Building the Legal Knowledge Graph for Smart Compliance Services in Multilingual Europe

D1.3 Technical architecture design

PROJECT ACRONYM	Lynx
PROJECT TITLE	Building the Legal Knowledge Graph for Smart Compliance Services in Multilingual Europe
GRANT AGREEMENT	H2020-780602
FUNDING SCHEME	ICT-14-2017 - Innovation Action (IA)
STARTING DATE (DURATION)	01/12/2017 (36 months)
PROJECT WEBSITE	http://lynx-project.eu
COORDINATOR	Elena Montiel-Ponsoda (UPM)
RESPONSIBLE AUTHORS	Filippo Maganza, Kennedy Junior Anagbo (ALP)
CONTRIBUTORS	Socorro Bernardos Galindo (UPM)
REVIEWERS	Julian Moreno-Schneider, Stefanie Hegele (DFKI), Ilan Kenerman (KD)
VERSION | STATUS	V1.0 | Final
NATURE	Report
DISSEMINATION LEVEL	Public
DOCUMENT DOI	https://zenodo.org/communities/lynx/10.5281/zenodo.2580245
DATE	28/02/2019

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 780602

VERSION	MODIFICATION(S)	DATE	AUTHOR(S)
0.1	First draft	09/11/2018	Filippo Maganza, Kennedy Junior Anagbo
0.2	Document structure	14/11/2018	Filippo Maganza, Kennedy Junior Anagbo
0.3	Added Introduction	20/11/2018	Filippo Maganza, Kennedy Junior Anagbo
0.5	Added Chapter 2	30/11/2018	Filippo Maganza, Kennedy Junior Anagbo
0.6	Added Chapter 3	15/12/2018	Filippo Maganza, Kennedy Junior Anagbo
0.8	Added Chapter 4	04/01/2019	Filippo Maganza, Kennedy Junior Anagbo
0.9	Added Chapter 5, 6 and Annex 1,2	15/01/2019	Filippo Maganza, Kennedy Junior Anagbo
1.0	Integration of reviewers' comments	16/02/2019	Filippo Maganza, Kennedy Junior Anagbo

ACRONYMS LIST

API	Application Programming Interface
BB	Building Block
CRUD	Create Read Update Delete
DAG	Directed Acyclic Graph
DevOps	Development Operations
ESB	Enterprise Service Bus
HMAC	Hash-based Message Authentication Code
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IaaS	Infrastructure as a Service
JSON	JavaScript Object Notation+
JWT	JSON Web Token
LKG	Legal Knowledge Graph
MSA	Microservices Architecture
NIF	NLP Interchange Format
NLP	Natural Language Processing
PaaS	Platform as a Service
PDF	Portable Document Format
QoS	Quality of Service
REST	Representational State Transfer
RSA	Rivest–Shamir–Adleman (Algorithm)
SaaS	Software as a Service
SHA256	Secure Hash Algorithm
SOA	Service-Oriented Architecture
TCP	Transport Control Protocol
TR	Technical Requirement
UI	User Interface
URL	Unique Resource Locator
XML	Extensible Markup Language

EXECUTIVE SUMMARY

The Lynx project as a compliance service-oriented platform that seeks to address compliance and regulatory related needs of its clients and users, comprises many services ranging from the fields of Information Extraction and Linking, Knowledge Management and Sematic Web; for which there is the need for a high level technical architecture design that integrates these services in order to address the objectives of the project.

This document, therefore, provides a detailed description of the technical architectural design solution for the Lynx platform. Architecture design in any software development project as in the case of Lynx serves as a blueprint that drives the smooth design and implementation of the software system. Once implemented, fundamental structural choices are costly to change.

In this deliverable, standard architectural patterns such as the Monolithic architecture pattern, SOA and MSA are described narrowing down on the choice of the architectural patterns that suites the architectural design needs of the Lynx platform with reference to the functional and technical requirements of the Lynx platform.

For the Lynx platform, we adopted the **MSA pattern** which supports loose coupling between services; a feature that allows for easy separation of work and which makes this pattern most adaptable to the Lynx platform requirements.

The document also provides detailed description of the logical and physical architectures.

The logical architecture describes the following:

- Access control system
- Microservices coordination system
- Communication system
- Foundational microservices
- How the software microservices are wired together to form a larger software system

The physical architecture describes the type of infrastructures, the platform and the deployment architecture of the software application and how to deliver the deployable system.

TABLE OF CONTENTS

TABLE OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Technical architecture design refers to the high-level structures of a system and the discipline of creating such structures and systems. **The IEEE standard 1471** [1] **defines technical architecture (software architecture) as the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution**.

Technical architecture design functions as a blueprint for the system and the project under development. Once implemented, fundamental structural choices are costly to change. The documentation of technical architecture design helps to facilitate communication among stakeholders, captures early decisions about the high-level design, allows reuse of the design components among projects and serves as a basis for work-breakdown structure.

Lynx is envisioned as a platform capable of assisting customers in searching between relevant legal compliance information documents as detailed in the "Pilots requirements analysis" of D4.1; this objective entails several technical challenges that must be considered in the platform architecture design as described in the technical requirement analysis report D1.2.

These technical requirements coupled with the functional requirements analysis report D1.1, and D4.1 will drive the design and implementation of the architecture of Lynx.

1.1 PURPOSE AND STRUCTURE OF THIS DOCUMENT

This document provides a detailed high-level description of the technical architecture of the Lynx platform. This solution aims to address all requirements described in D1.1, D1.2 and D4.1 of the Lynx project.

Descriptions of the major components that will make up the design solution, the dependencies between them, and how they will work together are also provided in this document. The document also captures all design considerations and architecture level technical design decisions.

This document serves as the central point of reference regarding the technical architecture design of the Lynx platform.

It is worth noting that, this is a living document and therefore the technical architects will update this document as the project progresses until the system is promoted into production.

The rest of this document is organized as follows:

Section 2 details out important assumptions in the architectural design of the Lynx platform.

Section 3 describes the possible architectural patterns considered for the architectural design of the Lynx platform and the choice of an adaptable architectural pattern for the platform.

Section 4 presents some adopted patterns and industry standards, describes the foundational microservices (structural components) of the platform and presents the logical architecture of Lynx.

Section 5 describes the physical architecture of the platform focusing on some software deployment technologies with the deployment architecture description closing the section.

Section 6 concludes the deliverable and presents future work.

2 ASSUMPTIONS ON THE DESIGN ARCHITECTURE

Design decisions are usually influenced by a lot of information from both the problem to be addressed and best practices from the field, often imposed as standards as well as the experience of the designers and implementers. All these data help to drive some preferences for the design and eventually steer the assumptions to consider in the design process.

The following assumptions were made relating to the design of the Lynx architecture:

- The technical requirements that demand for a flexible and modular architecture take precedence over the performance requirements.
- The functional requirements of the Lynx platform specified in D1.1 “Functional Requirements Analysis Report” and in D4.1 “Pilots Requirements Analysis Report” are complemented with the functional requirements described in the ANNEX of this document.
- The performance requirements specified in D1.2 “Technical requirements analysis report” may change in the future.

3 ARCHITECTURAL DESIGN PATTERNS

An architecture pattern expresses a generalised approach to the structural organization or schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them [2].

In this section, we present a general overview of the patterns considered for the Lynx platform; then we compare them by explaining their advantages and disadvantages and finally discuss the choice of the pattern for the Lynx platform.

3.1 MONOLITHIC ARCHITECTURAL DESIGN PATTERN

A monolithic application describes a software application in which all the functionally distinguishable aspects (e.g. data input and output, data processing, error handling, and the user interface) are interwoven into a single program from a single platform [3]. The application is packed and deployed as a monolith.

3.2 SERVICE-ORIENTED ARCHITECTURAL DESIGN PATTERN

The service-oriented architecture (SOA) is a style of software design where services are provided to the other components by application components through a communication protocol over a network [4].

An SOA application is built by assembling a collection of self-contained service components. Each service represents a business activity with a specified outcome and may consist of other underlying services.

3.3 MICROSERVICE ARCHITECTURAL DESIGN PATTERN

The microservice architecture (MSA) is a variant of SOA in which the application is structured as a **collection of loosely coupled services** which implement business capabilities.

Two important concepts of MSA are the “bounded context” and the Single Responsibility Principle. The former refers to the coupling of a service component and its data as a single unit with minimal dependencies while the latter explains that, “each software module should have one and only one reason to change” [5].

