Contract No. H2020 – 826172

SEMANTICS FOR PERFORMANT AND SCALABLE INTEROPERABILITY OF MULTIMODAL TRANSPORT

D3.3 – Design of Architecture, Testing Infrastructure, Test Cases and Benchmarks of the IF (C-REL)

Due date of deliverable: 31/10/2019

Actual submission date: 24/04/2020

Leader/Responsible of this Deliverable: UPM

Reviewed: Y

Document status		
Revision	**Date**	**Description**
1	15/09/2019	Initial table of contents
2	30/09/2019	Revised table of contents
3	03/10/2019	Testing infrastructure material + revision of table of contents
4	10/10/2019	Reorganized content of the draft
5	24/10/2019	Revised reorganization of the document
6	05/11/2019	Final contributions received and document reorganised for final QA
7	20/11/2019	Final version after TMC approvaland Quality Check
8	17/04/2020	Final QA
9	24/04/2020	Final version after TMC approval and quality check

Project funded from the European Union's Horizon 2020 research and innovation programme		
Dissemination Level		
PU	Public	X
CO	Confidential, restricted under conditions set out in Model Grant Agreement	
CI	Classified, information as referred to in Commission Decision 2001/844/EC	

Start date of project: 01/12/2018 Duration: 25 months

EXECUTIVE SUMMARY

This deliverable describes the output of Tasks 3.3 and 3.4 of the SPRINT project, for what concerns the C-REL milestone. In particular, the deliverable:

- Provides an overview of the Shift2Rail Interoperability Framework (S2R IF) and its components, so as to provide a self-contained description in a single document as well as to update the existing description of the S2R IF with the work done in these tasks.
- Suggests a range of possible architectural alternatives to satisfy the requirements identified in deliverable D3.2.
- Describes the designed logical and technical infrastructure to be used for the testing of the performance and scalability of two of the components of the S2R IF, namely the federated SPARQL query processor and the Converter.

Therefore, the main contributions presented in this deliverable are: (i) the proposal and analysis of different architectural options that would make it possible to implement the S2R IF (as discussed in section 3), with a special focus on the core components being addressed in SPRINT: the asset manager, the converter and the federated SPARQL query processor; and (ii) the design of a benchmark that allows testing the scalability and performance of the federated SPARQL query processor and the Converter under a combination of materialised and virtual RDF datasets, what is expected to appear in the context of the S2R IF (as discussed in section 4).

It is important to note that this deliverable corresponds to the C-REL milestone, and therefore does not provide final decisions on the architectural components nor a final set of testbeds and benchmarks to assess all the S2R IF components that are addressed in SPRINT. This will be included in the corresponding F-REL milestone deliverable.

ABBREVIATIONS AND ACRONYMS

Abbreviation	Description
ADMS	Asset Description Metadata Schema
AM	Asset Manager
CQRS	Command Query Responsibility Segregation
DCAT	Data Catalog Vocabulary
DCAT-AP	DCAT Application Profile
GTFS	General Transit Feed Specification
IF	Interoperability Framework
IT	Information Technology
OBDA	Ontology Based Data Access
OBDI	Ontology Based Data Integration
RDF	Resource Description Framework
RML	RDF Mapping Language
R2RML	RDB to RDF Mapping Language
xR2RML	eXtended R2RML
S2R	Shift2Rail
RDF	Resource Description Framework
WSDL	Web Services Description Language

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

This deliverable provides two main contributions that are associated with the C-REL milestone of the SPRINT project.

The first contribution is focused on the proposal and analysis of different architectural options that would make it possible to implement the Shift2Rail Interoperability Framework (S2R IF). First, a general set of descriptions for a preliminary design of the S2R IF is provided. Taking that into account, as well the previous identification work that was done in D3.2 “Performance and Scalability Requirements for the IF (C-REL)”, several recommended patterns have been analysed in the view of how they may be applied for the S2R IF. The pros and cons of each of these patterns in the context of the development of the S2R IF are provided in Section 3, with a special focus on the core components being addressed in SPRINT: the asset manager, the converter and the federated SPARQL query processor.

Next, in Section 4, we discuss about the design of a benchmark that allows testing the scalability and performance of two of these components, the federated SPARQL query processor and the Converter. This benchmark is inspired by previous works in the state of the art in the area of performance and scalability testing of SPARQL endpoints, as well as ontology-based data access solutions, given the focus on the use of converters in S2R IF, which may use a combination of materialised and virtual RDF datasets. We describe the datasets, queries and mappings that have been created as a testbed, based on the use of original data from GTFS feeds from the transport authority of Madrid (so as to provide a neutral while transport-related testbed) that are scaled up at different scales. We also describe the technical environment to be used for the testing infrastructure.

2. OUR STARTING POINT: OVERVIEW OF THE ORIGINAL INTEROPERABILITY FRAMEWORK ARCHITECTURE

Figure 1 provides an overview of the original architecture and internal components of the Shift2Rail Interoperability Framework (S2R IF), which we use as a starting point for the work presented in Section 3 in this deliverable. This architecture has been already presented in previous deliverables of SPRINT.

As it happens with many other similar architectures, we present an architecture that is divided into different layers (in this case, two). The Shared Data Layer includes a collection of data from heterogeneous data sources of travel services in the transportation sector. In this layer, a variety of data exists such as standards and specifications, transportation ontologies, code lists, etc. The Service Layer conceives a unified and smooth collaboration among various travel service providers as well as consumers of such services, including S2R IP4 applications.

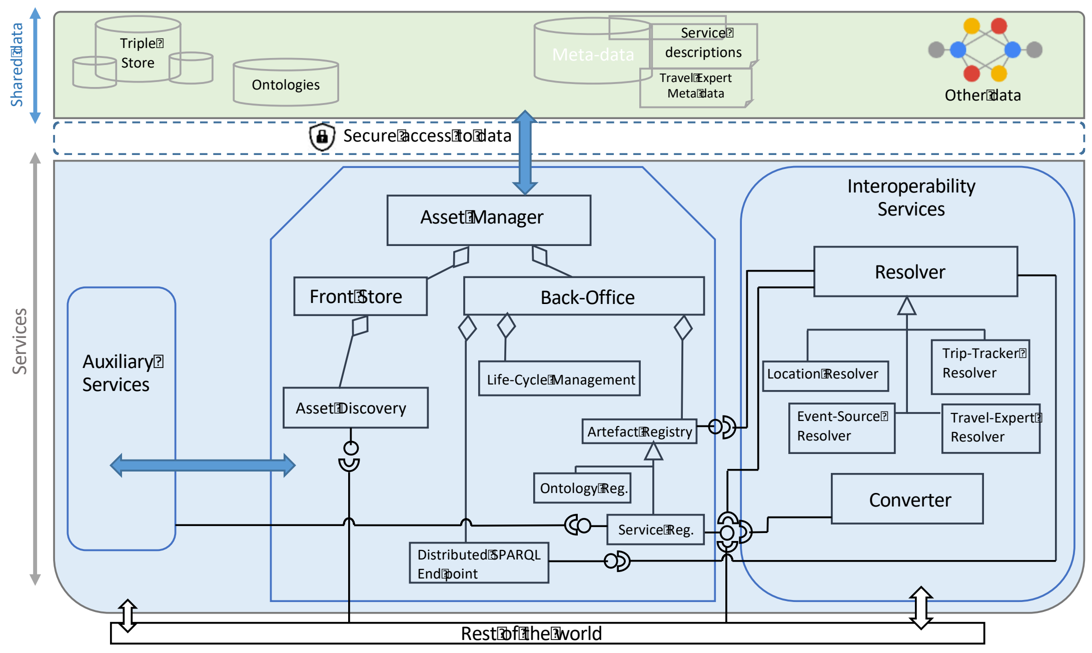


Figure 1. Internal Components of the Original Shift2Rail Interoperability Framework

Since the Data Layer accommodates a large heterogeneity of data in the context of the S2R IF and one of the goals of IF is to overcome barriers to data exchange in the transportation ecosystem, in our work we will focus on three of the IF components: Asset Manager, Distributed SPARQL Endpoint and Converter. The Asset Manager allows publishing, sharing, discovering and managing various artifacts that may be published/utilized by external clients and other internal components of the IF. The Distributed SPARQL Endpoint (aka federated SPARQL query processor) evaluates SPARQL queries over a set of materialised or virtual SPARQL endpoints that may belong to different organizations, providing a unified access to a complementary set of knowledge graphs. The Converter acts as an adapter between two distinct formats and is able to map the information expressed in one format to the other.

In Section 3, we will discuss on the different architectural alternatives that have been identified to design such an interoperability framework and will conclude with a preliminary design of the foreseen reference S2R Interoperability Framework, which will continue to be refined during the execution of the SPRINT project.

3. DESIGN ALTERNATIVES FOR THE SHIFT2RAIL INTEROPERABILITY FRAMEWORK

3.1 Analysis of Design Alternatives

An initial draft of the software architecture for the S2R IF can be based on the components identified in deliverable D3.1, describing the main macro features of the *Service* and *Data* layers. Such features, defining the high-level functional requirements, should be further designed according to a wide set of architectural patterns to guarantee the accomplishment of the non-functional requirements described in D3.2.

Architectural principles and approaches determine the way features are assigned to components, and the way each component exchanges information with the others, and therefore choosing a specific approach can heavily affect both the implementation, the quality of service and the possible future adoption of the solution.

In this section, we analyze how different architectural styles can affect the various components and layers that had been identified in D3.1 and that were presented in Section 2. We then perform an initial choice that will drive the implementation efforts leading to C-REL, according to the principles of choosing an effective architecture and encouraging industrial adoption.

The analysis performed in D3.1 investigates architectural patterns for distributed systems, comparing different deployment architectures and evaluating them on different dimensions to point out the advantages and shortcomings of each solution. According to the evaluation done, a modular service-oriented approach and, more specifically, a microservice architecture seems the more promising approach. A microservice architecture is based on a set of loosely-coupled and collaborating services, but designing the software architecture in detail requires to address several issues and to take several design decisions.

Considering the pattern language proposed in [1], we proceed to analyze different recommended patterns to solve typical issues in microservice architectures and to avoid anti-patterns in defining the IF reference architecture. Furthermore, we discuss the pros and cons of each pattern with respect to the IF functional and non-functional requirements. We will focus on patterns dealing with:

- **Decomposition strategy**: this topic deals with how to decompose an application into services and therefore on how and with what granularity features are assigned to components.
- **Data management**: this topic deals with how data are kept consistent among services and with how data are queried.

- **Communication**: this topic deals with how components exchange information with other components and the users, and how different components can be orchestrated to implement higher-level features.
- **Deployment**: this topic deals with how components are deployed on the physical distributed infrastructure.
- **UI**: this topic deals with the choice on how to provide a unique interface served by multiple services.

