# Scattered Directive: A Distributed and Automated Digital Data Marketplace Infrastructure

Jake Jongejans
*Complex Cyber Infrastructure*
*University of Amsterdam*
Amsterdam, The Netherlands
0009-0002-8324-1821

Alexandros Koufakis
*Complex Cyber Infrastructure*
*University of Amsterdam*
Amsterdam, The Netherlands
0009-0009-3946-9627

Ana Oprescu
*Complex Cyber Infrastructure*
*University of Amsterdam*
Amsterdam, The Netherlands
0000-0001-6376-0750

***Abstract*—Harnessing the power of data is paramount to unlocking more effective applications, especially with the rise of machine learning. Data regulation, both through legislation and company policies, makes it impractical for different companies to learn from multiple datasets at once. *Digital Data Marketplaces* aim to solve this issue, by providing a platform where organisations can exchange data based on (dynamic) policies. The digital data marketplace enforces these policies, allowing only data exchange that does not violate policies.**

**This work considers *distributed* digital data marketplaces. We design and implement Scattered Directive, a state-of-the-art distributed infrastructure deployment utilising FABRIC, capable of automatically deploying a digital data marketplace platform. Through numerous improvements and fixes, we provide a stable, widely applicable and distributed digital data marketplace platform.**

***Index Terms*—digital data marketplaces, FABRIC, distributed DDMs.**

## I. Introduction

Data is the new gold, and interest in it is sky-high. As the world explores new technologies such as Machine Learning (ML) in many different sectors of the industry from healthcare [1] to self-driving cars [2] and as we explore the possibilities of data within research, data is becoming the cornerstone of progress. The more complementary data you have, the better ML algorithms become, the more you can learn from experiments, and much more.

Legislation such as the GDPR [3], HIPAA [4] and COPPA [5], as well as internal company policies define what is allowed and what is not. When these laws, regulations and policies change, manual labour is required to adhere to these new policies. Mistakes are easily made when performing manual labour, and data integrity suffers.

A Digital Data Marketplace (DDM) platform is able to utilise the power of policies to dynamically adapt the usage of infrastructure and data-sharing methods. It does this based on the policy agreements defined between participants or *agents* within the system. These policy agreements form data exchange *archetypes*, idiomatic definitions of data distribution and exchange permissions. With a set of allowed archetypes, DDM platforms can utilise different ways of facilitating these, such as through network configurations or communication moderation and service orchestration [6, 7]. However, we find a lack in automated, distributed deployments for such systems that can be utilised readily.

### A. Research questions

This work sets out to design and implement a distributed DDM, with a focus on automation and wide applicability. This forms the research goal as follows, with three research questions:
*How feasible is an automated and fully distributed policy-driven DDM?*

**RQ0**: How can a DDM be made generally applicable?
**RQ1**: What is required for an automatic deployment of DDMs on distributed infrastructure?
**RQ2**: To what extent can a DDM be deployed on distributed infrastructure?

### B. Contributions

In this work, we describe how we developed Scattered Directive, which defines a automated and reproducible environment for Digital Data Marketplaces. Scattered Directive's high degree of automation offers a reliable and easy to use environment that is widely applicable for multiple research directions (e.g. automated policy evaluation, distributed infrastructure, and expansion on DDMs). Moreover, we describe how we prepared the infrastructure for its distributed deployment (spanning multiple continents) on the FABRIC testbed[1]. Moreover, Scattered Directive uses the DYNAMOS[2] middleware to demonstrate the policy-aware DDM application. The main contributions of this work can be summarized as follows:

**C0**: We configured and implemented a geographically distributed kubernetes-based Digital Data Market infrastructure on the FABRIC testbed.
**C1**: We developed a fully automated deployment procedure for the DYNAMOS middleware to highlight the policy-aware DDM capabilities of Scattered Directive.
**C2**: We made the code of the developed Scattered Directive open source[3].

[1] portal.fabric-testbed.net/
[2] github.com/Jorrit05/DYNAMOS
[3] github.com/Javernus/Scattered-Directive

## II. Background

### A. Digital Data Marketplaces & Data Exchange Policies

A DDM is a place where entities can securely share their resources with other participants within the marketplace. Using a set of data exchange policies, the DDM determines whether resources are allowed to be sent or be requested from one party to another. Only if the policies allow this to take place, does the marketplace allow the actual sharing of resources [6, 8, 9].

Two kinds of resources are generally distinguished: *software* and *data*. These resources are distinguished for their origin can differ when considering a computation. Next to software and data, the *output* of a computation is considered a third resource.

Data exchange policies are a type of contract within a DDM, describing what is allowed to be shared in terms of data and output, or used in terms of software. Furthermore, policies can affect what executions need to take place on data before sharing it, such as anonymisation algorithms. A set of related policies together form an agreement, which could state a certain set of organisations is okay with certain data to be shared with each other [9].

When a DDM works with static or predefined policy agreements, the DDM and policy enforcement systems have to be defined *once*, based on the given agreement definitions [6]. The reality is more muddy with dynamic policies agreements. Whether new legislation defines different rule sets surrounding the usage and sharing of the data in question, or companies change their own policies within the agreement, a DDM should adapt to it and take these new policies into consideration. As Shakeri *et al.* [6] show, it is possible to translate policies within a DDM to a network configuration to restrict the sharing of data.

Data sharing archetypes define how different resources within a DDM can flow. Shakeri *et al.* define data sharing archetypes in [6], defining the nine two-party data sharing archetypes and three example three-party data sharing archetypes. Stutterheim *et al.* utilise these archetypes in DYNAMOS, utilising various forms of data sharing archetypes to dynamically adapt where data is and is not allowed to go based on policies. Utilising these archetypes and being able to model these with containers/microservices paves the way to creating a functioning DDM, using a policy-aware middleware.

