send query result to next Microservice	Original user request should be enriched with next MS destinations
Proper error handling	Breaking errors are thrown to side-car to handle being returned to the user. Computation stops
Implement distributed tracing	Install Istio in the Kubernetes cluster

C.1.0.2 Query service

Requirement	Specification
Retrieve data from a 'data pod'	Support querying MySQL and CSV files To query CSV files, data should be loaded into SQLite This loading should only be done if the data changed (hash comparison) Create a datastore where datasets are matched to query type
Nice to have: Caching of sql results	

C.1.0.3 Anonymize service

Requirement	Specification
Identify columns that need anonymization	Information should be stored as metadata with the dataset
Recognize a type of anonymizing (Generalization, Masking or Perturbation)	Information should be stored as metadata with the dataset
Do anonymizing	Create anonymization functions generalized for different types of data (strings, dates etc.)

C.1.0.4 Algorithm service

Requirement	Specification
Perform calculations on SQL results	Create generic functions that can calculate averages and sum and other basic calculations Recognize what to calculate from message input Generalize for different types of datasets

C.1.0.5 Algorithm service

Requirement	Specification
Perform calculations on SQL results	Create generic functions that can calculate averages and sum and other basic calculations Recognize what to calculate from message input Generalize for different types of datasets

C.1.0.6 Graph service

Requirement	Specification
Convert data to graphs Convert the graphs to PNG	Graph type based on input

C.2 Protocol messages

```
1
2  // The sidecar definition.
3  service SideCar {
4    rpc InitRabbitMq(InitRequest) returns (google.protobuf.Empty) {}
5    rpc Consume(ConsumeRequest) returns (stream SideCarMessage) {}
6    rpc SendRequestApproval(RequestApproval) returns  (google.protobuf.Empty) {}
7    rpc SendValidationResponse(ValidationResponse) returns  (google.protobuf.
         Empty) {}
8    rpc SendCompositionRequest(CompositionRequest) returns  (google.protobuf.
         Empty) {}
9    rpc SendSqlDataRequest(SqlDataRequest) returns  (google.protobuf.Empty) {}
10   rpc SendTest(SqlDataRequest) returns  (google.protobuf.Empty) {}
11   rpc SendMicroserviceComm(MicroserviceCommunication) returns  (google.protobuf
         .Empty) {}
12   rpc CreateQueue(QueueInfo) returns  (google.protobuf.Empty) {}
13 }
14
15 message InitRequest {
16   string service_name = 1;
17   string routing_key = 2;
18   bool queue_auto_delete = 3;
19 }
20
21 message QueueInfo {
22   string queue_name = 1;
23   bool auto_delete = 2;
24 }
25
26 message ConsumeRequest {
```

```
27    string queue_name = 1;
28    bool auto_ack = 2;
29  }
30
31  message SideCarMessage {
32    string type = 1;
33    google.protobuf.Any body = 2;
34    map<string, bytes> traces = 3;// Binary or textual representation of span
          context
35  }
36
37  message Auth {
38    string access_token = 1;
39    string refresh_token = 2;
40  }
41
42  message DataProvider {
43    repeated string archetypes = 1;
44    repeated string compute_providers = 2;
45  }
46
47  message ValidationResponse {
48    string type = 1;
49    string request_type = 2;
50    map<string, DataProvider> valid_dataproviders = 3;
51    repeated string invalid_dataproviders = 4;
52    Auth auth = 5;
53    User user = 6;
54    bool request_approved = 7;
55  }
56
57  message User {
58    string id = 1;
59    string user_name = 2;
60  }
61
62  message RequestApproval {
63    string type = 1;
64    User user = 2;
65    repeated string data_providers = 3;
66    bool sync_services = 4;
67  }
68
69  message AcceptedDataRequest  {
70    string type = 1;
71    User user = 2;
72    Auth auth = 3;
73    map<string, string> authorized_providers = 4;
74  }
75
76  message CompositionRequest {
77      string archetype_id = 1;
78      string request_type = 2;
79      string role = 3;
80      User user = 4;
81      repeated string data_providers = 5;
82      string destination_queue = 6;
83      string job_name =7;
84  }
85
86  message SqlDataRequest {
87    string type = 1;
88    repeated string data_providers = 2;
```

```
  string query = 3;
  bool graph = 4;
  string algorithm = 5;
  map<string, string> algorithm_columns = 6;
  User user = 7;
  RequestMetada request_metada = 8;
}

// The sidecar definition.
service Microservice {
  rpc SendData(MicroserviceCommunication) returns (google.protobuf.Empty) {}
}

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;
  RequestMetada request_metada = 6;
  map<string, bytes> traces = 7;// Binary or textual representation of span
      context
}
service Health {
  rpc Check(HealthCheckRequest) returns (HealthCheckResponse);
  rpc Watch(HealthCheckRequest) returns (stream HealthCheckResponse);
}

message HealthCheckRequest {
  string service = 1;
}

message HealthCheckResponse {
  enum ServingStatus {
      UNKNOWN = 0;
      SERVING = 1;
      NOT_SERVING = 2;
      SERVICE_UNKNOWN = 3;  // Used only by the Watch method.
  }

  ServingStatus status = 1;
}
service Generic {
    rpc InitTracer(ServiceName) returns (google.protobuf.Empty) {}
  }

message RequestMetada {
    string correlation_id = 1;
    string destination_queue = 2;
    string job_name = 3;
    string return_address = 4;
}

message ServiceName {
    string service_name = 1;
}
service Etcd {
  rpc InitEtcd(google.protobuf.Empty) returns (google.protobuf.Empty){}
  rpc GetDatasetMetadata(EtcdKey) returns (Dataset) {}
}

message EtcdKey{
  string path = 1;
```

```
}
message Dataset {
  string name = 1;
  string type = 2;
  string delimiter = 3;
  repeated string tables = 4;
}
```

Listing C.1: protocol buffer messages

C.3 Example API body

```
{
    "type": "sqlDataRequest",
    "user": {
        "ID": "1234",
        "userName": "jorrit.stutterheim@cloudnation.nl"
    },
    "dataProviders": ["VU","UVA","RUG"],
    "syncServices" : true
}
```

Listing C.2: Example message body for requestApproval

```
{
    "type": "sqlDataRequest",
    "query" : "SELECT * FROM Personen p JOIN Aanstellingen s LIMIT 10",
    "graph" : true,
    "algorithm" : "average",
    "algorithmColumns" : {
        "Geslacht" : "Aanst_22, Gebdat"
    },
    "user": {
        "id": "1234",
        "userName": "jorrit.stutterheim@cloudnation.nl"
    },
    "requestMetadata": {
        "jobId": "jorrit-stutterheim-a4ec6229"
    }
}
```

Listing C.3: Example message body for sqlDataRequest