To support design choices in determining a reference architecture, we will describe the patterns and we will explain the consequences of applying a specific pattern discussing benefits, drawbacks and issues.

3.1.1 Decomposition strategy

Decomposition strategy is the first design decision in defining a microservice architecture and deals with the fundamental task of identifying services and assigning them functional requirements. A good decomposition strategy should guarantee a functional decomposition assigning to each service a set of focused and cohesive responsibilities. The accomplishment of this task is not trivial, but it is extremely important to facilitate development, to reduce complexity and to guarantee independent horizontal scalability of services.

Two different strategies are proposed as patterns to approach this issue: *decomposition by business capability* and *decomposition by subdomain*. The former proposes clustering functionalities according to the business logic and activities the application should perform, the latter a clustering based on the specific subdomains identifiable in the domain addressed.

Elaborating on this distinction with respect to the IF we obtain two feasible approaches:

- *Cross-Asset Decomposition*: in this scenario service decomposition is by business capability and therefore each service offers a specific functionality identified for the IF.
- *Asset-Dedicated Decomposition*: in this scenario service decomposition is by subdomain and therefore each service is specialized on a single type of asset and provides all functionalities related to the specific asset type.

To provide a concrete example considering the Lifecycle Management feature, adopting the first approach a single service will provide this functionality for all type of assets, adopting the second one the service related to a specific asset type is also responsible for all the different management aspects related to its lifecycle.

A *cross-asset* approach should ensure configurability for different asset types without introducing too much complexity or degradation of performances in relation to an increasing number of assets. On the other hand, an *asset-dedicated* approach can optimize functionalities development and performances on the specific asset, but it can cause duplication of similar sub-services in the system. For these reasons, considerations to choose a preferred approach are related to the relative expected growth in the number of assets and functionalities in the IF, and to the overlapping logic required to offer similar functionalities to different types of assets.

An additional aspect is related to the expected deployment configuration. To favour industrial adoption, it may be useful to define a minimum installation template to lower the barriers in entering the ecosystem. In this direction, the discussion should consider the minimum set of services required for an IF installation and therefore if it is more reasonable to reduce a set of *asset-dedicated* or *cross-asset* services.

Although a consistent choice of decomposition strategy is preferred, also a mixed approach can be a feasible choice. Indeed, in an architecture based on a *cross-asset* decomposition, it may be necessary to opt for an *asset-dedicated* decomposition of a specific functionality requiring a strictly custom logic for each type of asset.

To conclude the discussion on decomposition strategies, an orthogonal aspect to the patterns listed should be pointed out: the granularity of decomposition. To design a good strategy, it is extremely important to determine at which level of detail we should split functionalities into separated services. A relevant example considering the IF is related to the Converter component, a low-granularity strategy can integrate the converters functionalities in a single service implementing the conversion pipeline. However, a higher-granularity strategy can split the converter into several services, each one implementing a single step of the pipeline and allowing converter technology providers to independently scale performance-critic functionalities.

The cross-aspect decomposition strategy seems to be the most applicable. Putting in place an Asset-dedicated decomposition would require replicating several key components which are usually capable of providing features for all the different assets alone. Replicating such components for each single asset type would be a waste of resources, since that would mean having each replica (each one consuming system resources) managing a very small amount of assets. Applying a cross-asset decomposition on the opposite side will allow reusing server components like process engines, RDF databases and object storage servers, which will then be configured to provide “per-asset type” features. Such approach will also likely improve adoption of the IF, as it will be easier to integrate it with the software already adopted by Transport operators.

3.1.2 Data management

Data management patterns focus on three critical issues in distributed systems: Data Storage, Data Consistency and Data Querying. Designing a microservice architecture most services need to persist their data, but it is important to still guarantee loosely-coupled, isolated and independently-scalable components.

The main design choices are related to the overall organization of the data layer, i.e., between the *Shared Database* pattern and the *Database per Service* pattern*.*

The simplest solution to cope with data management is the *Shared Database* pattern. In this scenario, multiple services share an external database that ensures ACID transaction enabling consistency and querying of up-to-date data. This pattern offers a straightforward solution, but it introduces coupling between services that, depending on the application requirements (e.g. in data-intensive applications), may become a too strict constraint especially with respect to performance.

To strictly adhere to principles advocated by a microservice architecture a *Database per Service* pattern should be implemented. Each service has its own database enabling separated development and management of data structures and keeping required coordination among services only based on interfaces. However, it is important to point out that this requirement does not imply each service should have its own database server, it is enough that each service manages its own data structures. In any case, this is what the S2R IF is proposing with the use of a federated SPARQL query processor when it comes to querying the data.

This second pattern avoids introducing coupling between services but, on the other hand, presents different issues that require further design decisions among different patterns.

The first issue is related to *Data Querying*. Having a database for each service means application data are split and the problem of querying data resembles typical issues of querying distributed databases. To cope with this problem, we can employ two strategies:

- *API Composition* pattern: a set of APIs is defined to hide data fragmentation. The service implementing those APIs knows which service contains data needed and performs a set of queries and an in-memory join to compute the query result and give-back a response to the caller.
- *Command Query Responsibility Segregation* (CQRS) pattern: a view of data required for a specific type of queries is materialized and keep up-to-date processing events emitted by different services owning the data.

The second issue is related to *Data Consistency*. Separated databases cannot guarantee consistency of data as a unique local transaction. To solve this problem, we need to employ a *Saga* pattern implementing a single transaction in a sequence of local transactions and coordinating them. Two different coordination strategies can be used:

- *Choreography*: each service publishes an event as a result of its own local transaction triggering the correct local transaction in other services. It is important to notice that also error events and amendment actions should be handled.
- *Orchestration*: a specific object is responsible to coordinate the different services and knows at each moment which local transaction should be performed. This is what is normally done by a federated SPARQL query processor, for instance.

Choreography and *CQRS* are patterns based on the definition of events that should be carefully designed. Often these types of patterns are combined with the *Domain Event* pattern that prescribes the organization of business logic on a set of objects, based on domain-specific concepts (Domain-Driven Design), that emit domain events when created or published.

A further pattern inherently based on events to maintain data consistency is the *Event Sourcing* pattern. The main idea of this pattern is to keep consistency relying on the atomic concept of event. The state of an object is not directly stored and updated somewhere, instead the sequence of events related to an object is stored allowing to reply events and to reconstruct the current state avoid inconsistencies. Optimizations are often used, e.g., periodic snapshots of object state, but this approach requires a non-trivial programming mindset to optimize complexity and performances.

Considering IF requirements, a shared database is a feasible scenario since data are mainly accessed in reading mode and few concurrent writings are expected. This type of approach can keep the architecture simpler and help programmers approaching an IF installation in integrating existing infrastructure, e.g. an already in use database server, with the IF. Patterns described for the *Database per Service* pattern can, however, offer suggestions to implement a distributed network of IF installations requiring separated databases to guarantee scalability.

Data Heterogeneity

Beyond general issues already mentioned, a specific aspect related to data management is how the IF deals with *Data Heterogeneity*. The main component involved is the IF Converter, which is the component that adapts and maps two distinct formats allowing interoperability for different actors in the IF ecosystem.

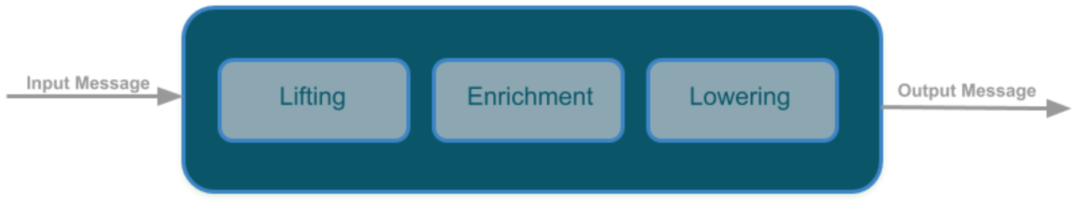


Figure 2. General process followed by a Converter to transform input data into output data

In the ST4RT project[1], the Converter (whose general process is depicted in Figure 2) exploits an annotation-based approach to translate Java objects representing data described in a source format into RDF graphs (*lifting*). Also, it is able to take an RDF graph to build instances of Java classes representing data described in the target format (lowering). Such approach is described in the literature as *materialisation*, since RDF triples are actually generated by lifting and added to a temporary, in-memory RDF repository to be queried and/or converted to a different format by mean of lowering procedures. Moreover, this approach easily allows for the enrichment of the initial dataset with other triples uploaded to the same RDF repository during the conversion, as depicted in Figure 3.

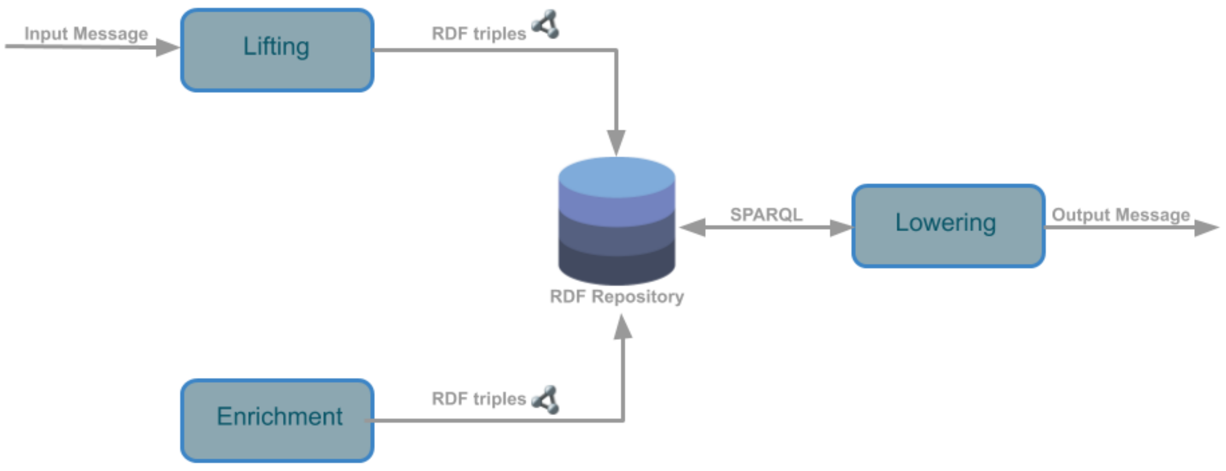


Figure 3. Materialisation approach for the Converter conversion pipeline

As discussed in D3.1, this high-level conversion pipeline can be implemented exploiting different tools and architectural patterns to model the different stages. In particular, the SPRINT project aims at extending the ST4RT Converter introducing a modular approach to favor the customization with respect to the different scenarios and the composability of conversion pipelines among different formats. This will be covered in Section 3.2.