### B. Architecture

DYNAMOS (Dynamically Adaptive Microservice-based OS) is a data exchange middleware system. In previous work, DYNAMOS has been used to implement core Data Distribution Management functionalities and provides a fertile ground for research on distributed DDM systems and infrastructure automation. Key features of DYNAMOS include its policy evaluator component, which ensures compliance with predefined regulations, and its Kubernetes-based microservice architecture, which supports scalable, modular, and resilient deployment of distributed data exchange workflows. To explain the DYNAMOS system, there are a few key terms to highlight.

- **Agent**: An agent is a party within the data exchange that owns a collection of data, and is willing to exchange its data according to a set of policies.
- **Trusted third party**: A trusted third party does not necessarily own a collection of data, but provides the infrastructure or software as a service and is trusted by its users to handle data properly.
- **Nodes**: a node is a physical or virtual machine that is part of the DDM and is tied to a single agent or trusted third party.
- **Policy enforcer**: the policy enforcer is a service within DYNAMOS that validates whether a request may be executed according to a set of predefined regulations and agreements.
- **Orchestrator**: the orchestrator is a service within DYNAMOS that instructs the relevant nodes from agents and third parties to setup a microservice chain (described below) adhering to the allowed archetypes.
- **Sidecar**: DYNAMOS utilises the sidecar design pattern to facilitate communications between microservices and core DYNAMOS services.
- **Ephemeral Jobs**: ephemeral jobs are microservice containers that run a single time, destructing themselves after their computation is complete.
- **Microservice chain**: a microservice chain is a set of microservices that has a defined communication pathway that can be described as a directed acyclic graph (i.e. a graph without cycles). DYNAMOS's current implementation is a one-to-one link between each microservice.

DYNAMOS is a system that can create different microservice chains for its operation. It creates service execution pipelines that adhere to the defined policies, where each step in the chain represents data being exchanged—either from one entity to itself or to another entity. This means that it can replicate data exchange archetypes. However, an important note is that there are more advanced operations that can be done within DYNAMOS. For example, a microservice chain can be setup to execute anonymisation of data before sharing it with another agent or third party. Multiple services can thus be chained into a microservice chain to execute more advanced operations. Furthermore, it can adapt itself to changes in policies, creating different chains based on these requirements [7]. These benefits come from using microservices. They do not require knowledge about their context, so long as their interface is defined properly. This allows them to run in various environments without issue [10].

Figure 1 illustrates the DYNAMOS' architecture. The orchestrator and policy enforcer live within the core orchestrator layer of DYNAMOS. This layer handles receiving a request, validates whether the request is allowed to execute and instructs the relevant agents to create services to execute it if allowed. The exchange layer encompasses the messaging service and the knowledge base that holds information about

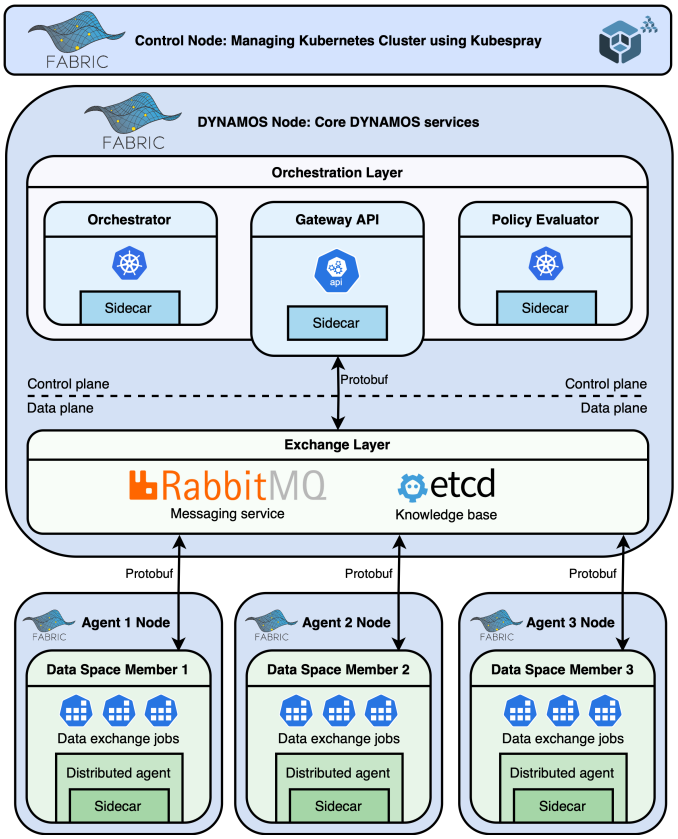


Fig. 1. The DYNAMOS architecture, as running on different FABRIC nodes. FABRIC nodes are represented by the blue boxes, and the respective DYNAMOS services and agents are represented inside of each of these FABRIC node boxes.

the state of the system, including the policy agreements. The agents and third parties are shown at the bottom.

An important limitation within the DYNAMOS system is the inability to create a cross-node microservice chain. This limitation is due to a core design choice in DYNAMOS: microservice chain jobs are spawned on a single node. This design choice stems from the symmetric use-case used to validate the initial DYNAMOS PoC [7]. It was assumed that a request would not require resources to flow between different agents and their respective nodes. Instead, each node runs identical code on their own respective data, providing responses from multiple agents separately.

## III. Related Work

This work spans core elements of *distributed infrastructure* and *policies*. We highlight a small number of the most relevant related works.