The diagram in Figure 1 shows an application constructed as a series of microservices where each microservice (e.g. Service B1 and Service B2) has a clear team owner (Team B), and each team has a clear, non-overlapping set of responsibilities; a unique distinguishing feature of the microservice architectural design pattern.

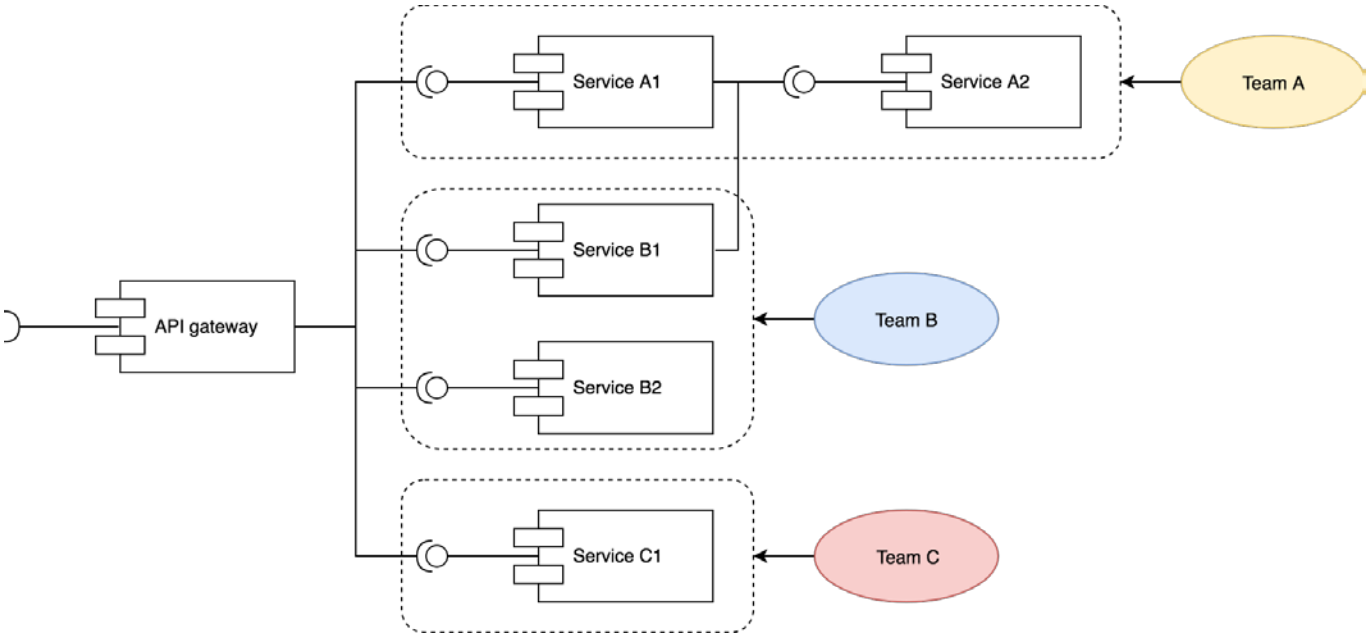

Figure 1 An application designed using MSA

3.4 COMPARISON OF MSA WITH TRADITIONAL SOA

Despite the reliance of both MSA and SOA on services as their main components, there exist some distinctive variations in their service characteristics and functioning. Therefore, in this section, we present some major differences between MSA and SOA as explained in Table 1.

Comparison context	SOA	MSA
Service intercommunication	Promotes the propagation of multiple heterogeneous protocols through its intercommunication middleware component called Enterprise Service Bus (ESB).	Promotes the use of simple and lightweight protocols like REST and SOAP using API gateway.
Service granularity	Services usually include much more business functionality and they are often implemented as complete subsystems.	Service components are generally single purpose and execute a specific task according to the Single Responsibility Principle.
Service reusability	Enhances component sharing. Thus, services and their functionality can be reused.	Minimize component sharing through “bounded context”.
Decoupling of services	The “share as much as possible” approach implies strong coupling of services.	Enhances decoupling of services through “bounded context”.
Data storage	Data storage is shared across multiple services.	Each service is provisioned to have a dedicated and an independent storage.

Table 1 Differences between MSA and SOA.

3.5 COMPARISON OF SERVICE-ORIENTED ARCHITECTURES WITH MONOLITHIC ARCHITECTURE

In this section, we give some significant advantages of service-based architectures (like SOA or MSA) over monolithic design structures:

- **Better testability:** small, independent services are easier to test, and debug as compared to massive chunks of code in monolithic architecture.
- **Service interoperability:** service-oriented architectures facilitate the development of a complex product by integrating different products from different vendors independent of the platform and technology.
- **Improved scalability**: multiple instances of a single service can run on different servers at the same time; this increases scalability because the whole application is divided into smaller units of single services that can scale independently. Furthermore, horizontally scaling monolithic applications can often be a challenge when there is an increasing data volume but much easier with service-based architectures.
- **Improved fault isolation**: larger applications can remain mostly unaffected by the failure of a single module in service-oriented architectures compared to monoliths where the entire application is affected.
- **Continuous deployment**: it is easier to achieve continuous deployment with service-oriented architectures since each service can be independently deployed without other services with its changes automatically propagated into production.
- **Increased flexibility**: difficulty in adopting new and advanced technologies with regards to monolithic design structures regardless of how easy the initial stages may seem compared to service-based architectures since changes in frameworks and tools affect an entire application.

Despite the numerous advantages of service-based architectures over a monolithic application, it has some drawbacks such as:

- **Overhead**: in service-oriented architectures, every time a service interacts with another service, a transmission of data over a network occurs with a complete validation of every input parameter. This increases the response time and machine load, and thereby reduces the overall performance.
- **Complex asynchronous communication**: there may exist a high amount of asynchronous calls between services.

3.6 ARCHITECTURAL PATTERN FOR THE LYNX PLATFORM

Considering the requirements of the Lynx project coupled with D1.1, D1.2 and D4.1 (in particular TR1, TR3, TR7, TR10 and TR28 of D1.2), the main architectural pattern adaptable to the technical architectural design of the Lynx platform is MSA.

Nevertheless, there is one Lynx requirement that does not comply well with the MSA pattern: the platform shall provide “a Legal Knowledge Graph for smart compliance services” (from the original proposal document with the topic “ICT-14-2016-2017”). This requirement calls for the reusing and sharing of data and metadata of the LKG across multiple services; a requirement adequately supported by the SOA pattern.

Consequently, **we shall adopt MSA as the architectural pattern in the technical architecture design of the Lynx platform** except for the requirement stated before that does not conform well to the MSA pattern and therefore uses the SOA pattern.

4 LOGICAL ARCHITECTURE DESCRIPTION

The logical architecture of the system focuses on the interfaces of its software components and how they are wired together to form a larger software system. The aim is to provide developers with a clear architecture perspective on the entire system without constraining the architecture to a particular technology or environment.

In deliverable D1.2 "Technical requirements analysis report", we presented the components of the Lynx platform as building blocks (BBs) dividing them into two categories namely: foundational and peripheral BBs; but since BBs are loosely coupled architecture components with only one main responsibility and a well-defined interface, a definition that is synonymous to microservices, **we henceforth substitute the name BB with the name "microservice" reflecting a choice of the architectural design pattern** (see section 3.5).

Consequently, all the foundational BBs shall be referred to as **foundational microservices** while the peripheral BBs shall be called **peripheral microservices** throughout this document.

4.1 ACCESS CONTROL SYSTEMS DESIGN

In this section, we introduce the JSON Web Tokens standard and the OAuth2 framework as they are necessary prerequisites to the understanding of the concepts we shall discuss later in this section; particularly, the detailed description of the Lynx access control designs.

4.1.1 JSON Web Tokens (JWTs)

JSON Web Token (JWT) is an open standard (RFC 7519) [6] that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.

Tokens can be encrypted and/or signed; signed tokens can be used to verify the integrity of the claims contained within it, while encrypted tokens hide those claims from other parties. Json Web Tokens can be signed using asymmetric or symmetric keys.

A JWT comprises three parts separated by a dot (.) as follows:

- **Header**: the header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC-SHA256 or RSA.
- **Payload**: this part contains the claims. Claims are statements about an entity (typically, the user) and additional data.
- **Signature**: the signature is used to verify the message wasn't changed along the way, it ensures that the token is valid.

Therefore, a complete JWT is represented as **Header.Payload.Signature**

For the Lynx architecture design, we are using the signed JWTs; each time a request is sent to a server, the server can determine:

- The validity of the token based on the signature and also the expiration time.
- The identity of the client and its authorities.