However, the conversion problem is not simple and its requirements can be very different from case to case. In particular, data can be too large for implementing materialisation, and therefore another approach can be exploited using OBDA (Ontology Based Data Access) to deal with data heterogeneity by leveraging virtualisation techniques. In this regard, two different cases (depicted in Figure 4 and Figure 5) can be approached defining a Converter leveraging virtualisation techniques based on OBDA tools.

[1] http://www.st4rt.eu/

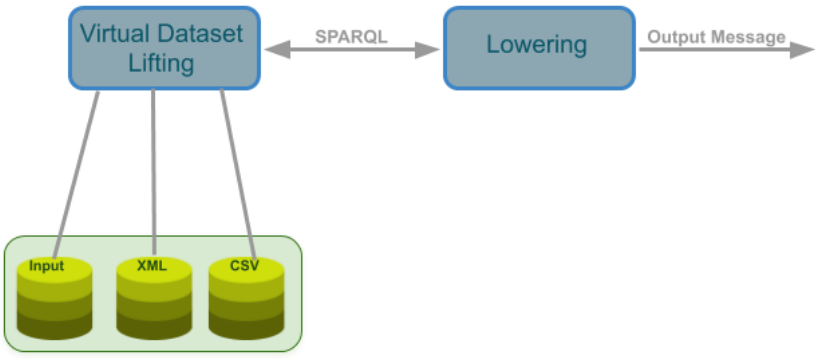


Figure 4. An approach for the virtualisation of datasets in the Converter pipeline

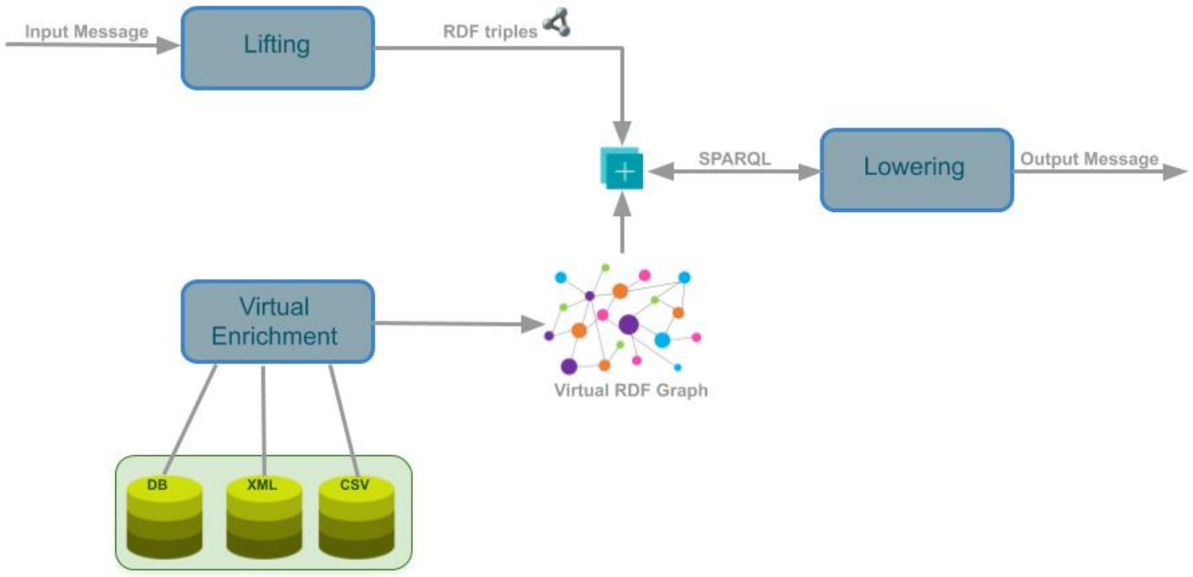


Figure 5. An alternative approach for the application of virtualisation in the Converter pipeline

The RML language can be considered as a good candidate to be included in the SPRINT converter because it does not require developers to modify existing source code; RML [2] is a mapping language defined to express rules that map data in heterogeneous structures to the RDF data model. This could be a benefit, since full access to source code is not always granted, and also because the requirement of having a Java class representation of the whole standard to be mapped could slow down the adoption of the solution.

Figure 6 presents an architecture for mapping datasets between RDF and other formats (XML, CSV, JSON and relational database). To adapt two formats, OBDA systems provide a unified access in terms of an ontology. In this architecture, the SPARQL query is specified over a virtual knowledge graph and then transformed into underlying query language of a data source. During the query translation, some optimisation techniques can be performed to generate a more efficient query. Finally, the result is translated to RDF when the transformed query is evaluated on the data source. Such approach can be used to implement the lifting phase of the conversion process in two different cases: when the input of the conversion problem is a large dataset in a batch-like scenario, or when a message conversion requires accessing large datasets.

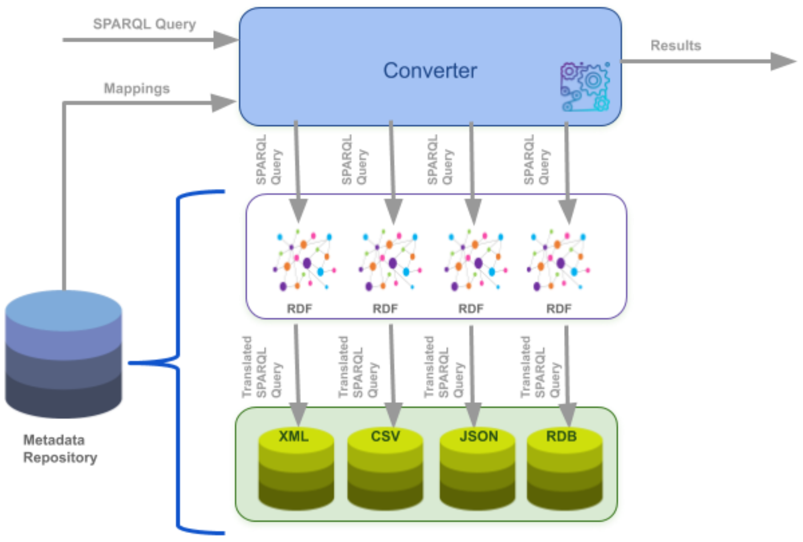


Figure 6. OBDA Architecture

3.1.3 Communication

Modularization of software applications is an architectural principle widely spread even before the definition of the microservice architecture pattern. The problem with classic modularization is that often the barriers among components are lowered during development, resulting in tightly bonded components. The success of the microservice architecture pattern is also related to the inherent modularization of services that, since they are developed and run separately, offer impermeable boundaries that are difficult to violate. This aspect guarantees most of the benefits of a microservice architecture, but it opens several design issues on the service discovery, on the communication between services, and on the interface offered to communicate with users external to the system.

3.1.3.1 Service Discovery

Service Discovery is an issue related to how components become aware of the presence of other services in the system. Moreover, service discovery should make service locations transparent with respect to dynamically changing addresses and multiple-instance services.

The solution to this problem is identified in the *Service Registry* pattern (which will be implemented by the Asset Manager, as described in section 3.2). A service registry is implemented storing, for each service, the list of instances and the updated related addresses. Each component needing a specific service shall consult the registry to obtain the up-to-date location. To improve the reliability of the information stored, the service registry may periodically health-check service instances to verify if they can handle requests.

This pattern can be associated with different strategies to determine how a client can discover the location of a service and how to register and unregister services from the registry.

Client discovery of services can be handled with the following patterns:

- *Client-side Discovery* pattern: the client itself exploits a registry-aware HTTP client that queries the service registry to obtain the service location.
- *Server-side Discovery* pattern: the client makes a request through a load balancer running at a known location and responsible to consult the service registry and forward the request.

Service registration can be handled with the following patterns:

- *Self-registration* pattern: the service itself is responsible to self-register on startup, to renew its registration periodically to confirm it is still alive and to unregister itself before shutdown.
- *3rd party registration* pattern: an external registrar is responsible to register and unregister services on startup and shutdown respectively.

Considering the IF, the location of main services can be configured statically during installation, since it is not supposed to change. However, if the deployment is made exploiting an orchestrated container-based system, the load balancing between different instances of a service and the DNS features are offered as basic functionalities ready to be used. On the other hand, dynamically changing services like converters, resolvers and other auxiliary services can exploit the asset manager as a sort of service-registry without health-check controls. In both cases, registration is not demanded to the specific service.

3.1.3.2 Communication Style

Communication style identifies different patterns to implement communications between services in a system. Three main methods can be implemented, and different services can employ more than one method to communicate with other services.

The *Remote Procedure Invocation* pattern adopts a request/reply-based protocol to make requests to services. This type of pattern comprises a wide set of possibilities: REST APIs, Remote Procedure Call (RPC) protocols and binary serialization systems (e.g., Thrift[2], ProtoBuf[3], Avro[4]).

[2] https://thrift.apache.org/

[3] https://github.com/protocolbuffers/protobuf/releases

[4] https://avro.apache.org/

The *Messaging* pattern exploits asynchronous communications over a messaging channel (e.g., Apache Kafka[5], RabbitMQ[6]). Different mechanisms can be implemented, the most common are:

- *Notification:* the sender sends a message without expecting a reply.
- *Request/Asynchronous response:* sender sends a request and expects a reply eventually.
- *Publish/Subscribe:* sender publishes a message related to a specific topic, zero or more recipients subscribed to the same topic receive the message.

The *Domain-specific protocol* pattern exploits a custom-defined protocol for inter-service communication.

In designing the communication mechanisms at each level in the architecture, a further design choice should be made on where to implement the business logic. A feasible option is to build a set of specific services offering specific functionalities to a client-service knowing the logic, making right service calls and merging responses to produce the expected result. Another option is to implement a set of higher-level services already implementing business logic and offering a response to a light client.

Considering the IF architecture, it seems not needed to identify or define a domain-specific protocol. Other communication means can be instead considered to implement different types of communication between services.

3.1.3.3 External Communication

External communication patterns deal on how features offered by a system are exposed to users hiding services fragmentation in the architecture.

The main pattern used for external communication is the *API Gateway* pattern that identifies a single entry point for all external client requests. The gateway is responsible to forward requests to the appropriate service adopting protocol translation if needed and merging results from several sub-requests if a composed answer should be provided. A possible variant of this pattern is called *Backends for Frontends* pattern and defines a different gateway for each frontend, e.g., mobile app, web-based, etc.

The IF should employ this pattern to facilitate the development of different frontends abstracting from the internal architecture.

[5] https://kafka.apache.org/

[6] https://www.rabbitmq.com/

3.1.4 Deployment

Once the set of services defining a microservice architecture are identified it is important to design the deployment strategy.