### A. Distributed Infrastructure

FABRIC [11] is a programmable research testbed that provides the necessary infrastructure for this work. Their work describes how they design and create a research infrastructure that allows for the configurability and programmability of the entire testbed, both at the edge—i.e. the individual nodes—and the network itself. On top of that, it provides researchers with powerful hardware. With this, novel research can be done in many areas, where network measurements, control, and the ability to create custom network designs are all provided.

Another DDM is Nokia's Data Marketplace [12]. Their platform facilitates secure, real-time data and AI model sharing across enterprises and communications service providers. Nokia focusses on the monetisation of data, utilising blockchain technology for the backbone of their secure data sharing platform.

### B. Policies

The Amsterdam Data Exchange (AMdEX) [13] is working on creating a digital data exchange, turning legal contracts into a digital form that can be enforced[4]. Next to this, they are working together with the University of Amsterdam and SURF[5] (Dutch Academic IT Cooperative) to create a *Research Data Exchange*, where universities can more easily share data based on legal contracts, or apply statistical analysis and only receive the output[6].

Data-sharing policies and archetypes are defined in more detail by Shakeri *et al.* [6], defining the foundation of policy definitions where software, data and output as distinguished, and different constellations of ownership determine the possibilities in terms of archetypes. Furthermore, their work considers how a DDM can be defined through the use of network overlays, harnessing the power of programmable infrastructure to enforce policy agreements.

In the EPI framework [14], Kassem *et al.* work toward a *privacy-by-design* framework that is aimed specifically at the healthcare sector. In this sector, data is incredibly sensitive and can only be shared with explicit permission. However, the enforcement of this comes with constraints and restrictions, one of which being that consent to share data implies the existence of data, which is sensitive data in its own regard. They utilise eFLINT [15], a formal language for normative reasoning, to define policies and reason with them.

## IV. Scattered Directive

Scattered Directive is a distributed and automated DDM infrastructure, providing a foundation on which research can be executed. Scattered Directive provides the means for an efficient and simplified way for distributed deployment of both Kubernetes clusters and DDMs. Core principles of Scattered Directive are *automation* and reproducibility, which result in faster deployment and reduced variance between deployments.

Fig. 2. The Logo for Scattered Directive.

[4]amdex.eu/
[5]surf.nl/en
[6]amdex.eu/news/sharing-research-data-under-ones-own-conditions/

To create Scattered Directive, we utilise the research testbed FABRIC and the policy-aware middleware DYNAMOS[7]. FABRIC [11, 16] is an adaptive, programmable research infrastructure that allows us to experiment with powerful compute at scale, where we can control both compute and network resources programmatically, across continents. DYNAMOS is a policy-aware middleware that implements a DDM that can *dynamically* change how, where and to whom data is allowed to flow given changes in policy agreements.

### A. FABRIC

This is a dedicated, programmable server and network research infrastructure, allowing repeatable experiments to be executed across 33 sites. We see three main benefits in using FABRIC: programmability, site locations and networking [11, 16].

The FABRIC network and server infrastructure is programmable. The FABRIC API, provide the means to create and configure nodes, network interfaces, network connections, add-in cards and more. However, all the configuration can also be done automatically using Python. FABRIC's programmable infrastructure is simple, intuitive and debuggable. Everything from the nodes to the network is accessible through the API, and changes can be made easily [11, 16].

The second benefit of FABRIC is the extensive number of site locations it includes. FABRIC operates sites across the world, from Europe to the United States of America to Asia. While Cloud providers offer more than the set of 33 sites that FABRIC offers, the benefits of the FABRIC infrastructure outweigh this drawback.

Finally, the networking. FABRIC's networking infrastructure utilises dedicated connections between sites to offer connection speeds of up to 1.2 Tb/sec. This dedicated fiber network means that any experiments are not bottlenecked by insufficient network connections, which provides cleaner data for experiments that may be run and a stabler DDM service.

Other FABRIC benefits include standardised and powerful hardware configurations for all of their servers[7], allowing a wide set of options for diverse experiments. This allocation is also dedicated, and no trickery is used to over-allocate servers, and allocations are not moved to other hardware to avoid overusage of resources.

### B. Cluster Management

For the Kubernetes cluster we need for deploying our DDM, we have a handful of requirements, partly inspired by our core principle of *automation*:

**KCR0**: The cluster must be deployable on a wide range of target machines:
- FABRIC nodes must work.
- On-premise machines should be supported.
- Cloud instances preferably work.

**KCR1**: The cluster setup should be easy to deploy.

**KCR2**: The cluster deployment should be as automated as possible.

**KCR3**: The cluster setup should be performant.

With these requirements, we are able to determine the best tools for our use-case. This starts with the consideration between a Kubernetes-focussed operation system and Kubernetes installers. While the former hardens the operating system to only work for Kubernetes, decreasing the attack vector, support for these operating systems is not ubiquitous. FABRIC only provides a limited set of operating system images, which means that this option does not work for our use-case based on **KCR0**.

The second option is Kubernetes installers. Some provide a Kubernetes binary for a single machine, while others also create a cluster and set up networking for you. In terms of setup, the former requires one to first install Kubernetes with the installer and then define the cluster through some other method. Kubernetes installers that also manage the cluster creation make many of the decisions for you, providing configuration that is proven to work as default. Based on **KCR1**, we have decided to utilise a Kubernetes *cluster* installer.