4.1.2 OAuth 2.0 protocol

From the early stages of the design of the Lynx platform, we decided to adopt the OAuth 2.0 protocol to model our authorization flows. OAuth 2.0 is an industry-standard protocol for authorization that focuses on client developer simplicity while providing specific authorization flows for different types of applications [7].

The OAuth 2.0 defines four roles for owners, clients and servers as follows:

- **Client application**: this is an application that can make protected resource requests on behalf of the resource owner.
- **Resource owner**: this entity owns the resource and therefore can grant access to a protected resource or a service. Usually, the resource owner is a person or an application.
- **Resource server:** this is the server hosting the protected resource. The resource server has to implement the access control policies for the hosted resources.
- **Authorization server:** this server supplies access tokens to the client after successfully authenticating the resource owner.

The OAuth 2.0 framework also defines a group of grant types that can be used to get an access token. The grant types that will be used in the Lynx platform are:

- **Client credentials:** "the client credentials grant is used when applications request an access token to access their own resources, not on behalf of a user" [8] (i.e. the client application is the resource owner). It is important to notice that in the Lynx project context, all the client applications from the pilot use cases will own the service resources associated to their corresponding pilot use case.
- **Password grant**: the password grant type is used by first-party clients to exchange a user's credentials for an access token. This authorization flow will be used to enable password logins from the Lynx client application that will be accessible from all Lynx users.

Clients can use the granted access tokens to access protected resources. The complete specification of the OAuth 2.0 framework is currently available at [9]. Figure 2 shows the control flows for the client credentials grant type with the adoption of JWT.

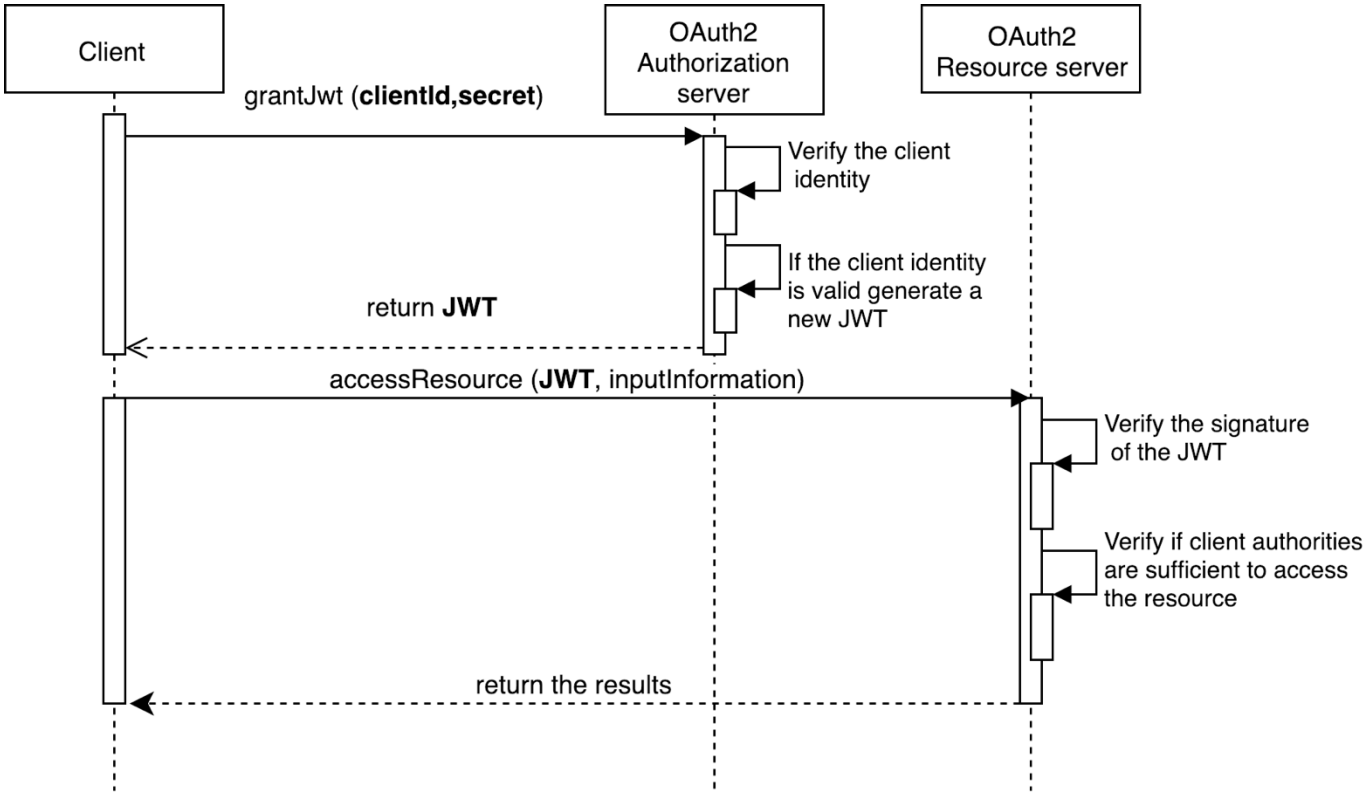

Figure 2 UML sequence diagram of the OAuth2 client credentials grant type with JWT tokens

4.1.3 Authentication

The term authentication describes the process of validating an entity's credentials to verify its identity. The Lynx platform provides authentication mechanism capable of verifying the clients' applications and users' identities that are based on the OAuth 2 framework.

4.1.4 Authorization

An authorization process defines policies that determines the resource(s) an entity is permitted to access after a successful authentication of the entity (client application or user in the Lynx context). The Lynx platform defines some permission levels (authorities) for its client applications and users based on the OAuth 2 framework.

4.1.5 Access control

Under the microservice architecture, an application is split into multiple microservice processes, and each microservice implements the business logic of one module in the original single application. Therefore, there is the need to protect the resources of each microservice by granting authorization to the users with the right authorities.

It is worth noting that, **all the Lynx microservices will be resource servers and so they will be responsible for protecting their hosted resources (services or data)**; therefore, access control to resources in the Lynx platform can be conceptualized and divided into four security levels:

- At the level of the API Gateway.
- At the level of the Workflow Manager.
- At the level of the Peripheral Microservices.
- At the level of the LKG Manager.

An API key can also be used to protect the resources of external Microservices. Table 2 shows the different security levels that we have envisioned for the Lynx platform, their access control granularity and their required information.

Security Level	Access Control Granularity	Required information
API Gateway	Workflow	Original Client JWT
Workflow Manager	Workflow	Original Client JWT
Peripheral Microservices	Task	Original Client JWT + API Key (only for external Microservices)
LKG Manager	Document, Annotation	Original Client JWT + API Key (possibly)

Table 2 Access control levels for the Lynx platform

4.2 MICROSERVICES COORDINATION SYSTEM DESIGN

One of the main requirements of the Lynx platform is to process business processes composed of many atomic service tasks. Examples of these business processes are the curation workflows associated with each use case that are described in D4.2.

There are two different approaches to controlling business processes in microservice architecture:

- Orchestration: a centralized approach in which a "central brain" orchestrates the other microservices driving by itself the execution of the business processes.
- Choreography: a distributed approach in which the microservices observe an event environment and act reacting to events autonomously; therefore, advancing the state of execution of the business process.

Figure 3 offers a graphical interpretation of the difference between orchestration and choreography.

Microservices architecture normally uses choreography to drive the purpose of low coupling (an important factor to achieving high scalability); nevertheless, **for the Lynx control system design, we opted for orchestration**. The main reasons are:

- Monitoring of the state of execution is easier with orchestration.
- Modelling business process with events is difficult (it is necessary to have a tool that can translate a business process model into an event model)

In addition, we envisioned the possible "upgrading" of the system to a choreography-based control system in the future.