The straightforward solution is to apply the *Service Instance* or *Multiple Service Instances per Host* pattern deploying each service, possibly replicated, on a single host (physical or virtualized, e.g., Virtual Machine). This approach is feasible and should be employed whenever the architecture is composed of a small set of services requiring dedicated or a predictable amount of resources.

The preferred solution for microservice architectures is however based on the *Service Instance for Container* pattern. The usage of containers allows isolation of services as previous patterns, but exploiting lightweight virtualization techniques allows optimizing resource usage on each node packing multiple containers and guaranteeing easier scalability of services. Moreover, the employment of the *Service Deployment Platform* pattern, exploiting technologies like Kubernetes, allows for even easier management of services and containers offering the possibility of defining the desired state and taking actions to ensure it is guaranteed at any time.

Another option is the *Serverless* pattern that exploits managed services to hide any infrastructural concept and to allow specifying only the portion of code to be executed once a request is received. This approach is extremely scalable, but it poses several limitations on the overall architecture and on the input/output manageable by requests.

Considering the IF, different deployment strategies may be devised. For sure a microservice architecture better fits a container-based approach that should be considered as default option also to help optimization of resources and to comply with identified performance and scalability requirements. However, to increase adoption it may be useful to consider also host-based deployments that can be a better option to help integration with legacy systems. Moreover, a discussion on the best deployment pattern is also related to the scale of the IF, e.g. TSP-level or National-level, and to the optional components deployed that can determine different requirements (for example the definition of a runtime environment better fits a *Service Deployment Platform* pattern). A serverless approach instead does not seem to be an option due to the need of recurring to external providers and to their unpredictable costs.

3.1.5 User Interface

The last set of patterns is related to the issues in composing a user interface hiding the service fragmentation present in the architecture. Two possible options are:

- *Server-side Page Fragment Composition* pattern: each service is related to a component that returns the corresponding portion of interface (e.g., HTML fragment), then the different portions are aggregated server-side to fill in a specified UI template for the interface offered to the user.

- *Client-side UI Composition* pattern: each service is related to a client-side component that implements a portion of the interface. The different components are tied by a UI template that contains the various elements and composes the interface offered to the user.

The IF requirements do not impose any particular restriction on the choice of UI patterns.

3.2 Preliminary Design

In section we illustrate the preliminary design of the architecture of the S2R IF. The architecture, details and consolidates the concepts presented in Section 2, and it concretizes the principles laid down in Section 3.1, making suitable choices according to them.

Figure 7 illustrates the internal architecture of the S2R IF in more detail. As mentioned in Section 2, the architecture is based on two layers, a Data Layer (handled by the Triple Store subsystem shown in **Error! Reference source not found.**), and a Service Layer. In addition to data stores, there are two main components, namely **Asset Manager** and **User Manager**, which are designed to handle the various aspects of two entities of the system, Asset and Users, respectively.

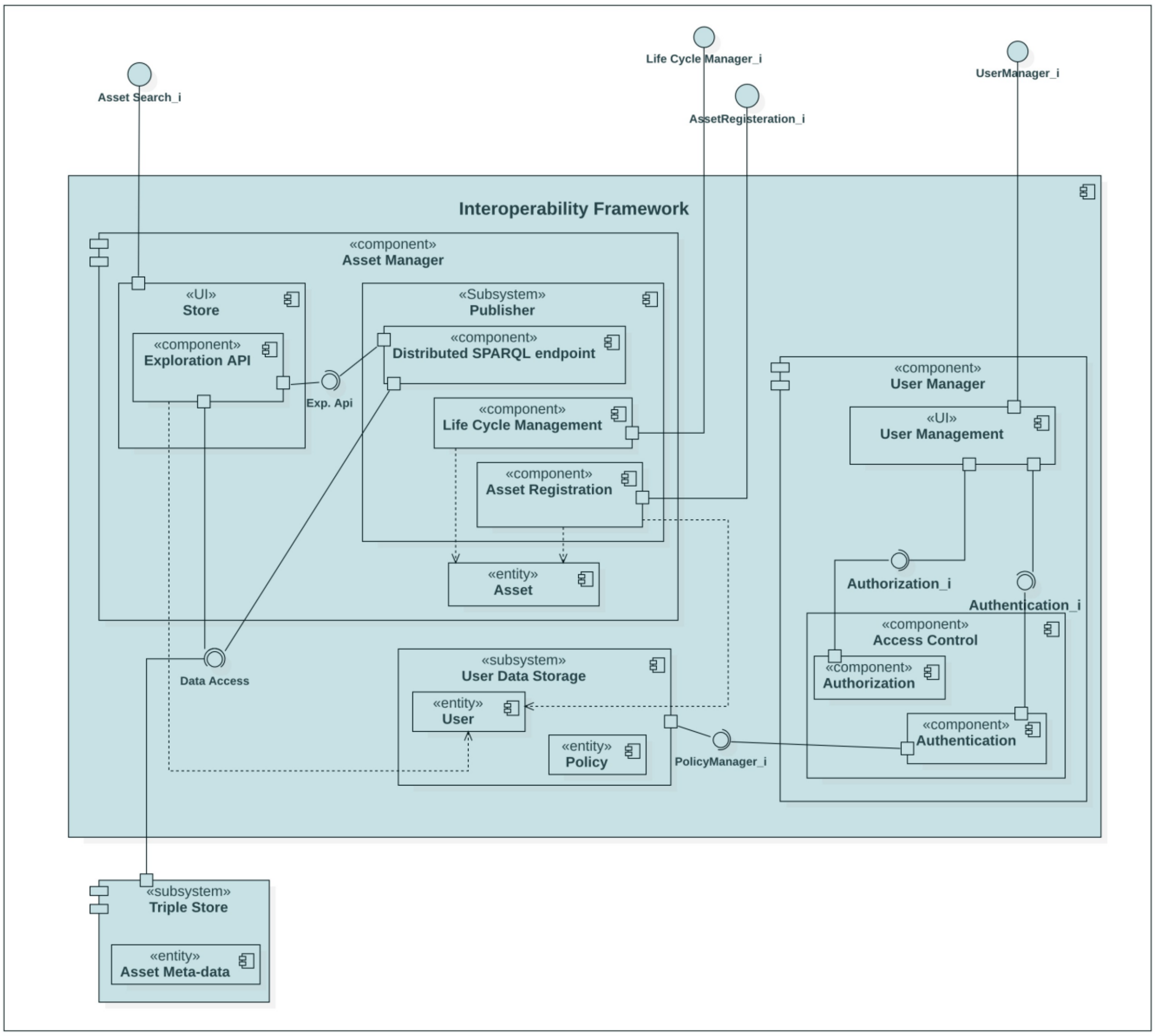


Figure 7. Component Diagram of the S2R IF

3.2.1 User Manager

As Figure 8 shows, another entity of the S2R IF is the User, which is handled by the User Manager component. A human user of the S2R IF may be an individual or a representative of a transportation operator. Apart from the Administrator Role, the possible interactions of Users with the IF may be encapsulated in two different logical roles, namely IF Provider Role and IF Consumer Role. IF providers are composed of a wide range of service/infrastructure providers in the transportation domain, including transport authorities, transport service providers, infrastructure managers, retailers and travel agency distributors [3]. Similarly, IF consumers are also transportation actors such as travel service providers, social networks, and IT suppliers and software applications.

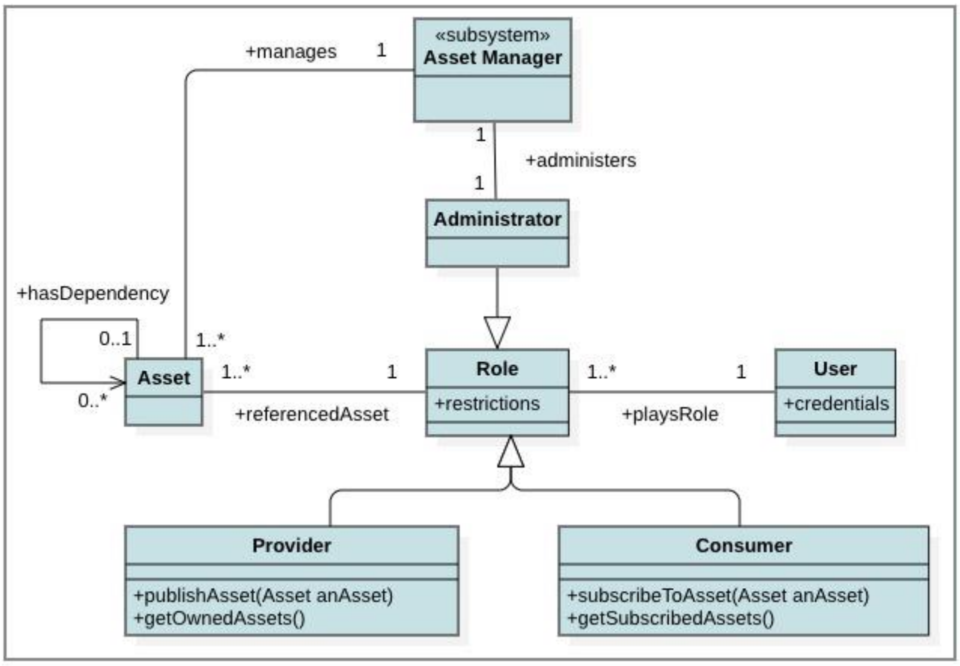


Figure 8. User Roles

The main responsibility of the User Manager is to provide the means for a user to register on the S2R IF and then govern the access rights and authorization for granting/denying various permissions to get access and operate with Assets based on the user's role and following a role-based access control mechanism.