We opt to use *Kubespray*. Kubespray is a Kubernetes cluster management tool that utilises kubeadm[8] under the hood and works with Ansible playbook files, making setup easy. Kubespray works on any target machines and automates many parts of the setup. There are alternatives, but they lack in some areas. kOps focusses on Cloud clusters, and also handles provisioning of Cloud resources for you, but would not work with FABRIC or on-premise machines (**KCR0**). Rancher makes setting up clusters simple through their UI integration, going against **KCR2**.

One slight issue with setting up Kubernetes is the recent departure from their Google-hosted repositories in favour of Kubernetes own, community-owned repositories.[9] Because of the recency of the change, many Kubernetes guides still refer to the old Google repositories, as did FABRIC's own Kubernetes example up to recently[10].

### C. Configuration and Network Challenges

FABRIC manages its own networking, and utilises advanced techniques to make its nodes secure. Furthermore, it uses IPv6 instead of IPv4. This means that the network configuration needed to be dialled in to work.

For inter-node communication, we assign a network interface to each node, and create a Layer 3 (L3) network between the nodes. This provides an IPv4 network between the network interfaces of the nodes that lets nodes communicate with each other. FABRICS provides other options for L3 networks, but these are either IPv6, allow pods to be accessed on the internet, or act as a VPN, all of which is not necessary for our use-case.

With a L3 network, we are able to assign a network interface (a shared Mellanox ConnectX-6 card) to each node, which

[7] learn.fabric-testbed.net/knowledge-base/fabric-site-hardware-configurations

[8] kubernetes.io/docs/reference/setup-tools/kubeadm/

[9] kubernetes.io/blog/2023/08/15/pkgs-k8s-io-introduction/

[10] We brought up this issue with the FABRIC team and they resolved the issues.

is given an IPv4 address. We utilise this IP address in the creation of the cluster in the Ansible playbook file that we create. Furthermore, our network interface is given a specific name, which needs to be indicated to the container network interface plugin so it tries to contact other nodes through the right interface. Each interface is then linked to the network.

The default container network interface plugin, *calico*, had issues with connecting the nodes in a cluster. A lot of configuration attempts are made to get calico to comprehend the network topology of FABRIC and communicate between nodes, but they are all in vein. By switching to Flannel, another container network interface plugin that is simpler and more straight-forward to setup, we solve these networking issues. Calico provides more features and is generally more performant compared to Flannel, which related to **KCR3**, so it would have been the preferred choice if we were able to configure it properly.

Configuring Flannel is easy, as it only needs to have the physical network interface name configured. We utilised a regular expression to match with the correct network interface names.[11] The rest of its configuration happens on its own. The small trade-off in terms of performance (**KCR3**) make it possible to adhere to **KCR1** and **KCR2** much better.

### D. Intercontinental Deployments

For the intercontinental Kubernetes cluster on FABRIC nodes, FABRIC's documentation states that their L3 network supports "any number of interfaces across any number of sites".[12] This is not the case. Requesting a network interface across multiple sites resulted in an error message that the network type is limited to work on a single site only.

There are multiple ways to solve this challenge. The first is to create Layer 2 (L2) site-to-site networks between each of the nodes. This would result in a lot of configuration overhead and thus may impact performance (**KCR3**). The second solution was discussed in the FABRIC forum[13]. This solution considers exposing the right subnets of L3 networks at each site to each other. We create multiple L3 networks at every separate site, and utilise the subnets and gateways of these networks to add IP routes to each other network on all nodes.

### E. Integrating DYNAMOS

DYNAMOS runs natively on Kubernetes clusters, which eases deployment. Not everything is smooth sailing, however. The original DYNAMOS implementation is not suitable for distributed deployments.

First of all, DYNAMOS utilises persistent volume claims for certain services to persist relevant data, which lacked the respective persistent volumes to claim from. We assume these were automatically being generated by the Docker Kubernetes Engine on local hardware. Simply defining these explicitly solved this issue.

The second change is one of dependency management and resource efficiency. We noticed a singular dependency on Docker to hash a RabbitMQ password. This code can be seen in Listing 3. We replaced this with a simple bash script (Listing 4), alleviating the dependency and increasing start-up efficiency.

```
rabbit_pw=$(openssl rand -hex 16)
hashed_pw=$($SUDO docker run --rm \
            rabbitmq:3-management rabbitmqctl \
            hash_password $rabbit_pw)
actual_hash=$(echo "$hashed_pw" | cut -d $'\n' -f2)
```

Listing 3. The original Bash code for generating the RabbitMQ password, running a Docker container to hash the password.

```
function encode_password() {
S=$(od -A n -t x -N 4 /dev/urandom)
P=$S$(echo -n $1 | xxd -ps | tr -d '\n' | tr -d ' ')
P=$(echo -n $P | xxd -r -p | sha256sum | head -c128)
P=$(echo -n $S$P | xxd -r -p | base64 | tr -d '\n')
echo $P
}
rabbit_pw=$(openssl rand -hex 16)
actual_hash=$(encode_password $rabbit_pw)
```

Listing 4. The Bash code for generating the RabbitMQ password. No docker containers are run in this code. `P` stands for `Password` and `S` for `Salt`.

Prometheus and Grafana were not starting up properly either. Utilising `kube-prometheus-stack` solved this issue. This stack provides defaults that are proven to work, and deployed without issue. It provides tooling for Prometheus, Grafana and Loki in one, further reducing configuration. With this simplicity and automation, it aligns with our core principle.