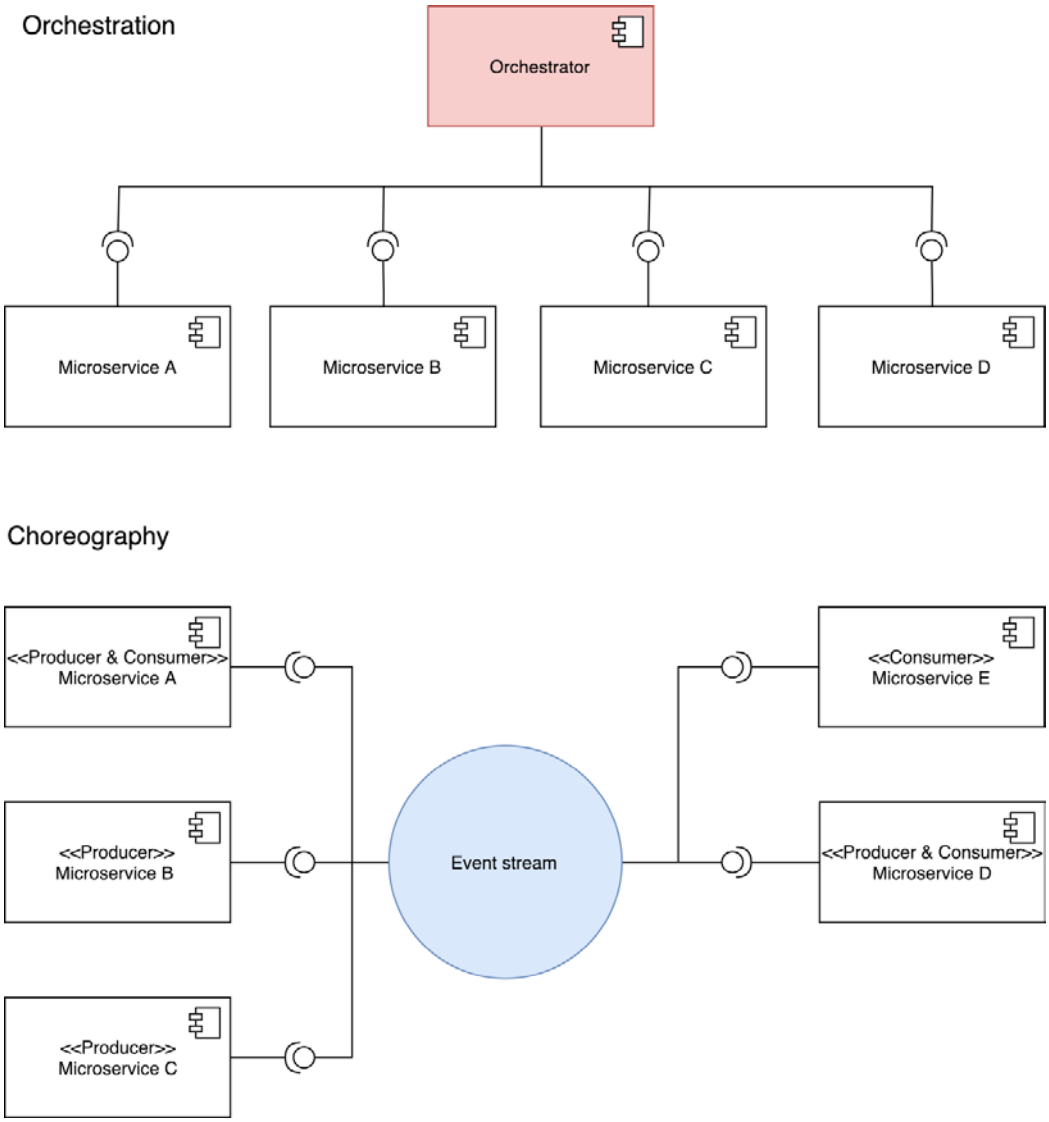

Figure 3 Orchestration and choreography, two possible control approaches in service-oriented architectures

4.3 SERVICE INTERCOMMUNICATION SYSTEM DESIGN

Since the Lynx architecture shall comply with the REST architectural pattern (as specified in TR4 of D1.2), we are constrained to using the HTTP protocol for the design of the communication system.

For the Lynx platform, we envisioned two types of communication:

- Synchronous: the client sends a request opening a connection with the server and awaits the result, once the result is returned, the connection is closed.
- Asynchronous: in our context (inter-service communication), asynchronous communication is a means whereby a client makes a request to a server, opening a connection, then it closes it and keeps on executing without blocking the connection. The result is delivered to the client later upon completion.

Two possible benefits of asynchronous communication are:

- An increase in performance is possible since the client is not blocked just for awaiting a response for the required information; within the same timeframe, it is able to deliver more.
- Optimization of network utilization is attainable since a connection between the client and server is terminated immediately a request is sent freeing up network resources for other purposes.

To enable asynchronous communication, we envisioned the two following requirements:

- An asynchronous delivery mechanism over the HTTP protocol (required by REST), like WebHook or Polling that will be explained in this section.
- The program that made the request has to be capable of continuing the execution even if the response is not ready.

While the first requirement is sufficient to optimize the network utilization, the second is necessary to increase the performance of the system.

4.3.1 WebHook

A Webhook is a type of asynchronous delivery mechanism where data are sent between two service applications over the HTTP protocol; the flow is as follows:

1. The requesting service provides a callback URL to the endpoint where the data or information will be consumed.
2. The endpoint will post any new occurring event to the specified callback URL.

One strong requirement to fulfill for the adoption of WebHook is that, the client must host an HTTP server with an endpoint to handle the callback when it occurs.

As shown in Figure 4, serviceA makes a request to serviceB and provides a callback URL which is used by serviceB to send back the response after the task has been completed and the data has been elaborated.

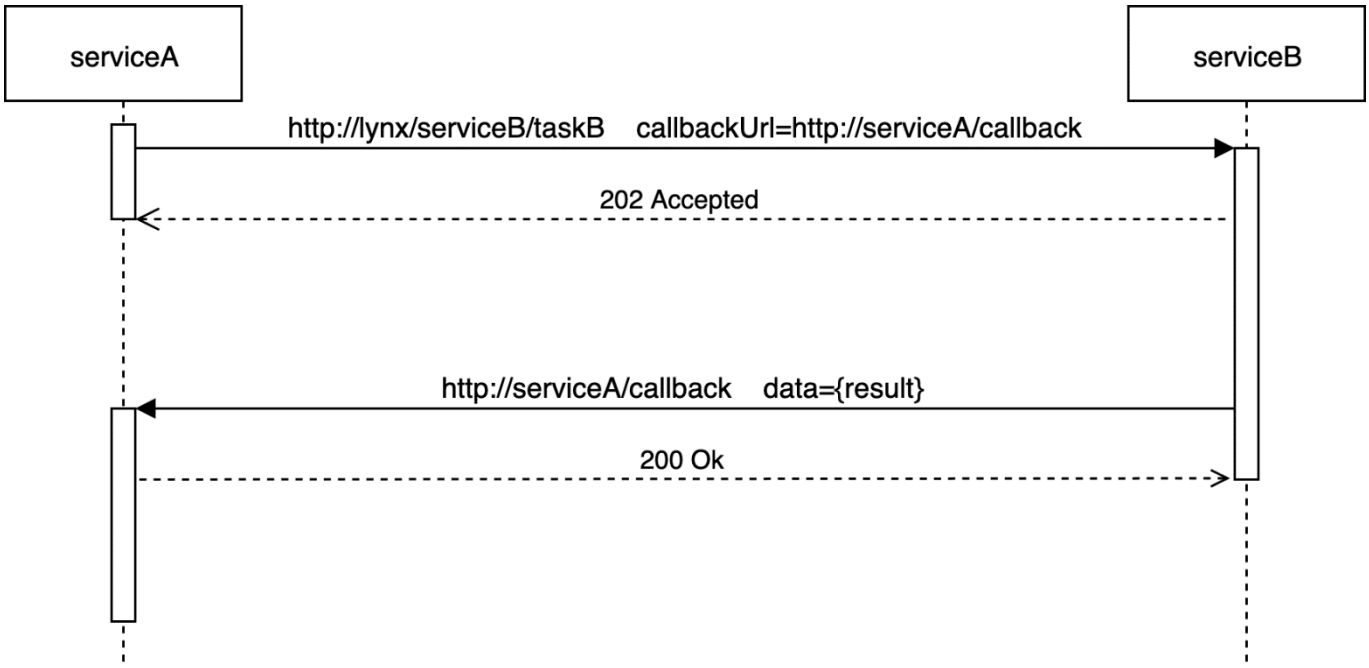

Figure 4 Asynchronous communication using webhook mechanism over the HTTP protocol

4.3.2 Polling

Polling is a technique where a service application makes a request to an endpoint for a task execution, data or information consumption and continuously in a pre-determined frequency polls the status endpoint of the server service application for the result.