3.2.2 Asset Manager

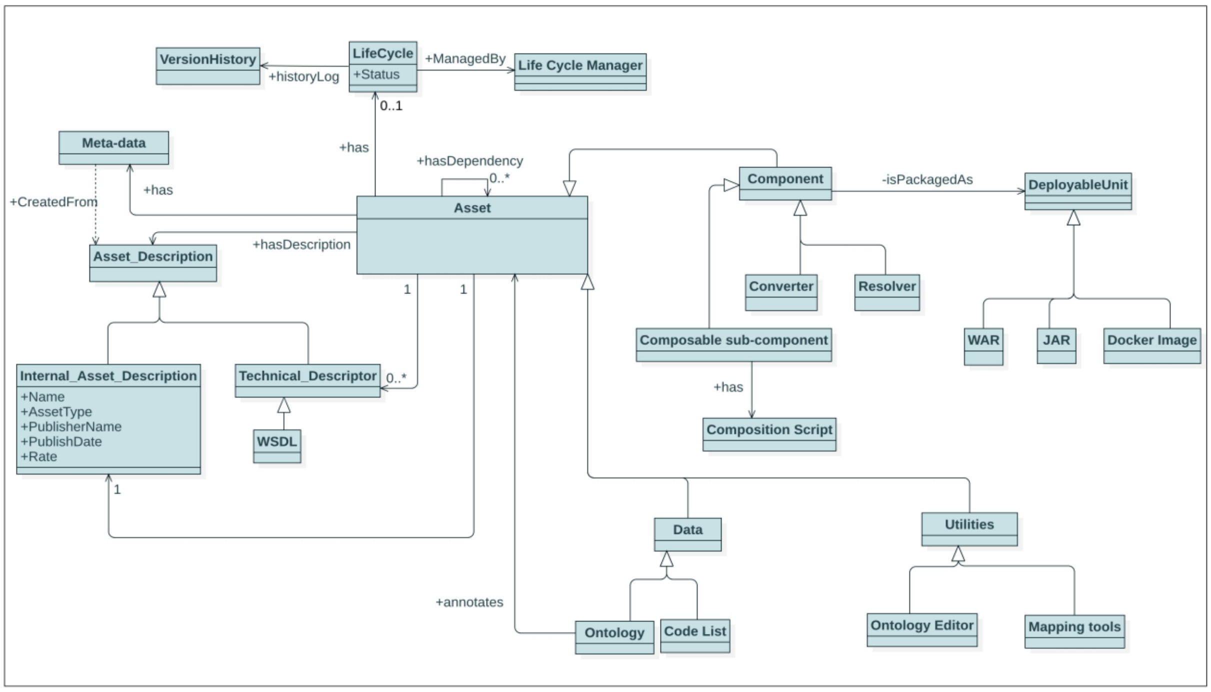
The Asset Manager is a pivotal component of the S2R IF. It offers the basic functionality to publish, share, discover, maintain and manage various artefacts that may be published/utilized by external and internal components of the IF. It acts as a catalogue of **Assets** which are subject to specific publication processes.

With respect to the principles described in Section 3.1, we can notice that the decomposition strategy (see Section 3.1.1) of the Asset Manager follows the *Cross-Asset Decomposition* pattern; indeed, as explained later in this section, each subcomponent of the Asset Manager handles a separate concern (e.g., lifecycle management, registration) for all types of assets. Different subcomponents, in turn, realize specific patterns, as it will be shown below. In addition, the Asset Manager acts as a registry of assets, and allows clients to discover services and use them. As a consequence, it realizes the *Service Registry* and the *Client-side Discovery* patterns of Section 3.1.3.

Asset

In the scope of the IF, any resource that a generic actor of the transportation domain may be interested in – to read, share and utilize it – has been referred to an Asset. More specifically, as represented in

Figure 9, an asset is an artifact that has some descriptions, a definable lifecycle, and it is



discoverable by Users and other Assets.

Figure 9. Asset Domain Model

An Asset Description is a central element in IF since it determines the discovery domain. In other words, only the aspects which have been stipulated in a description may be later searched by any interested user. An asset description hence defines the characteristics of an asset. However, the set of characteristics of an asset which is of interest of a particular – logical or actual - user may differ with other. Accordingly, we have anticipated two types of descriptions:

- **Internal Asset Description**
  A description which defines generic characteristics of an Asset which are mainly important for the internal usages by the Asset Manager itself, as well as for the end user to find out human understandable details of that asset. For instance, the publication date, the publisher, and the rate of the asset that shows its popularity.
  Any asset in the IF must have one and only one Internal Asset Description, which would be generated by the Asset Manager during the asset registration time. Some of the information inside the description may be obtained by the publisher itself, and others may be determined by AM.

- **Technical Description**
  It provides further technical details about the asset, which may be necessary to use/deploy/interact with the asset. For instance, the WSDL description of an asset that has been packaged as a service is required by the user to understand the interfaces of the service and how to bind with them. Differently from the "Internal Asset Description", it is not compulsory for an asset to have a technical description.

In addition, a machine-readable (RDF) description of the Asset, called metadata, may be generated for any asset based on these asset descriptions, in order to facilitate semantic based discovery of assets using distributed SPARQL endpoints. The AM stores the asset metadata in RDF according to the DCAT-AP [4] and ADMS [5] vocabularies.

Furthermore,

Figure 9 depicts three categories of Assets (Data, Utilities and Components) along with their subcategories. For each type of asset, we have also identified several actual instances as their subcategory. The subcategories represented in the figure for each type of asset are not exclusive in we might extend them in future deliverables.

Data Asset includes any types of data, and it is a materialization of the **Data Abstraction**. **Utilities** and **Components** are the realization of the **Service Abstraction**. In other words, Utility Assets (e.g., Ontology Editor, Mapping IDE) and Component Assets (e.g., Converter) are tools and services to enhance interoperability.

It is important to highlight that such assets may be provided and utilized by external actors as well as by the IF itself. In addition, as represented in

Figure 9, a Component Asset may be packaged as different deployable units. It enables multiple deployment and engagement choices for IF's clients and widens the usability of the IF. Finally, the type of an Asset determines its lifecycle and the specification of its description.

Finally, as depicted in

Figure 9, IF has anticipated various packaging and deployment strategies for engagement with any type of assets. Indeed, with respect to the deployment patterns descried in Section

3.1.4, the proposed IF architecture supports various solutions, to achieve maximum flexibility. Such decision has been made based on our requirement analysis reported in previous deliverables with the goal of addressing the requirements for various application domains and covering a wider range of use case scenarios. To this end, consumers/provides of IF can choose between three options for utilizing/offering various interoperability components as follows:

- **Direct Access**.
  It is a standard approach for facilitating a loosely coupled and service-oriented interoperability among transportation actors. A publisher can advertise an already running component in IF by providing a generic description of IF which includes its endpoint. The Asset Manager then creates MetaData out of its description so that it may be discoverable through distributed SPARQL endpoints. In such cases, the IF is mainly a service repository that bridges service descriptions into semantic web and fosters a wider discovery range. After the discovery phase, the role of the IF is terminated, and the customer would be redirected to the provider system where it has engaged with the desired service. This solution is an instance of the *Service Instance* or *Multiple Service Instances per Host* pattern of Section 3.1.4.

- **Runtime executable environment**.
  It is as an extension of the previous model that promotes a Platform-as-a-Service approach. It broadens the features of the Asset Manager to more than a catalogue manager. The AM becomes a command and control tool that actively manages deployable artifacts onto a cloud platform. To this end, either the service provider itself or any other external infrastructure provider should authorize the IF to utilize such a cloud environment upon user request. Accordingly, after the discovery phase, AM takes the appropriate executable artefact of the selected component, runs it on the expected cloud environment and returns the endpoint to access the service by the client. This approach realizes the *Service Deployment Platform* pattern of Section 3.1.4.

- **Direct Download.**
  Finally, to cover the use cases where clients prefer to run the desired component locally or integrate it as an internal part of their system the IF supports the Direct Download approach, where the publisher can upload the component implementation and the IF lets users download it. More precisely, a component of the IF may be wrapped up and published through different downloadable and runnable artefacts including a container image, JAR and WAR. A component may be packaged in multiple forms, what gives the client the possibility of choosing the most suitable method to engage with the desired component. For example, if the client’s internal system follows a micro-service architecture which is most often realized by utilizing technologies such as Docker [6], then a docker image of a component (e.g., Converter) that could readily be deployed on a Kubernetes [7] cluster and integrated with rest of the system seems the best option for the user. In other cases, if the client

prefers a conventional service-oriented approach, a component packaged as a WAR file that may be integrated. This third solution is an example of the *Service Instance for Container* pattern of Section 3.1.4, although the management of the container, once it is downloaded, is left to the client.

The previous discussion highlights the choice made in the design of the IF to leave to each published service the decision concerning the mechanisms to be used to interact with it. In principle, different choices regarding the communication patterns (*Remote Procedure Invocation*, *Messaging*, even *Domain-specific protocol*, see Section 3.1.3) could be made by different services. The Asset manager itself, instead, favors a *Remote Procedure Invocation* approach, as highlighted by its exposing a frontend (the UI component) to interact with it.

AM subcomponents

AM is composed of two subcomponents, **Store** and **Publisher**. The former is the end user interface, and the point of interaction of the end user with the IF (the IF design does not mandate a specific approach to build the user interface, so any of the *Server-side Page Fragment Composition* or the *Client-side UI Composition* can be used in its implementation). Through that, user can view and search through available assets. The main back-end service which enables such search functionality is the **Exploration API**.

- **Exploration API**

  With the aim of easing access to a potentially high number of asset metadata descriptions via common Web technologies, the Exploration API asset type (of Component Category) has also been defined. This asset type is used to describe parametric SPARQL queries, which are then exposed by the Asset Manager as Web APIs with a mechanism akin to grlc [8] and basil [9]. Using a normal HTTP GET request, the user can provide values to the parameters of the SPARQL query and obtain the results. Publishing parametric queries as assets in the catalogue allows a much higher level of control over the users' behaviors, since users are only allowed to call specific Web APIs according to their security permissions. The introduction of the Exploration API is an instance of the *API Gateway* pattern described in Section 3.1.3.

On the other hand, the main role of the **Assets Publisher** is to provide required interfaces to contribute an asset to the catalogue through the Asset Registration. Notice that the IF architecture does not currently restrict who can or cannot register assets through the Publisher component; as a consequence, this solution can support both the *Self-registration* and the *3rd party registration* patterns of Section 3.1.3. The other subcomponents include lifecycle management and distributed SPARQL endpoint.

- **Lifecycle Management**

Assets to be used in a wide ecosystem need to be managed in a consistent way to foster trust among the parties. To that extent, each asset type in the Assets Publisher is linked to a lifecycle process, which covers all the aspects from the initial submission to the final publication, plus all the aspects related to change management and the effects of a change on dependent assets. Since such lifecycle processes (expressed using the BPMN 2.0 standard) define roles and responsibilities in a distributed environment, they become the real “contract” between the parties participating in the ecosystem.

- **Distributed SPARQL endpoint**

  A distributed SPARQL endpoint is a query engine capable of processing queries over heterogeneous data. When a distributed SPARQL endpoint receives a SPARQL query, it first identifies which data sources can be used to answer the query using mapping information stored in the metadata repository. Previously, these data sources have been described in terms of mappings to a common domain (ontology). According to Figure 10, there is a mapping process that integrates several heterogenous data sources (XML, CSV, JSON and relational databases) through a virtual global RDF or virtual knowledge graph; the information about the mappings between a virtual global RDF and its data sources are gathered in a metadata repository. It is noteworthy that the architecture shown in Figure 10 is based on the OBDI (Ontology Based Data Integration) concept, i.e., a three-level architecture constituted by a virtual knowledge graph, multiple heterogenous data sources and the mapping between them. In this architecture, the user formulates SPARQL queries over the virtual knowledge graph, which is transformed into underlying query languages of the data sources.