These changes make DYNAMOS boot and execute without error. However, agents and third parties are still not assigned to the proper FABRIC nodes that they belong to (e.g. the DYNAMOS or an agent's FABRIC node). For a functioning distributed DDM, each organisation must have ownership over its respective FABRIC nodes and its services. So should the DYNAMOS core, running the orchestrator, policy enforcer, and API gateway. With updates in the configuration files, this is resolved (Listing 5). Similarly, the persistent volumes had to be constrained to their respective nodes (Listing 6).

```
    nodeSelector:
      kubernetes.io/hostname: <node-name>
```

Listing 5. Assigning a pod to a node can be done through a `nodeSelector` definition within the `yaml` files.

Finally, the services running inside of DYNAMOS needed to be assigned to the respective nodes, too. This touched on

[11] The regular expression is `/enp[5-9]s\d/`, which accepts any network interface starting with `enp`, then a digit above 4, an `s` and then any number. While this regular expression makes some assumptions, we find it works properly in all of our test cases on different FABRIC nodes.

[12] learn.fabric-testbed.net/knowledge-base/network-services-in-fabric/

[13] learn.fabric-testbed.net/forums/topic/inter-as-intra-as-routing-test/

```yaml
  nodeAffinity:
    required:
      nodeSelectorTerms:
        - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
              values:
                - <node-name>
```

Listing 6. Assigning a persistent volume to a node cannot be done through a `nodeSelector` definition, and must therefore be done using a `nodeAffinity` definition within the `yaml` files.

other systems within DYNAMOS, and required similar configuration as the services for the agents and the DYNAMOS core (Listing 7).

```
job := &batchv1.Job{
    ObjectMeta: metav1.ObjectMeta{ ... },
    Spec: batchv1.JobSpec{ ... ,
        Template: v1.PodTemplateSpec{
            ObjectMeta: metav1.ObjectMeta{ ... },
            Spec: v1.PodSpec{ ... ,
                NodeName:   <node-name>,
}   }   }   }
```

Listing 7. Assigning a microservice job to a node can be done by adding a `NodeName` specification into the `Job` object, as seen on line 8, which is done in the Go code for DYNAMOS's `agent` service.

These changes combined allow DYNAMOS to be deployed on the Kubernetes cluster running on FABRIC, assigning relevant services to dedicated nodes. The assignment of DYNAMOS services to FABRIC nodes is demonstrated in Figure 1. Five FABRIC nodes (blue boxes) run their respective DYNAMOS services or DYNAMOS agents.

### F. Automation

Scattered Directive focusses on automation. From a single configuration object, the cluster and DDM are both defined and configured in full. Nodes can be defined with a role (e.g. the DYNAMOS system or an agent, or Kubernetes control), resources (CPU, RAM, storage) and location (based on FABRIC sites). These are automatically conglomerated into one Kubernetes cluster and the DYNAMOS DDM is installed and configured based on the aforementioned roles. Through new automated scripts, DYNAMOS can be configured with a set of agents and third parties, without requiring manual work. This further generalises DYNAMOS, removing the example use-case within the original version.

Most automations are portable bash scripts, interfacing directly with the nodes, reducing reliance on FABRIC and allowing these scripts to be ported to other systems easily. This allows Scattered Directive to adapt to the needs of its users. Furthermore, we automatically install K9s[14]. K9s is a Kubernetes management tool that works inside the terminal and comes pre-installed on Scattered Directive nodes. This allows easy insights into Kubernetes clusters to check the status of Kubernetes pods and containers, see the logs of specific containers, terminate certain jobs and more.

[14]k9scli.io/topics/install/

### G. Introducing Generic Requests

With Scattered Directive providing a DYNAMOS deployment, we focus our attention on making DYNAMOS more widely applicable. The first change we make is generalising the requests in DYNAMOS. Requests are what fuel DYNAMOS, so we find it imperative that researchers can implement any kind of request. This touches upon the entire codebase, but the main change is in relation to the new `Request` type.

```
type SqlDataRequest struct {
        state            protoimpl.MessageState
        sizeCache        protoimpl.SizeCache
        unknownFields    protoimpl.UnknownFields

        Type             string
        Query            string
        Algorithm        string
        AlgorithmColumns map[string]string
        Options          map[string]bool

        User             *User
        RequestMetadata  *RequestMetadata
}
```

Listing 8. The (simplified) definition of the SQL data request within the original DYNAMOS codebase. The `struct` hardcoded the possible request parameters within it.

We move from the `SqlDataRequest` (Listing 8) toward a `Request` type (Listing 9). By using a generic name and abstracting the hardcoded request parameters into an arbitrary `Data` parameter, developers are given the freedom in data definitions for each request type.

```
type Request struct {
        state           protoimpl.MessageState
        unknownFields   protoimpl.UnknownFields
        sizeCache       protoimpl.SizeCache

        Type            string
        Data            map[string]*structpb.Value

        User            *User
        RequestMetadata *RequestMetadata
}
```

Listing 9. The new generic definition of the DYNAMOS `Request` type. There are no hardcoded parameters, and instead, a `Data` field is provided with a map of `Values` as its type, which can be seen as an arbitrary dictionary of data that can be attached.

The changes for this must be propagated through the DYNAMOS system, such as in the RabbitMQ definitions. Usage of the type also differs now, but in a meaningful way. Before, data could be fetched from the predefined parameters (e.g.

```
sqlDataRequest = rabbitTypes.SqlDataRequest()
msComm.original_request.Unpack(sqlDataRequest)
result = execute_request(sqlDataRequest.query)
```

Listing 10. The original (simplified) SQL Query code that utilised the `SqlDataRequest` type with its `Query` definition.

```
request = rabbitTypes.Request()
msComm.original_request.Unpack(request)

try:
    query = request.data["query"].string_value
    result = execute_request(query)
except Exception as e:
    handleError(e)
```

Listing 11. The (simplified) SQL Query code that utilises the new `Request` type with its `Query` data within the `Data` parameter. The extra error-handling when the value or its `string_value` is missing is required, as the type definition now does not provide type-safety.

query, see Listing 10). Now, developers must be considerate of missing data, and error checking is required (Listing 11).