There are two types of polling: the classical (short) polling and the long polling. For the purpose of this document, we shall describe only the classical polling. As shown in the diagram in Figure 5, serviceA makes a request to serviceB for a task execution and terminates the connection; but periodically polls the status endpoint of serviceB for the status of the task.

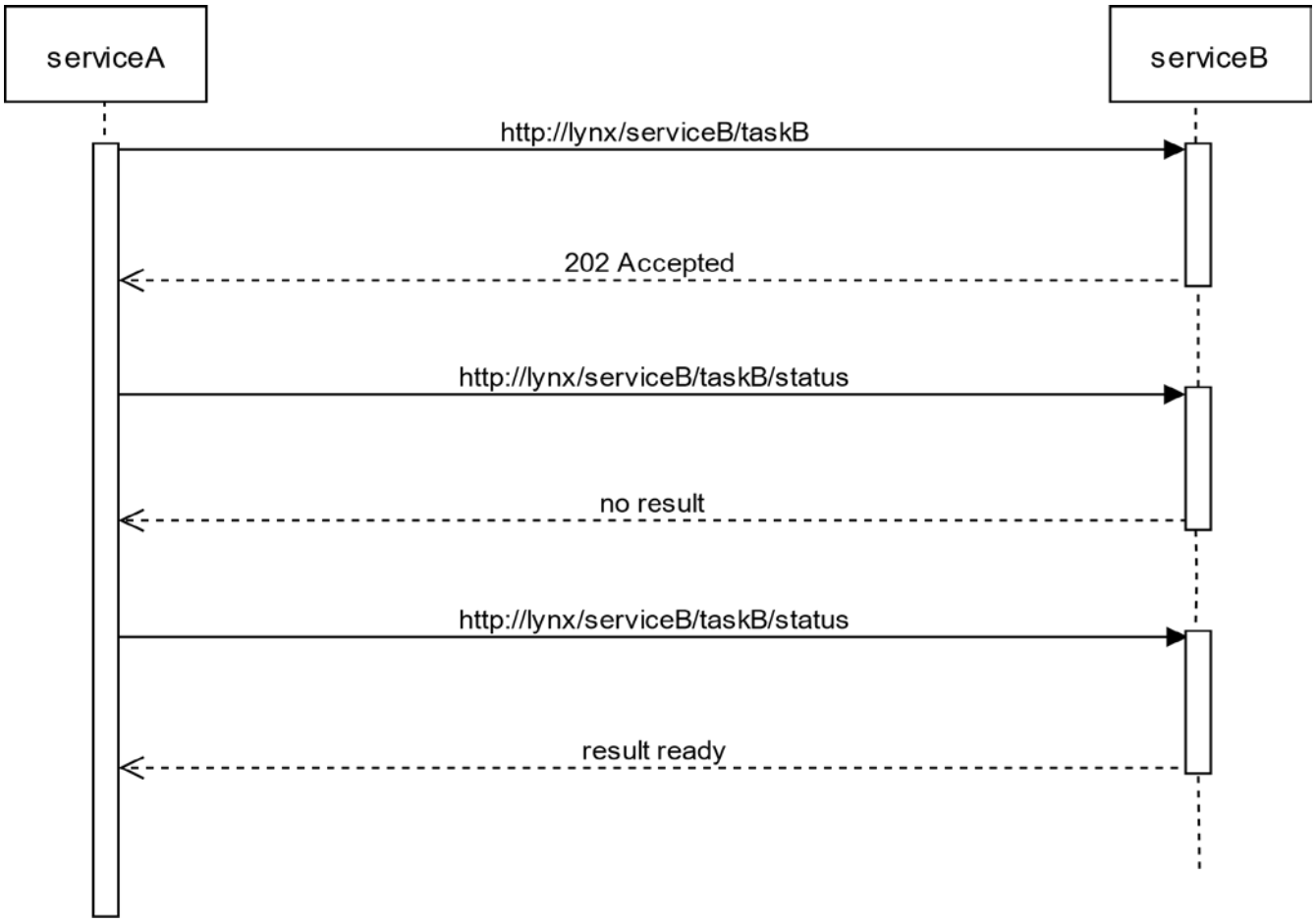

Figure 5 Asynchronous communication using short polling mechanism over the HTTP protocol

4.4 FOUNDATIONAL MICROSERVICES

The foundational microservices are the direct consequence of the previous systems design (see section 3 and 4).

Foundational microservices form the skeleton of the system and their interfaces are unlikely to change and therefore give a logical or conceptual architectural view. As a result, the behavior and interfaces of these components are architecture relevant and will be discussed in this document.

Peripheral microservices, on the other hand, depend strictly on the design of the foundational microservices for their operation and functioning; thus, they do not influence the architectural design and therefore will not be described in this document.

This section presents a description of the **Lynx foundational microservices** focusing on their main responsibilities.

4.4.1 API manager (API)

A common problem of service-oriented architectures is the mismatch in granularity between the APIs of the individual services and the data required by the clients.

A standard way to solving this problem is by the adoption of an API gateway pattern. The purpose of this pattern is to provide an additional layer of abstraction between the client applications and the microservices.

The main responsibility of the API gateway is to perform routing of the incoming (external) requests to the correct microservices and possibly enforcing throttling and security policies.

The Lynx API manager implements the **API gateway pattern** and inherits its main responsibility (section 4.1.1 of this document); moreover, it will also be responsible for:

- The very first level of access control (see Section 4.2).
- The throttling of the incoming requests in order to mitigate overwhelming the platform with too many requests within a specified time interval and against possible Denial of Service attacks.

With an API manager, the platform is guaranteed to provide well-defined and secure public APIs to its client applications and users.

4.4.2 Workflow Manager (WM)

The workflow manager block is a very crucial component of the Lynx architecture since it is responsible for the effective orchestration of the microservices for the execution of workflows.

In the Lynx project scope, workflows are combinations of both parallel and sequential tasks and are specified using Directed Acyclic Graphs. The initial workflows for the pilot use cases are specified in D4.2.

It has been established that, the document interchange format will be NIF 2.1 with an asynchronous communication between the WM and the microservices.

4.4.3 OAuth 2.0 Authorization server and Identity manager (AUTH)

The AUTH microservice has two main responsibilities:

1. Managing of the users and client applications identities and the associated information.
2. Supporting the **authorization server** functionalities for the OAuth 2.0 protocol (see section 4.1.3).

For the first point, we included in the interface a set of CRUD (Create, Read, Update and Delete) operations both for the **User** and the **Client Application** data models.

With regards to the second point, the OAuth 2.0 authorization flows that will be initially supported by the Lynx platform are **client credentials** and **password.**

4.4.4 LKG Manager (LKGM)

The LKG Manager forms a central part of the Lynx platform in terms of the general platform functioning capabilities; this is where the LKG is stored and maintained. Its basic functionalities include the storing of documents and their annotations (in the form of RDF); with special emphasis on keeping the synchronization among them, providing read and write access, and update of documents and annotations.

The LKG Manager can be queried in terms of annotations (e.g. “which documents contain mentions of this entity”), and in terms of documents (e.g. “what are the contents / annotations of document X”). The former are implemented via a SPARQL query to the underlying triplestore while the latter as a combination of similar queries and queries to a document indexer. All access to the LKG Manager is done via a REST interface which is still under specification with its current documentation here. The interface includes a set of CRUD APIs to manage the following specifications within the Lynx platform: **documents** and **collections**.

- A **document** is a piece of information in plain text, RDF or JSON format which may contain content and annotations. Technically, annotations are metadata as they give additional information about an existing piece of data; a document is stored in the LKG as a set of triples.

- A **collection** represents a group of documents with an associated label. A document can belong to many collections and deleting a collection does not delete the documents therein.

There are some important parameters worth mentioning during the design of the CRUD APIs:

- The **document parameter**- it is either an RDF following the model description online here or a JSON model.
- The **format parameter**: this parameter has possible values of RDF, TXT and JSON.

4.5 LOGICAL ARCHITECTURE VIEW

A view of the Lynx logical architecture is shown in Figure 6: the UML component diagram describes the dependencies between the Lynx microservices and their REST interfaces; the dependencies highlighted in grey are necessary to perform asynchronous communication between microservices using the webhook mechanism described in 4.3.1.