  To transform a SPARQL query, several sub-queries are generated to be evaluated over each data source and also a query plan is created with the order in which will be executed these sub-queries. Then, the sub-queries are rewritten to other queries considering potential inferences from the ontology and information in the mapping. Afterward, each sub-query is translated to its correspondent source query engine language to be finally executed by the underlying data sources. Lastly, results obtained for each subquery are translated to RDF (or as SPARQL bindings) using the rules provided in the mappings and are aggregated, including the removal of duplicates and the linking of resources.

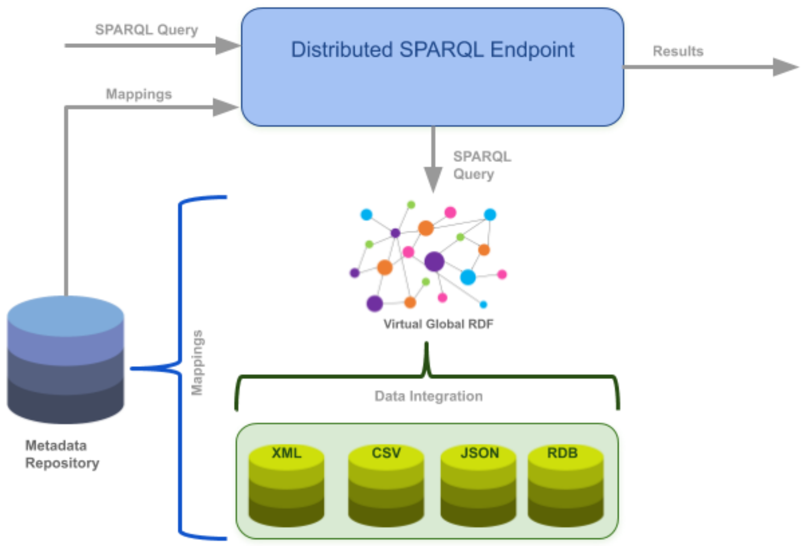


Figure 10. Distributed SPARQL Endpoint Architecture

The Distributed SPARQL Endpoint component highlights the decision made in the design of the IF to support a *Database per Service* pattern (see Section 3.1.2). Indeed, the component allows to retrieve data from different endpoints, without even requiring the materialisation of the information. In addition, by delegating the handling of queries to a suitable, separate component, the IF architecture can support different data querying and data consistency patterns. Finally, within that IF, the Distributed SPARQL Endpoint component is responsible, together with Converter assets, to handle data heterogeneity issues.

4. DESIGN OF THE S2R IF TESTING INFRASTRUCTURE

The Interoperability Framework, as designed in D3.2, is a complex system whose components belongs to two different categories. We can name those two categories as the "Interoperability design support" and the "Interoperability execution". The Asset Manager belongs to the former category, as it will be used as a catalogue (or "yellow pages") to discover the available artifacts and data models when designing an interoperability solution to join the Shift2Rail IP4 ecosystem. In Section 4.3 we discuss details about performance and scalability for the test cases of the Asset Manager. Additionally, Converters and Resolvers belong to the latter category, as they are the components which are called each time a message must be converted and each time aggregated data must be accessed to help achieve interoperability.

The C-Rel testing infrastructure will focus on the "Interoperability execution" category, since the performance and scalability of Converters and Resolvers have the greatest impact on achieving an effective interoperability. Converters and Resolvers differ in their purpose, but they anyway share data access as the fundamental part which affects performance and scalability.

Converters have a well-defined process, which have been initially explored by the ST4RT project, where incoming data is "lifted" to RDF according to a specific set of ontologies, and then "lowered" from RDF to the specific output specifications and format. Measuring performance and scalability for this class of components means measuring how such aspects are tackled by the conversion algorithm and by the underlying data access layer.

Resolvers are components designed and developed to fulfill task-specific purposes. They generally provide lookup functionalities, and therefore they should access data previously collected and merged in an RDF graph according to a specific set of ontologies (Shift2Rail ontology and previously IT2Rail ontology). Performance and scalability for this class of IF components are strictly tied to their specific purposes, so no general benchmark can be designed. We can anyway focus on the impact of their data access phase, since performance and scalability of a Resolver should largely depend on it.

As already discussed in the previous sections, one the main objectives of the S2R IF is to provide a unified view and access to heterogeneous data sources used in the transportation domain. However, these data sources are normally been published or exchanged in non-RDF formats, such as CSV, JSON and XML. Some examples of typical messages that may be exchanged in this context are the following:

- GTFS timetables
- NeTEx datasets containing timetables, routes and network descriptions
- TAP/TSI code lists

In this context, two different approaches can be used to semantically tackle data heterogeneity considering a common ontology (from the S2R ontology network) for accessing and exchanging data:

- Integrating all the source data in an RDF representation following the common ontology and then querying it (***materialized*** *graph*);
- Using query translation to access information directly in the original data sources (***virtualized*** *graphs*) and then providing results using the common ontology.

The proposed testing activities analyses the duality between the materialization and virtualization approaches considering the two critical aspects for performances and scalability of the IF and the related components, i.e., *querying* and *converting* heterogeneous data.

Considering *querying,* performance and scalability of the *materialized* approach are strictly tied to the ones of the RDF Triplestore used to store triples and run queries. Several studies already tested different triplestore implementations, results are widely documented in the literature [10] and many industrial actors already leverage this technology in their systems. On the other hand, performance and scalability of *querying* a virtualized graph over heterogeneous data sources should be investigated furthermore [11]. For this reason, in Sections 4.1- 4.2, we will study the possibilities of generating a benchmark in order to evaluate different approaches and tools exploiting a virtualization approach. This assessment would be useful to determine which state-of-the-art tools perform better and their readiness to be integrated into a production-ready solution. Moreover, tools considered in these tests should deal with distributed input data sources, therefore, tests performed also aims at testing tools to implement the distributed SPARQL endpoint functionality within the IF. In our testing infrastructure we will consider this context of large heterogeneity, which is expected in the context of the S2R IF, and we will develop a testbed that can be used to evaluate the performance and scalability of this type of solutions that will be applicable in the implementation of C-REL. This is what will be presented in Section 4.4.

Considering the *conversion*, materialization and virtualization are two different feasible approaches of implementing the lifting portion of the conversion, i.e., to obtain the set of triples input of the lowering procedure. In this context, performance and scalability assessment should consider different alternatives for lifting and lowering, and the overall conversion as implemented by the Converter. As a result, in Section 4.5 we discuss how to leverage on the same testbed to describe different testing scenario for this component.

4.1 Testbed For Scalability and Performance

In the state of the art, several benchmarks have been proposed in the area of OBDA and federation of SPARQL queries through different SPARQL endpoints [12] [13] [14] [15]. However, none of the existing benchmarks contemplates virtual knowledge graph access with a unified global view and data sources in different formats while providing a real-world setting, i.e., they are not OBDI benchmarks.

The testbed presented here was designed for virtual knowledge graph access. Thus, by using a benchmark setup for multiple formats, the performance and scalability of OBDA and OBDI engines can be compared, and the strengths and weaknesses of engines in both OBDA and OBDI scenarios can also be evaluated.

In the design of our testbed we have decided to work with data sources that are not particularly attached to the S2R domain, but share many of its characteristics, especially with the objective that the testbed provides a neutral view over the whole transport domain (so that other researchers can also make use of it more easily to take decisions on the systems to use for a particular task) while providing useful insights for the core technology components that may be part of the implementation of the aforementioned components of the S2R IF.

Our testbed is published in a Github repository[7] and should be adapted in the future to other S2R domains. It measures two aspects:

(1) Performance. It is a set of quantitative criteria whose values are obtained by running the testbed on the data from heterogeneous data sources of travel services in the transport ecosystem. For performance, scores are assigned according to the results of the execution time and then, a weighted average of the obtained score is also computed.

(2) Scalability. It measures the trend of performance with increasing data volume.

Table 1-Table 2 show the criteria that can be used in general to measure the performance and scalability of an RDF Data Storage and the converter. It is worth mentioning that some of these criteria are only used to measure Performance and Scalability on OBDA solutions or data integration platforms such as rewriting time of the query, query translation time and Query translation time. Source/endpoints selection time only applies to the query federation tools (OBDI). In accordance with the testbed objectives, weights can be assigned to each of these criteria, with a score in a range, for example, a score from 1 (less relevant) to 3 (more relevant), or a relative proportion of the weights between the different criteria, between 0 and 1, where the sum of the weights of all the criteria must be 1 in order to obtain a ranking of the RDF Data Storage in terms of performance and Scalability.

[7] https://github.com/oeg-upm/gtfs-bench

Metric	Description	Unit
Load time	Resource loading time during the starting phase when ontology, mappings and query are loaded	Ms
Total execution query time	Average execution time of queries	Ms
Total execution time per query type	Average execution time by type of queries	Ms
Query rewriting time	Average time in which the system rewrites the query, i.e. it expands the original query to a set of queries, taking into account potential inferences from the ontology and information in the mapping	Ms
Query translation time	Average translation time, taking the mapping into account, from the original query to another query that is expressed in the supported language by the underlying data sources.	Ms
Translation time of results	Average translation time of results where the results of the queries to the original data sources are translated into results expressed in the language of the original query (e.g., results in SPARQL)	Ms
Source/endpoints selection time	Average time for selection of data sources required to execute queries.	Ms
Query generation time	when the set of subqueries to be evaluated over each data source is created, and the query plan is generated	Ms
Mapping translation time	Time required by the engine to translate a provided mapping into another one in in a different language, maintaining a set of properties between them	Ms
Query execution time	Time when the translated queries are evaluated against the underlying data sources and the results are translated to RDF or as SPARQL bindings using the rules provided in the mappings	Ms
Query aggregation time	when the results obtained for each sub-query are aggregated, including the removal of duplicates and the linking of resources	Ms
Throughput	Number of tasks executed per unit of time	Queries/sec

Required data space	Required space for data including indices and any other secondary data access structure	MB
Successful queries	Percentage of queries that ended successfully	%
Correct queries	Percentage of queries with a set of correct results w.r.t. a Golden Truth, i.e., a reference standard	%
Complete queries	Percentage of queries with a set of complete results w.r.t. a Golden Truth	%

Table 1 Metrics for Performance and Scalability for RDF Data Storages

Metric	Description	Unit
Converter artifact generation time	Time required to generate the Converter downloadable artifact.	Ms
Response Time for conversion	The time required to convert a data set or a message.	Ms
Throughput for runtime data/message converter	Number of conversion requests that can be handled per unit of time	Requests/sec
Lifting time	Time required to convert data from the input format to their ontological representation.	Ms
Lowering time	Time required to convert data from their ontological representation to the output format.	Ms

Table 2 Metrics for Performance and Scalability for Converter

As in any other similar testbed, we will describe the datasets and mappings used in the testbed and the queries used to analyse the system behaviour. Finally, in section 4.2 we will propose the execution environment where the tests will be executed.