These changes to the request type make it possible for developers to more easily create new requests and define their own data structure within the request, instead of misusing parameters, which can cause confusion.

### H. Persistency of Services

DYNAMOS utilises *ephemeral* microservices for executing requests. This has security benefits as every execution has a new environment to run in, so no data or state could affect subsequent runs. However, we find that this decision can have big negative influences. For one, when working with a Vertical Federated Learning (VFL) use-case [17], this would imply that machine learning models need to be serialised, deserialised and reinitialised for every round, creating a massive communication and computation overhead. Furthermore, for stability improvements covered later, being able to *ping* a service without it shutting down is very beneficial. The persistency of services that we propose can be considered weakly persistent. Services are only persistent for as long as they are required for a single request, and are still terminated afterwards.

To enact this change, the DYNAMOS system must now keep track of which services are running. DYNAMOS must now only start a new service if it doesn't already exist. Through the already implemented job identifiers, it is possible to distinguish between running services and determine whether the service was already running.

This implementation lead to a peculiar issue. When sending multiple requests to the same microservice, the first request sent to agents would always be received, and any subsequent requests made to the agent would return an error code: `502 Bad Gateway`. An inspection of the full DYNAMOS pipeline does not provide a clue why this happens, as there is no built-in logic for denying a second request which has been authorised.

The crux was found within the agent and service definitions. These definitions, as provided by the initial DYNAMOS repository and its documentation, underspecified the service selector, which defined the TCP port that was used for communicating with its agent. Whenever a microservice was launched and operational, this service would match with both the agent itself and the microservice job, which means that it would either not function at all (because it had two matches) or it tried to round-robin and the requests sent to the microservice job were ignored because it was not made to handle these requests.

The solution is to specify the selector of the service in more detail. By adding an additional `type` label to refer to the agent as `type: agent`, we avoid the problem of matching multiple services. This change not only enables persistent services now, that can handle multiple requests, but it also ties into the general stability of DYNAMOS. Due to this underspecification, it was impossible to send in a second request to DYNAMOS while the first was running. With this fix, however, this issue has been resolved.

With persistency allowed within DYNAMOS, the services can now define new request handlers for utility functions. The first one is the `shutdownRequest`, which tells a microservice that its weak persistency period has ended and it should exit its program.

### I. Stability Improvements

DYNAMOS has issues surrounding service start-up and lifespan. Firstly, services are set to shut down after a set lifespan, meaning long-running requests are aborted. Second of all, heavy services may take a long time to start up, meaning requests to that service are timed out and the request fails.

The lifespan can be considered a safe-guard for the system that when services are not shut down properly, they do so themselves after a suitable time period. Therefore, the best solution is to increase the lifespan. This can be done statically (one lifespan for all services) or dynamically, based on service configuration. We opt for the former option, as this system is a fallback and safe-guard foremost, and additional configuration is another point of failure. Based on long-running tasks such as a VFL workflow [17], we find a 200 minute lifespan suitable and updated the original 1 minute lifespan with this new value.

To improve the stability when it comes to the start-up delay of microservices, another utility function is implemented in services, the ping service. This allows the DYNAMOS core system to validate that the service is started before it sends the request to it. This is only possible thanks to the persistency that we implement, which allows multiple request calls to be made to a single service.

## V. Experimental Design & Results

Scattered Directive is tested using a custom VFL workflow built on top of it [17]. This workflow included the execution of subsequent requests (automated through new functionality). It was running with the Kubernetes control node, DYNAMOS, one VFL client and the VFL server in Amsterdam, one client

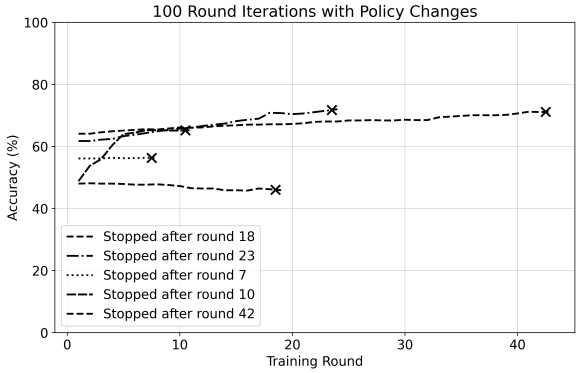


Fig. 12. The accuracy over time for five 100 round runs, with a policy change occurring midway through. The crosses correspond to when the policy within DYNAMOS was altered, where the third client left the agreement. None of the five runs continued with another training round after this point. The label indicates after which round the policy was changed.

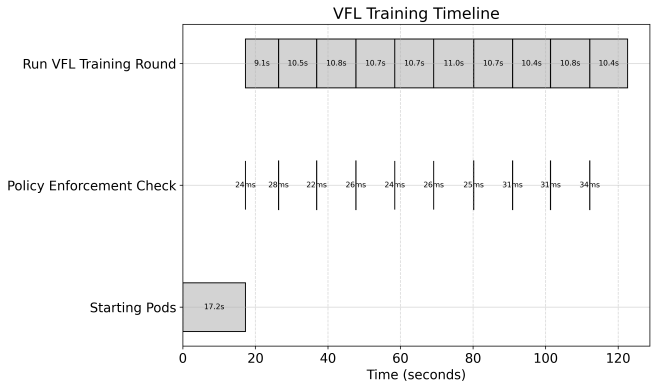


Fig. 13. The duration of each separate operation of the VFL training request.

in Los Angeles and one in Tokyo. The Kubernetes control node has 2 cores and 8GB RAM, while the rest of the nodes had 4 cores and 16GB RAM.