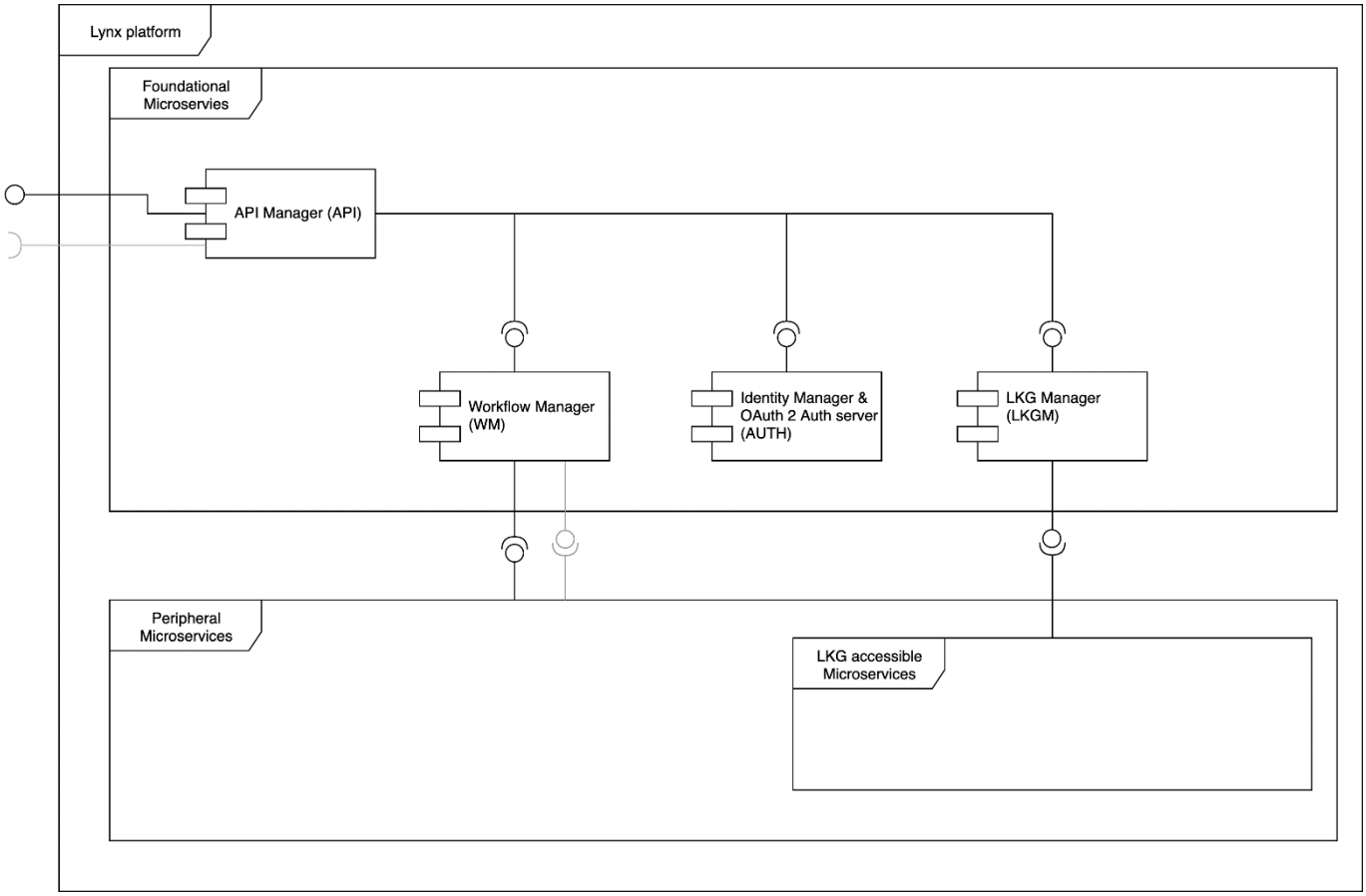

Figure 6 UML component diagram of the Lynx logical architecture

5 PHYSICAL ARCHITECTURE DESCRIPTION

The physical architectural description of a software system describes amongst other things the installation, configuration, and deployment of the software application and how to deliver the deployable system.

The choice of a proper infrastructure for the deployment is closely related to the deployment architecture design; and considers primarily the technical requirements of the system such as availability, reliability, fault-tolerance, performance, and scalability. In addition, we established from D1.2 that **the Lynx deployment architecture shall support different infrastructures within different environments**.

In this section, and for the purpose of this document, we describe some concepts and integration technologies relevant to the deployment architecture of the Lynx platform: the **cloud computing** infrastructure and the **OpenShift container platform**.

Eventually we provide a comprehensive description of the deployment architecture.

5.1 CLOUD COMPUTING

Cloud computing is shared pools of configurable computer system resources and higher-level services that can be rapidly provisioned on-demand to users with minimal management effort, often over the internet. Cloud computing depends on resource sharing to achieve coherence [10].

Cloud computing exhibits different key characteristics worth mentioning such as:

- It relies on **distributed systems**
- It increases users' **flexibility** with re-provisioning, adding, or expanding technological infrastructure resources
- It provides several forms of **transparency**, amongst them are:
    - Network transparency: it is a situation where a service allows users to access a resource without the user needing to know, and usually not being aware of, whether the resource is located on a local machine or on a remote machine. It promotes device and location independence.
    - Replication and scaling transparency: replication transparency enables multiple instances of resources to be used without knowledge of the replicas by users or application programmers whiles scaling transparency allows the system and applications to expand in scale without change to the system structure or the application algorithm.
    - Failure transparency: this enables the concealment of faults, allowing users and application programs to complete their tasks despite the failure of hardware or software components.

There are different deployment models offered by cloud computing to address different organizational needs. The popular and most patronized models are:

- **Public cloud**: this type of deployment model is used to render services over a network that is open for public use and may be free.
- **Private cloud**: this is a cloud infrastructure dedicated and operated substantially for individual organizations; where the hardware, storage and network are dedicated to a single client or company.
- **Virtual private cloud:** this is an on-demand configurable pool of shared computing resources allocated within a public cloud environment; it is a multi-tenant environment where companies achieve networking isolation while keeping costs down by buying hardware slices with other tenants and creating private subnets.

- **Hybrid cloud**: this is a cloud computing environment that uses a mix of on-premises, private cloud and third-party, public cloud services with orchestration between the two platforms.

Each of the above deployment models comes with their unique advantages, a direct result of the business and requirements needs to be addressed.

Cloud computing provides these service models:

- **Platform as a Service (PaaS):** a cloud application platform that automates the hosting, configuration, deployment, and administration of application stacks in an elastic cloud environment.
- **Infrastructure as Service (IaaS):** a form of cloud computing that provides virtualized computing resources over the internet by exposing high-level APIs used to dereference various low-level network infrastructure like scaling, security, backup etc.
- **Software as a Service (SaaS):** it is a software distribution model in which a third-party provider hosts application and makes them available to customers over the internet.

5.2 DOCKER

Docker is a computer software program (open platform) for developing, shipping and running software applications. It provides the ability to package and run an application in a loosely isolated environment called a container [11].

Some important objects in Docker are the image and container objects:

- **Docker image:** it is a read-only template with all the requirements (code, system tools, system libraries and settings) for running a Docker container as well as metadata describing its needs and capability.
- **Docker container:** it is a runnable instance of an image.

Running multiple Docker containers across multiple machines is inevitable when using microservices and this entails a lot of work for example:

- Starting the right containers at the right time.
- Figuring out how these containers interact.
- Handling of storage considerations.
- Dealing with failed containers or hardware.

Doing the above manually would be a nightmare; therefore, there is the need for automation of these services and therefore we introduce Kubernetes in the next section which is an open source container orchestration platform integrated into the OpenShift container platform.

5.3 KUBERNETES

Kubernetes is an open-source container-orchestration system built on top of Docker for automating deployment, scaling and management of containerized applications.

It is a microservice friendly platform that provides a container-centric management environment. It orchestrates computing, networking, and storage infrastructure on behalf of user workloads [12].

In the OpenShift architecture, explained below, Kubernetes provides the cluster management and orchestrates containers on multiple hosts. Two important objects of Kubernetes that OpenShift leverages on are:

- **Pod**: it is one or more containers in the OpenShift container platform that are deployed together on one host; and the smallest compute unit that can be defined, deployed, and managed.

- **Service**: serves as an internal load balancer which identifies a set of replicated pods in order to proxy the connections it receives to them.

Figure 7 represents a diagrammatic illustration of the Kubernetes architecture showing important objects and their interactions. It shows one master node connected to two worker nodes with the description of the components found at [13].