4.1.1 Dataset

Datasets can be generated with different sizes allow us to study the performance and scalability for several IF components in the use of resources such as storage space and execution time. Before generating datasets of different sizes, the original dataset must be anonymized, i.e., the original dataset must be replaced with a version whose identifiers are fictitious so that they cannot be to associate the concept that owns them.

The original dataset instance can be scaled using VIG [16] as a tool to generate datasets with different scale values while taking account the domain information of ontologies and mappings. VIG receives as input a dataset D together with its schema E, a set of mappings M (together with the ontology) and the required scale S, and produces as a result a dataset D' with a size S times its original size. The transformation from D' to RDF can be performed using OBDA tools such as Morph-RDB[8] or Ontop[9]. It is worth mentioning that VIG also has options to anonymize the data.

In an OBDI/OBDA context, query engines translate their results into RDF. Thus, in order to evaluate the correctness and completeness of the queries, the knowledge graph can be materialised using a tool such as SDM-RDFizer[10], which generates a materialized RDF graph taking as input RML mapping rules.

To run our testbed, we have decided to use data about public transport in the city of Madrid. More specifically, the web portal of Madrid Regional Transportation Consortium has published information about public transport of Madrid in order to users and not-for-profit enterprises can find and reuse these data. With this in mind, a benchmark for virtual knowledge graph access in the transport domain following the General Transit Feed Specification (GTFS) has been defined by us. The result of this work is also published at [11].

The first step in a benchmark design is the generation of relevant datasets of different sizes. For an IF Reference Dataset based on real data from the transport ecosystem, it must be conveniently anonymized if necessary, as well as scaled to different sizes in order to stress the IF components to be tested.

4.1.2 Queries

A set of queries must be specified to run our testbed. The queries will be expressed according to:

a. Structure. It refers to structural characteristics related to the queries such as the number of triple patterns, join shapes (star, chain or mixed) and number of OPTIONAL clauses. A star-shaped join contains a group of triple patterns that are “joined" over the same subject or object variable. In a chain-shaped join, triple patterns are consecutively connected like a chain. Mixed queries combine star-shaped and chain-shaped joins.

   The structure properties of a query impacts on overall execution time and specifically they also impact on query generation, query rewriting, query translation, and query execution times. Moreover, query structure may affect query plans built during the subquery generation phase. On one hand, in an OBDA context, the query plans are generated by the underlying engine and therefore, performance and scalability will be impacted by the query structure. On the other hand, in an OBDI context,

[8] https://github.com/oeg-upm/morph-rdb

[9] https://github.com/ontop/ontop

[10] https://github.com/SDM-TIB/SDM-RDFizer

performance and scalability will be affected by the plan generated by the OBDI engine.

b. Expressivity. It refers to the use of different options provided by the query language such as arithmetic functions, aggregation, and filters. For instance, the use of language features like DISTINCT and ORDER BY may impact on the query execution time metric because they require an ordering of the tuples/entries of the underlying sources.

c. Heterogeneity. Some queries must be executed against one data source and others must be evaluated over a set of homogeneous sources (same data model) or heterogeneous sources.

d. Selectivity. It is a percentage of the accessed triple patterns by a SPARQL query. Constants in triple patterns together with FILTER with equality operators usually produce high selectivity of queries and are likely to reduce the cost of evaluating the query. However, using a FILTER relational operator specially in the case of open ranges, e.g. a FILTER with a > operator, may generate a large number of results.

e. General Predicates. It corresponds to the use of language properties such as rdf:type or owl:sameAs.

4.1.3 Mappings

Ontology-based data access (OBDA) refers to a variety of techniques, algorithms, and systems that can be used to address the problem of heterogenous data integration that is common within many organizations. In OBDA, ontologies are used to provide a global view of multiple local datasets and mappings are commonly used to describe the relationships between these global and local schemes. In this approach, queries written according to the global scheme are transformed into the query language supported by the original data sources, evaluated in the original data management systems, and the results are transformed back into the global view.

Mappings play a fundamental role in the testbed since they are the main elements used during the query translation. In the state of the art, many different types of OBDA mapping languages have been proposed with a great variety of syntaxes and formats: RML [2], R2RML [17], xR2RML [18], and Ontop [19] OBDA mappings.

The mappings represent the relation of one resource in the ontology with the corresponding data source. The features of the mappings are very relevant because they may impact on the performance of the virtual knowledge graph access engines. In general, each mapping in a mapping file is characterized by the related data source, number of classes, number of Predicate Object Maps, number of Predicates, number of Objects and number of joins (e.g. RefObjectMaps).

4.2 Technical Infrastructure for Executing the Experiments

This section describes a potential execution environment that can be used to evaluate and validate the performance and scalability of any new instance of the S2R IF. Based on the datasets from the travel services provided by different transportation sectors, automatically-created mappings and a set of queries, the behaviour of IF components can be experimentally evaluated.

Since the Data Layer relies on the back-end databases and management of database operations needed to handle collection, storage, and retrieval of data, it is necessary to include RDF stores keeping RDF graphs such as ontologies, enriched meta-data (according to the reference ontology) and meta-data generated by the Asset Manager describing different assets.

Several state-of-art benchmarks have been developed in order to evaluate the efficiency of different tools capable of executing SPARQL queries on RDF datasets, e.g., SP^{2}Bench [20], LUBM [21], BSBM [12], DBSPB [22], and NPD. They are usually composed of a set of queries that meet various features (e.g. operators or number of joins), a scale data generator and a set of measures such as total execution.

The steps to be followed for the execution of a Performance and Scalability Benchmark must consider that some triple stores are queried by means of a materialized RDF graph generated from data sources, while other triple stores are consulted over virtualized data from data sources using OBDA. The setup steps are as follows:

1. Generation of datasets with different sizes. Different sizes of datasets allow to show the scalability of any tool used for querying data with respect to the use of resources, the storage space and loading and execution time. For example, datasets can be generated with increments of 1 order of magnitude, 10, and 100 times larger.

2. Specification of mappings. A set of mappings between datasets and the reference knowledge graph must be specified to transform heterogeneous data sources into a common data model, which can then be accessed and processed in order to complete the querying tasks.

3. Transformation and loading of data. In this step, the data resulting from the previous step can be transformed to a supported format. This step is relevant only for triple stores incapable of directly working with a given format, for example, NoSQL and OBDA that don't implement query translation.

4. Query Generation. In this step, a set of queries are specified according to structure, expressivity, heterogeneity, selectivity and general predicates.

5. Tool Deployment in controlled environments. Some variables related to the platform can affect the measurements that must be controlled during the Benchmark execution. These are:

a. Cache on/off. It can be relevant if a query is executed with Warm or Cold Cache.

b. Available RAM memory.

c. Number of processors.

d. Network latency in terms of the delay that occurs to send packets through the network.

e. Initial delay of the endpoint/node.

f. Message size.

g. Number of endpoints/nodes.

h. Type of endpoint/node.

i. Distribution for packet transmission time by endpoints/nodes.

j. Result size limit.

k. Timeout.

6. Selection of Metrics. Depending on the IF component, a set of metrics must be selected to measure and analyze the behavior of such IF components. These metrics are defined in Table 1-Table 2.

7. Query Execution. A set of SPARQL queries will be evaluated in warm and in cold mode in order to analyze how the cache mechanism may affect the performance of the tools. Each query is run a certain number of times. In warm mode, each query will be evaluated discarding its first run and then it was run again a certain number of times to compute the average query execution time. In cold mode, the database server was restarted after each run to clean all the caches.

8. Result Analysis. The results of each tool are independently analyzed and discussed.

9. Data review. The configurations of the variables related to the dataset size are reviewed, the benchmark is run again if it is necessary and the final results are published.

All experiments will be performed using Docker containers to ensure reproducibility. For each tool, a docker image will be created considering the recommended settings and the datasets generated with different sizes will be loaded.

4.3 Testing Asset Manager Performance and Scalability

The Asset Manager is a metadata catalogue which is able to enforce different complex publication processes. It can be either used internally by a company or shared between different companies willing to establish a common data and service ecosystem. Given its purpose and its intended usage, we can analyze what the terms “performances” and “scalability” mean in this context.

The main purpose of the Asset Manager is to act as a metadata catalogue. Each asset therefore is described using metadata, and can optionally have “attachments” in case the described asset cannot be publicly accessed. Metadata is saved as RDF in a repository, while attachments are saved as files in a separate object storage.

The metadata schema which will be used by the Asset Manager is yet to be decided, anyway the amount of triples required to archive metadata information about an asset is quite low. Using DCAT-AP, metadata about a single asset can require about 20 RDF triples. One of the basic principles of the Interoperability Framework is to avoid centralization, therefore the Asset Manager is not intended to be deployed as a single instance to serve the users across all Europe. A likely scenario is that multiple instances of the Asset Manager would be deployed by different groups of competing Transport Operators, mimicking the situation in the air transport dominated by different “alliances” between airline operators. In this likely scenario, the assets managed by a single instance of the Asset Manager could be in the thousands. Off-the-shelf RDF repositories can manage billions of triples[11], therefore can be used to store metadata about hundreds of millions of assets. Scalability requirements for the RDF metadata layer of the Asset Manager can then be considered as satisfied without further investigation.

The same principles apply when querying a single RDF repository using SPARQL. Even by using DCAT-AP, the metadata schema does not allow for very complex SPARQL queries. Moreover, since those queries will be quite similar, they will be likely cached by the RDF repositories. As the SPARQL language is a standard for all RDF repositories, any KPI about performance requirements related to queries over metadata will just be related to the specific RDF repository which will be chosen for the implementation [10].

Performance is the most important aspect to be checked for the metadata layer. In this case, testing a distributed scenario is relevant when we can consider that the metadata layer will be implemented by a distributed SPARQL endpoint which federates many data sources. This can be the case of a transport operator owning different RDF repositories, each one containing metadata information. In this case, measuring query time with a growing number of endpoints and a growing size of metadata information can prove the possibility for the Asset Manager to be adopted in large and distributed companies. Since the part of the Asset Manager implementing such feature is the same Distributed SPARQL endpoint whose tests

[11] https://www.w3.org/wiki/LargeTripleStores

are defined in Section 4.4, we will refer to the tests defined for such component to evaluate the performances and scalability of the Asset Manager data access layer when dealing with distributed metadata.