The VFL workflow tests [17] run for up to 100 rounds, where subsequent requests are made to each client. During a random moment *after* the first five training rounds, and *before* the fiftieth training round, we execute this policy change. This policy change indicates that the third client left the agreement. As the then-running training round is already validated, we expect that the subsequent training round is rejected, as it violates the new policy agreement. We run five iterations of this, and find the request is always rejected after the policy change (Figure 12).

Execution time analysis (Figure 13) shows the time that each part of the system took to execute. This considers a VFL request (100 rounds truncated to the execution time for the first 10 rounds). Start-up takes a long time, due to the size of the service. Furthermore, the policy enforcement check has very limited effect on the execution time, which is mostly comprised of the actual request.

## VI. Discussion

The main research goal of this work is to determine *how feasible an automated and fully distributed policy-driven DDM platform is*. We answer three research questions to determine this:

**RQ0**: How can a DDM be made generally applicable?
**RQ1**: What is required for an automatic deployment of DDMs on distributed infrastructure?
**RQ2**: To what extent can a DDM be deployed on distributed infrastructure?

We find three areas in which DYNAMOS was able to be made more generally applicable (Finding 1). The first is in its deployment surrounding the agent setup. Through automation and removing the example use-case, DYNAMOS is more easily applicable and adaptable. Secondly, we consider the request type. Through a generic request implementation, we make it possible to more easily define the data that requests require, without having to rework the internal system. Thirdly, we implement *persistent* services. This persistency allows for more efficient computation (such as in VFL workflows [17]) and more advanced execution flows, and it helps with stability improvements.

**Finding 1:** DDM aspects surrounding automation, generic requests, persistency and service lifespans needed consideration to make the system more generally applicable.

Next to these applicability changes, we make stability changes (Finding 2). We increase the lifespan of request services to facilitate long-running requests. This makes it possible for researchers to implement workflows more easily that take longer than 60 seconds to complete, such as a VFL workflow. Next to that, through the use of persistency and new logic, we gracefully handle the start-up of request services, allowing heavy services to load without the system receiving a timeout.

**Finding 2:** Through stability improvements to start-up and for long-running requests, the DDM runs better and the research experience is improved.

The automation in Scattered Directive took considerable effort, both for the Kubernetes cluster and the DYNAMOS integration (Finding 3). Through debugging, the use of a different container network interface (Flannel) and network exposure for multi-site clusters, a Kubernetes cluster is now automatically deployable through Scattered Directive. Moreover, The FABRIC network significantly increased the complexity of the automation process.

**Finding 3:** A Kubernetes cluster is automatically deployable through Scattered Directive, despite the challenges from the advanced FABRIC network.

The distributed deployment of the DYNAMOS DDM is also possible (Finding 4). This requires even more changes, both with a focus on automation, and on proper deployment behaviour in the distributed context. DYNAMOS needed updates surrounding volatile configuration (lack of explicit persistent volume definitions, Prometheus stack not deploying properly), improper consideration for service deployment in distributed deployments (i.e. running services on their respective FABRIC nodes), and the lack of scripts for automatic definition of agents within the system.

**Finding 4:** Through extensive rework of the DYNAMOS system, deployment on distributed infrastructure is possible.

Through the concise configuration file defined in Scattered Directive, it is possible to deploy the entire system across the thirty-three FABRIC sites. The entire process bases itself on this configuration, from where to deploy the DYNAMOS core system, and setting up all the required network and cluster infrastructure. This seamless experience empowers researchers to focus on their own experiments (Finding 5).

**Finding 5:** Through automation based on simple configuration, Scattered Directive removes the overhead of handling Kubernetes cluster deployment and DDM deployment from researchers.

With these findings, we answer the research questions defined. We find through our implementation of Scattered Directive that it is certainly feasible to deploy an automated and fully distributed policy-driven DDM platform.

### A. Threats to validity

Distributed infrastructure is not easy, and the choice in physical infrastructure can greatly affect the stability of the platform and introduce possible bottlenecks to what is running on top of it. We strived toward stable infrastructure, both in terms of large hardware allocations, as with the choice for FABRIC for its truly dedicated hardware allocations. We believe we have made sure these threats did not materialise, but cannot be $100\%$ certain.

Next to this, we have not been able to test the limits and extensibility of the scalability of Scattered Directive. While the platform works for resource-intensive requests such as an VFL workflow, scalability tests have not been performed. We have improved the inner workings of DYNAMOS, removing a bottleneck that made running multiple requests impossible. Future work could investigate the performance of the system under high-concurrency requests or with a significantly larger number of participating clients, which would stress the message bus and the policy enforcer.

### B. Future work

There are several avenues that future research building on this study could consider:

- Integrating a formal language (e.g. eFLINT [15]) with DYNAMOS, harnessing the power of automated validity checking of policies.
- Introducing ephemerality of services and persistency of data across services is another area of interest. This can further harden the security of the system, while allowing for cross-service usage of data (within a single agent's node). Tying these features into policy agreements is also possible, allowing policies to dictate whether services may utilise persistent data, or run in a persistent manner.
- Research into microservice chain creation can lead to DYNAMOS being capable of running complex directed, acyclic graphs (DAGs) of services, further improving its applicability.