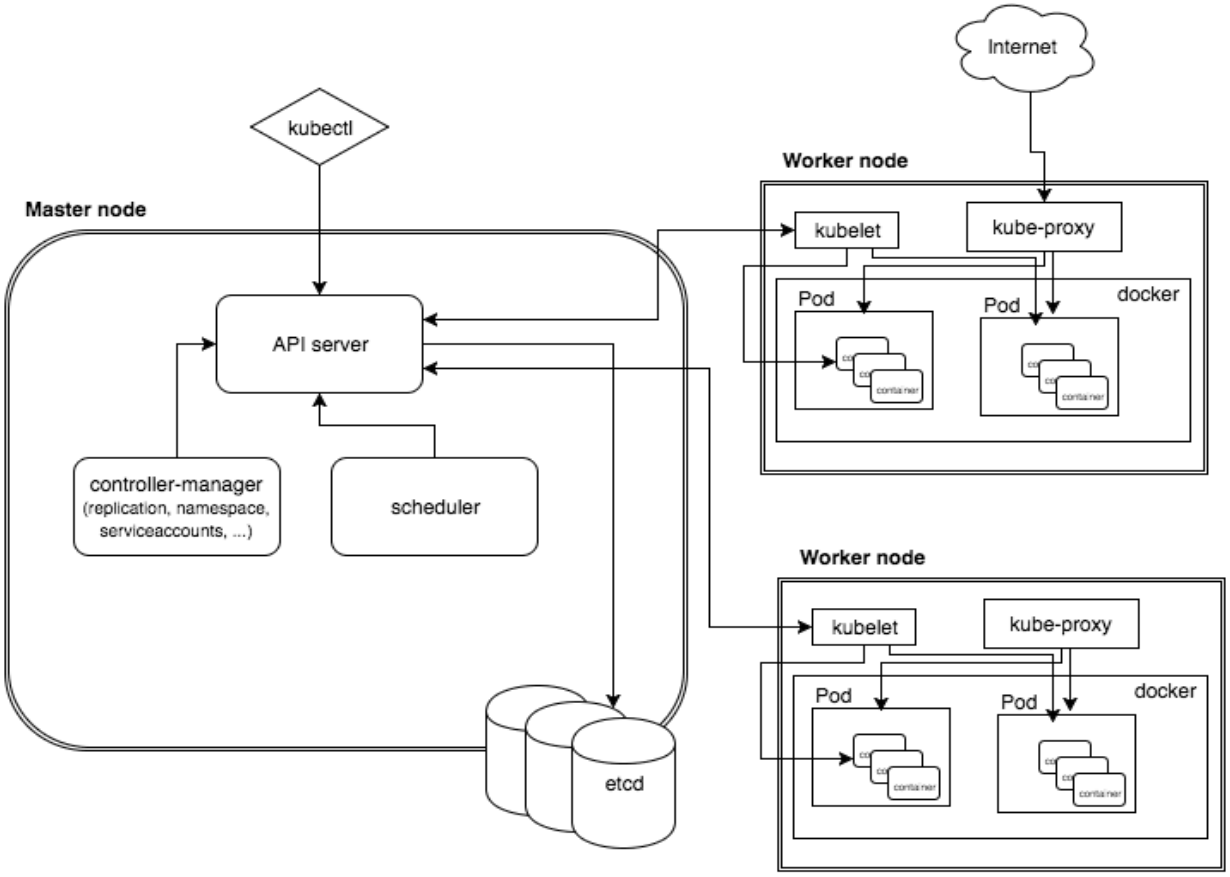

Figure 7 Kubernetes architecture

5.4 OPENSHIFT

OpenShift is a PaaS family of containerization software developed by Red Hat and built on top of Kubernetes. It is an application platform that allows automation of build, deployment and management of applications thus making developers pay attention to the writing of codes [14].

OpenShift, in addition to providing the same functionality of Kubernetes, comes with several advantages, amongst them are:

- It provides an easy way to deploy containers across various frameworks, languages, or databases.
- It also provides image build strategies like S2I (Source-to-Image).
- It supports continuous deployment; an important feature that helps to propagate automatically into production changes made in development without any manual intervention.

The diagram in Figure 8 shows the OpenShift architecture taken from the official Red Hat OpenShift website [15], where all the official documentation including description of the necessary components and their functions can be found.

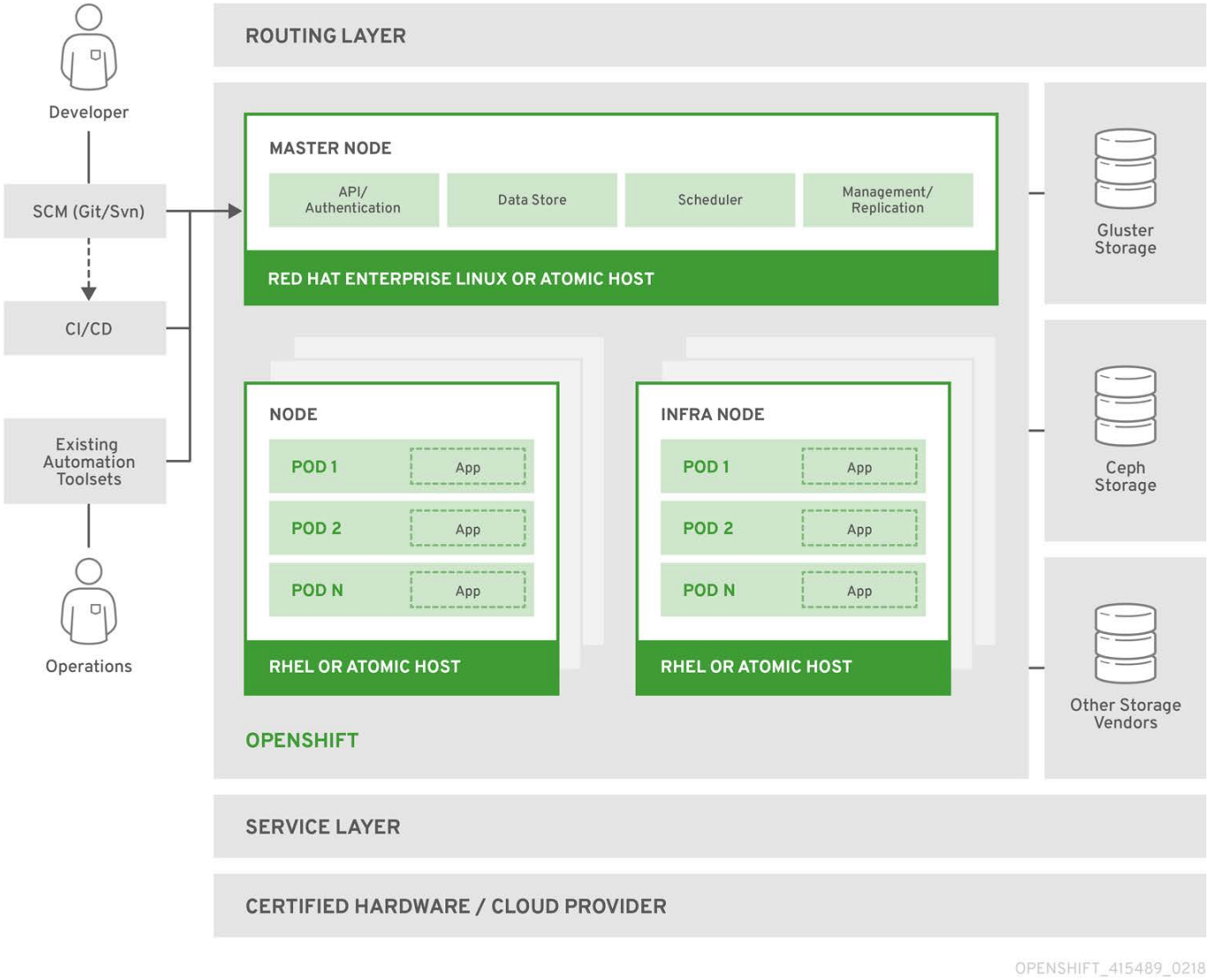

Figure 8 OpenShift architecture

5.5 DEPLOYMENT ARCHITECTURE DESCRIPTION

The deployment architecture of the Lynx platform is built on top of the OpenShift platform in order to leverage its functionalities in the quest to minimizing up-front IT infrastructure costs and allow us to get the application up faster and with different infrastructures using cloud computing (public cloud and on-premise).