Scalability means also the possibility to manage a growing number of users. Being the Asset Manager a metadata catalogue, such users can grow in numbers, but anyway their number will be limited and will not be comparable to the number of users of a social network. The Asset Manager users are expected to be developers looking for suitable assets, and asset publishers (therefore mostly transport operators and ITS providers). Most likely, the act of discovering new assets and publishing asset descriptions will not be performed many times per day by a single user. The C-Rel version of the Asset Manager will leverage containerization and will enable deployment on a single machine. F-Rel version of the Asset Manager will focus on its deployment on a cloud environment to leverage automatic scalability. Therefore, we will focus on testing performance and scalability of the Asset Manager when dealing with a large user base in F-Rel.

4.4 Testing Distributed SPARQL Endpoint Performance and Scalability

Since the function of the Distributed SPARQL endpoint is to answer distributed queries on heterogeneous data sources, our test case is based on testing the Distributed SPARQL endpoint with a variety of SPARQL queries of diverse complexity and on heterogeneous data sources that store data in various formats (heterogeneity). In addition, although our Distributed service/asset discovery scenario (Basic Scenario 2 in [3]) is about a federated query on metadata catalogues, we have decided to build our test case on data instead of metadata mainly because the volume of metadata is usually much smaller than the data and we can draw more significant conclusions in front of a larger volume of data. For example, we estimate 20 RDF triples for an asset stored in a metadata repository.

Particularly, following the steps proposed in section 4.2, we have tested federated queries on the data of the Madrid metro system which have been scaled 5, 10, 50, 100 and 500 times its size to measure scalability (Step 1). With respect to heterogeneity, the formats of the datasets will be the most commonly used in the data exchange: CSV, XML, RDB and JSON. For the Distributed SPARQL endpoint to be able to answer a SPARQL federated query, it requires a set of mappings that will be specified in RML [2], R2RML [17], xR2RML [18], and Ontop OBDA mappings [19] (Step 2) and a set of SPARQL queries varying the number of triple patterns, the number of data sources selected, the use of OPTIONAL clause or aggregation functions, equality (equal to) or range (relational) conditions in the FILTER clause, and the number of constants in a query (Step 4). Subsequently, those platform-related variables that can affect the performance measurements are controlled (Step 5) and the metrics from Table 1 are selected to measure performance and scalability of the Distributed SPARQL endpoint (Step 6). Then, the set of SPARQL queries generated during the Step 4 are executed for each CSV, XML, RDB and JSON format to measure query performance and they are also executed for each dataset size on a scale of 10, 50, 100 and

500 to measure scalability of the Distributed SPARQL endpoint (Step 7). Finally, the results are analysed in Step 8.

4.5 Testing Converter Performance and Scalability

The conversion process implemented by the IF Converter is composed by two main phases: *(i)* the *lifting* phase accesses the data in the input data format and generates the corresponding RDF representation adopting the reference ontology, and *(ii)* the *lowering* phase queries data represented using the reference ontology and generates converted data in the output data format. Both lifting and lowering can be implemented using different techniques and approaches. A first distinction in the possible techniques lies between annotation-based and declarative approaches. The ST4RT project demonstrated the feasibility of an annotation-based approach for converting messages, whereas declarative approaches like RML have their roots in batch conversion of bigger datasets. Another distinction is between materialization and virtualization. Lifting input data can be implemented by saving the corresponding RDF triples in an RDF database and then queried during lowering, therefore using a materialization approach. The virtualization approach instead provides results to the queries used for lowering by translating them on the fly into the corresponding and format-specific queries. Annotations, declarative languages, materialization and virtualization are therefore possible ways to achieve the same results. All of them provide advantages in some cases and are affected by weaknesses in other cases. Designing a performance and scalability test for the Converter can therefore help in understanding the limitations of each approach.

Since the ST4RT project already provided an evaluation of the annotation-based approach, in C-Rel we will focus on exploring the other technological possibilities to build efficient Converters: RML-based materialization and virtualization.

Materialization and virtualization have different advantages and disadvantages, different tools implement these approaches and can be combined to obtain a working Converter. We propose two test cases designed to test performance and scalability of the conversion implementation considering the two main scenarios described in deliverable [3]. The purpose of the designed test cases is to understand which typology of approach perform better in the two scenarios, to identify limits of the tools chosen and to overcome potential bottlenecks in the reference implementation developed within the project.

The steps described in section 4.2 can be adapted to test Performance and Scalability of the Converter:

1. First, we must generate different sizes of datasets to show the scalability of the specific tool used for conversion of data with respect to the metrics to be measured.
2. Second, a set of mappings between datasets and the reference knowledge graph must be specified to accomplish the lifting stage during the converter pipeline. This knowledge graph can be virtualized or materialised.

3. As queries are executed only during the lowering stage, we must create template lowerers which include a query to extract RDF data from (virtual or materialised) knowledge graph and template variables to bind RDF to their output.

4. In addition, a set of metrics must be selected to measure and analyze the behavior of the Converter. These metrics are defined in Table 2.

5. Subsequently, the experimental evaluation of the behaviour of the converter will be executed. For the conversion task, lifting and lowering stages must consider if the RDF graph is materialised or virtualised. During the lifting stage, data are transformed to an ontological representation which can be materialised or virtualised for later being queried during the lowering stage in order to extract and transform RDF data into the desired output format. Each complete conversion is run a certain number of times to calculate the average conversion execution time.

6. Finally, the results are analyzed.

The proposed approach can be employed in the two considered scenarios. However, it is important to notice that the two cases should evaluate different typologies of scalability in the input datasets. The *Batch Data Conversion* scenario considers the case where a batch dataset should be converted. Usually, this scenario does not have constraints on the conversion time, but it requires scalability with respect to the size of the dataset that can be in the order of hundreds of megabytes. The *Runtime Data/Message Conversion* (Service Mediation) scenario considers the case where a message or a small amount of data (in the order of bytes/kilobytes) should be converted to guarantee communication between two different systems. Usually, this scenario involves small size datasets, but it requires conversion time to be as short as possible to introduce low overhead in the communication. In this case it is important to verify the scalability of the approach in the case of increasing concurrent requests.

5. REFERENCES

[1] C. Richardson, Microservices Patterns, Manning Publications, 2018.

[2] A. Dimou, M. V. Sande, P. Colpaert, E. Mannens and R. V. d. Walle, "Extending R2RML to a Source-independent Mapping Language for RDF," in *International Semantic Web Conference (Posters & Demos)*, 2013.

[3] SPRINT, "D3.2 - PERFORMANCE AND SCALABILITY REQUIREMENTS FOR THE IF," 2019. [Online]. Available: http://sprint-transport.eu/Page.aspx?CAT=DELIVERABLES&IdPage=1e2645be-e780-4d99-8117-bae57b67b453.

[4] S. Goedertier, *DCAT application profile for data portals in Europe,* 2013.

[5] M. Dekkers, "Asset description metadata schema (adms)," *W3C Working Group,* 2013.

[6] "Docker," [Online]. Available: https://www.docker.com/). .

[7] "Kubernetes," [Online]. Available: https://kubernetes.io.

[8] A. Meroño-Peñuela and R. Hoekstra, "grlc makes GitHub taste like linked data APIs," in *European Semantic Web Conference*, 2016.

[9] E. Daga, L. Panziera and C. Pedrinaci, "Basil: A cloud platform for sharing and reusing SPARQL queries as Web APIs," in *CEUR Workshop Proceedings*, 2015.

[10] M. Saleem, G. Szárnyas, F. Conrads, S. A. C. Bukhari, Q. Mehmood and A. C. Ngonga Ngomo, "How representative is a sparql benchmark? an analysis of rdf triplestore benchmarks," in *The World Wide Web Conference*, San Francisco,USA, 2019.

[11] D. Chaves-Fraga, F. Priyatna, A. Cimmino, J. Toledo, E. Ruckhaus and O. Corcho, "GTFS-Madrid-Bench: A Benchmark for Virtual Knowledge Graph Access in the Transport Domain".

[12] C. Bizer and A. Schultz, "The Berlin SPARQL benchmark," *Int. J. Semantic Web Inf. Syst ,* vol. 5, pp. 1-24, 2009.

[13] D. Lanti, M. Rezk, M. Slusnys, G. Xiao and D. Calvanese, "The NPD benchmark for OBDA systems," in *CEUR Workshop Proceedings*, 2014.

[14] M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte and T. Tran, "FedBench: A Benchmark Suite for Federated Semantic Data Query Processing," in *International Semantic Web Conference*, 2011.

[15] G. Montoya, M. Vidal, O. Corcho, E. Ruckhaus and C. Buil-Aranda, "Benchmarking federated SPARQL query engines: Are existing testbeds enough?," in *International Semantic Web Conference*, 2012.

[16] D. Lanti, G. Xiao and D. Calvanese, "VIG: Data scaling for OBDA benchmarks," *Semantic Web,* vol. 10, no. 2, pp. 413-433, 2019.

[17] S. Das, S. Sundara and R. Cygania, "R2RML: RDB to RDF Mapping Language. W3C Recommendation," September 2012. [Online]. Available: https://www.w3.org/TR/r2rml/.

[18] F. Michel, L. Djimenou, C. Faron-Zucker and J. Montagnat, "xR2RML: Relational and Non-Relational Databases to RDF Mapping Language," 2014.

[19] D. Calvanese, B. Cogrel, S. Komla-Ebri, R. Kontchakov, D. Lanti, M. Rezk, M. Rodriguez-Muro and G. Xiao, "Ontop: Answering SPARQL queries over relational databases," *Semantic Web,* vol. 8, no. 3, pp. 471-487, 2017.

[20] M. Schmidt, T. Hornung, M. Meier, C. Pinkel and G. Lausen, "SP2Bench: A SPARQL Performance Benchmark," in *International Conference on Data Engineering*, 2009.

[21] Y. Guo, Z. Pan and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems," *J. Web Semant,* vol. 3, pp. 158-182, 2005.

[22] M. Morsey, J. Lehmann, S. Auer and A. Ngomo, "DBpedia SPARQL Benchmark - Performance Assessment with Real Queries on Real Data," in *International Semantic Web Conference*, 2011.

[23] "GTFS Static Overview," [Online]. Available: https://developers.google.com/transit/gtfs/.

[24] A. Poggi, D. Lembo, D. Calvanese, G. D. Giacomo, M. Lenzerini and R. Rosati, "Linking data to ontologies," *Journal on data semantics X,* pp. 133-173, 2008.

[25] "General Transit Feed Specification," [Online]. Available: https://www.transitwiki.org/TransitWiki/index.php?title=General_Transit_Feed_Specification.

[26] S. Harris and A. Seaborne, "SPARQL 1.1 Query Language W3C Recommendation," 21 March 2013. [Online]. Available: https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.