## VII. Conclusion

We set out to evaluate *how feasible an automated and fully distributed policy-driven DDM platform is*. With the design and implementation of Scattered Directive, an automated and reproducible DDM infrastructure, we show that this is **fully feasible**, backed by our results. Our tests demonstrate its validity and functionality. Policy enforcement is also shown to work properly. Therefore, we conclude that it is completely feasible to automatically deploy a distributed policy-driven DDM platform.

## References


[1] A. Beam and I. S. Kohane, "Big data and machine learning in health care.," *JAMA*, vol. 319 13, pp. 1317–1318, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:45552660.

[2] Z. S. Dhaif and N. K. E. Abbadi, "A review of machine learning techniques utilised in self-driving cars," *Iraqi Journal for Computer Science and Mathematics*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:274171685.

[3] European Parliament and Council of the European Union. "Regulation (EU) 2016/679 of the European Parliament and of the Council," of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). (May 4, 2016), [Online]. Available: https://data.europa.eu/eli/reg/2016/679/oj.

[4] *Summary of the HIPAA Privacy Rule*, https://www.hhs.gov/hipaa/for-professionals/privacy/laws-regulations/index.html, Public Law 104-191, U.S. Statutes at Large 110 Stat. 1936, Aug. 1996.

[5] United States Congress, *Public law 105-277: Omnibus consolidated and emergency supplemental appropriations act, 1999*, https://www.govinfo.gov/content/pkg/PLAW-105publ277/pdf/PLAW-105publ277.pdf, Stat. 112 pt. 2681 (signed into law on October 21, 1998), Oct. 1998.

[6] S. Shakeri, L. E. Veen, and P. Grosso, "Evaluation of container overlays for secure data sharing," *2020 IEEE 45th LCN Symposium on Emerging Topics in Networking (LCN Symposium)*, pp. 99–108, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:232149550.

[7] J. Stutterheim, A. Mifsud, and A. Oprescu, "DYNAMOS: Dynamic Microservice Composition for Data-Exchange Systems, Lessons Learned," in *2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C)*, ISSN: 2768-4288, Jun. 2024, pp. 8–15. DOI: 10.1109/ICSA-C63560.2024.00008. [Online]. Available: https://ieeexplore.ieee.org/document/10628120/.

[8] T. M. van Engers, R. Meijer, L. Gommans, and K. Nieuwenhuis, "Trusted big data sharing for aircraft mro using a secure digital market place mechanism," NWO/STW Workshop "ICT with Industry 2016", Lorenz Centre Leiden, Tech. Rep., Nov. 2016, Prof. dr. Tom M. van Engers (UvA), Prof. dr. Robert Meijer (UvA, TNO), Dr. ing. Leon Gommans (Air France KLM Group ICT Technology Office R&D, UvA), Dr. Kees Nieuwenhuis (Thales Nederland B.V., CTO Office). [Online]. Available: https://dl4ld.nl/2017-02-24-Secure-Virtual-Digital-Market-Places.pdf.

[9] S. Shakeri, "Secconnet: Smart and secure container networks for trusted big data sharing," Promotores: Prof. dr. ir. C.T.A.M. de Laat and Prof. dr. P. Grosso, PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, 2024, ISBN: 978-94-6483-983-8. [Online]. Available: https://dare.uva.nl/personal/pure/en/publications/secconnet(840b1af4-0d54-43fe-83e0-f60cf4142305).html.

[10] N. Dragoni *et al.*, "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:14612986.

[11] I. Baldin *et al.*, "FABRIC: A national-scale programmable experimental network infrastructure," *IEEE Internet Computing*, vol. 23, no. 6, pp. 38–47, 2019.

[12] Nokia, *Nokia launches blockchain-powered data marketplace for secure data trading and ai models*, Press release (Newsroom), Nokia Data Marketplace is a blockchain-based platform enabling trusted exchange and monetization of datasets and AI models across industries, May 2021.

[13] L. T. van Binsbergen, M. Oost-Rosengren, H. Schreijer, F. Dijkstra, and T. van Dijk, "Amdex reference architecture–version 1.0. 0," *by L. Thomas van Binsbergen*, 2024.

[14] J. A. Kassem, T. Müller, C. A. Esterhuyse, M. G. Kebede, A. Osseyran, and P. Grosso, "The epi framework: A data privacy by design framework to support healthcare use cases," *Future Gener. Comput. Syst.*, vol. 165, p. 107 550, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:274273931.

[15] L. T. van Binsbergen, L.-C. Liu, R. van Doesburg, and T. van Engers, "eFLINT: A domain-specific language for executable norm specifications," in *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE 2020, New York, NY, USA: Association for Computing Machinery, Nov. 16, 2020, pp. 124–136, ISBN: 978-1-4503-8174-1. DOI: 10.1145/3425898.3426958. [Online]. Available: https://dl.acm.org/doi/10.1145/3425898.3426958.

[16] K.-C. Wang *et al.*, "A glimpse of emerging networking and distributed computing research via experiments on the fabric testbed," in *Proceedings of the 34th International Conference on Computer Communications and Networks (ICCCN)*, Presented at ICCCN 2025, Chicago, IL, USA: IEEE, 2025, pp. 1–9.

[17] J. Jongejans, "Scattered directive: A distributed and automated digital data marketplace infrastructure for policy-driven vertical federated learning," English, Supervised by A.M. Oprescu. Program: FNWI MSc Software Engineering., Master's thesis, Faculteit der Natuurwetenschappen, Wiskunde en Informatica, Universiteit van Amsterdam, Amsterdam, The Netherlands, 2025. [Online]. Available: https://scripties.uba.uva.nl/search?id=record_56506.