The application can run in different environments which are:

- **Development**: in this environment, all the hardware, software and/or computing resources, and programming tools required to get the application built and running is provided. It also serves as an interface for testing, deployment, integration, troubleshooting and maintenance services.
- **Production**: the production environment is the setting where the software application and other products are put into operation for their intended uses by the client applications and their unique end users; making the software application available as a SaaS model.
- **On-premise**: the on-premise environment which will be used particularly by OpenLaws is an environment where the software application is installed and operated from the premises of a client.

The development environment is currently powered by eww ITandTel (https://www.eww.at/business/itandtel/) located in Austria through the IaaS model.

It is worth noting that, two microservices namely: Dictionary Access and Machine Translation will be deployed outside any environment described above; they will remain on the infrastructure of KD and Tilde respectively.

Figure 9 presents a simple view of the Lynx deployment architecture: its design is mainly focused on flexibility with regards to the infrastructure.

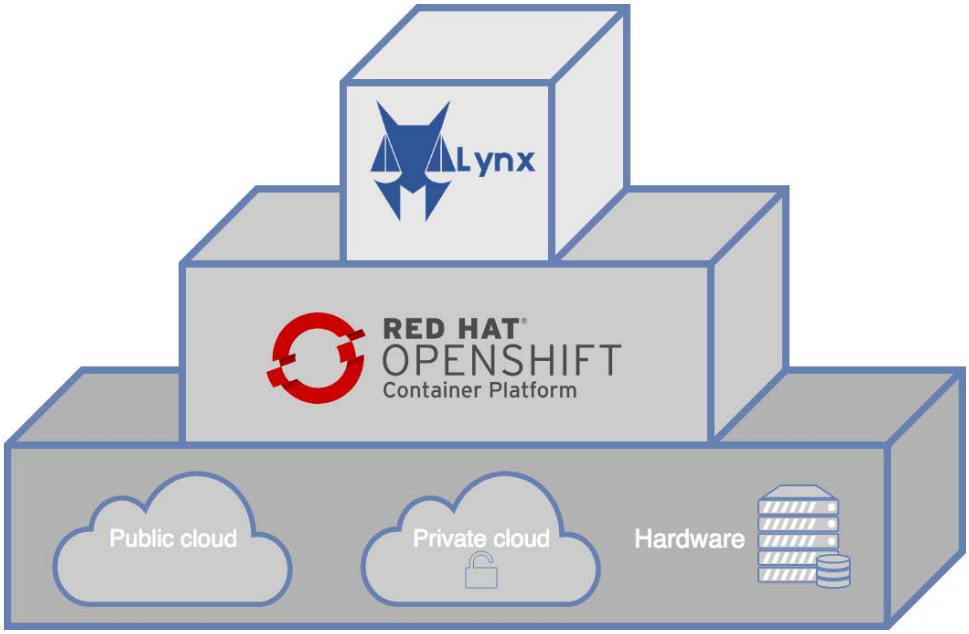

Figure 9 Lynx deployment architecture

6 CONCLUSIONS AND FUTURE WORK

In this deliverable, we analyzed and discussed some architectural design patterns frequently used in software systems such as Monolithic architecture pattern, SOA and MSA.

For the Lynx platform, we adopted the **MSA pattern** which supports loose coupling between services; a feature that allows for easy separation of work and which makes this pattern most adaptable to the Lynx platform requirements.

We presented a detailed study on some adopted patterns and industry standards after which we turn our focus to describing the logical architecture of the platform; this involved the description of the foundational microservices and a view of the architecture that focused on the dependencies between microservices and their interfaces.

The physical architecture is also presented where we provided a deployment architecture view of the platform. The **OpenShift platform** upon which the deployment architecture of the Lynx platform is built supports the minimization of up-front IT infrastructure costs, supports continuous deployment, allows the getting up of applications faster and with different infrastructures using cloud computing.

The realization of the first prototype of the platform shall take place in T3.5 “Services/platform integration”.

7 REFERENCES

[1] WIKIPEDIA, “IEEE 1471,” [Online]. Available: https://en.wikipedia.org/wiki/IEEE_1471. [Accessed 26 February 2019].

[2] R. M. H. R. P. S. M. S. Frank Buschmann, Pattern-Oriented Software Architecture, Chichester: John Wiley & Sons, 1996.

[3] WIKIPEDIA, “Monolithic system,” [Online]. Available: https://en.wikipedia.org/wiki/Monolithic_system. [Accessed 26 February 2019].

[4] Wikipedia: The Free Encyclopedia, “Service-oriented architecture,” 2018. [Online]. Available: https://en.wikipedia.org/wiki/Service-oriented_architecture.

[5] R. C. Martin, ““The Single Responsibility Principle”,” [Online]. Available: http://blog.8thlight.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html.

[6] Auth0, “Introduction to JSON Web Tokens,” [Online]. Available: https://jwt.io/introduction/. [Accessed 31 01 2019].

[7] OAuth 2.0, [Online]. Available: https://oauth.net/2/. [Accessed January 2019].

[8] OAuth 2.0, “Client Credentials,” [Online]. Available: https://www.oauth.com/oauth2-servers/access-tokens/client-credentials/.

[9] I. E. T. F. (IETF), “The OAuth 2.0 Authorization Framework,” [Online]. Available: https://tools.ietf.org/html/rfc6749. [Accessed 2 January 2019].

[10] WIKIPEDIA, “Cloud computing,” [Online]. Available: https://en.wikipedia.org/wiki/Cloud_computing. [Accessed 03 January 2019].

[11] WIKIPEDIA, “Docker(software),” [Online]. Available: https://en.wikipedia.org/wiki/Docker_(software). [Accessed 2nd January 2019].

[12] WIKIPEDIA, “Kubernetes,” 31st December 2018. [Online]. Available: https://en.wikipedia.org/wiki/Kubernetes. [Accessed 2nd January 2019].

[13] WIKIPEDIA, “Kubernetes Components,” [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/. [Accessed 27 February 2019].

[14] R. Hat, “openshift,” [Online]. Available: https://www.openshift.com. [Accessed 2nd January 2019].

[15] I. Red Hat, “Architecture Overview,” [Online]. Available: https://docs.openshift.com/online/architecture/index.html. [Accessed 3 January 2019].

ANNEX 1: ADDITIONAL FUNCTIONAL REQUIREMENTS

In this section, we provide some functional requirements to compliment those stated in the functional requirements analysis report D1.1 that the Lynx platform shall fulfill in the quest to developing a well-suited platform that provides up-to-date legal compliance information to its clients.

Functional requirements can be considered as behavioural requirements and may involve calculations, technical details, data manipulation and processing, and other specific functionalities that define what a system is supposed to accomplish.

Administrators of any service platform as in the case of lynx are entitled to perform certain operations for which the functional requirements of the platform should be able to adequately address. Taking into consideration the business requirements and the pilots requirements analysis report D4.1, it is evident that there is an absolute need to gather and specify some functional requirements for the Lynx platform that shall drive the processing of some back-office operations.

Consequently, we specify some functional requirements that the Lynx platform shall fulfill specifically for the back-office interface as detailed in

ID	Interfaces	Requirement Short Name	Requirement Description	Priority
FR1	Back-office, REST APIs	CRUD operations for users and client applications	The administrators of Lynx shall be able to perform CRUD operations for users and client applications through an administrative back office interface and through the Lynx REST APIs.	must
FR2	Back-office, REST APIs	CRUD operations for the LKG	The administrators of Lynx shall be able to perform CRUD operations for documents, collections and annotations through an administrative back office interface and through the Lynx REST APIs.	must
FR3	Free Search UI, REST APIs	Simple search of documents	All Lynx users (also not registered) shall be able to search public documents in the LKG through a user interface and through the Lynx REST APIs.	must

ANNEX 2: API SPECIFICATION STANDARD

In developing software related systems, several parameters and components must conform to certain acceptable standards and specifications that aid readability, compatibility and integrability with other systems using the same standards and specifications. This is important because it enables a worldwide acceptability and conformity of products and services.

Furthermore, the distributed, multi-disciplinary, multi-team nature of the Lynx project requires a well-established, and strong emphasis on the enforcement of standards and shared semantics in order to guarantee the smooth transition, exchange of information and reliable processing of data.

The Lynx system as a service platform exposes a lot of REST APIs for consumption by application clients and users and therefore the need for a specification method that is acceptable and widely used to describe these endpoints.

We, therefore, adopted the OpenAPI 3 specification as the description format for specifying our APIs for consumption.

The complete OpenAPI 3 Specification of the Lynx services can be found on the Lynx website: http://lynx-project.eu/api/doc/.