COGITO

CONSTRUCTION PHASE DIGITAL TWIN MODEL

cogito-project.eu

D7.2 – Digital Twin Platform Design & Interface Specification v2

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 958310

D7.2 – Digital Twin Platform Design & Interface Specification v2

Dissemination Level:	Public
Deliverable Type:	Report
Lead Partner:	UCL
Contributing Partners:	UPM, Hypertech, DTU, UEDIN, CERTH, NT, BOC-AG, QUE
Due date:	31-10-2022
Actual submission date:	01-11-2022

Authors

Name	Beneficiary	Email
Kyriakos Katsigarakis	UCL	k.katsigarakis@ucl.ac.uk
Giorgos N. Lilis	UCL	g.lilis@ucl.ac.uk
Zi Fang	UCL	zi.fang.15@ucl.ac.uk
Dimitrios Rovas	UCL	d.rovas@ucl.ac.uk
Raúl García-Castro	UPM	rgarcia@fi.upm.es
Salvador Gonzalez-Gerpe	UPM	salvador.gonzalez.gerpe@upm.es
Apostolos Papafragkakis	Hypertech	a.papafragkakis@hypertech.gr
Giorgos Giannakis	Hypertech	g.giannakis@hypertech.gr
Thanos Tsakiris	CERTH	atsakir@iti.gr
Evangelia Pantraki	CERTH	epantrak@iti.gr
Apostolia Gounaridou	CERTH	agounaridou@iti.gr
Michalis Chatzakis	CERTH	mchatzak@iti.gr
Tasos Sinanis	CERTH	tasos.sinanis@iti.gr
Vasilis Dimitriadis	CERTH	dimvasdim@iti.gr
Damiano Falcioni	BOC-AG	damiano.falcioni@boc-eu.com
Robert Woitsch	BOC-AG	robert.woitsch@boc-eu.com
Martin Straka	NT	straka@novitechgroup.sk
Bohuš Belej	NT	belej@novitechgroup.sk
Panos Andriopoulos	QUE	panos@que-tech.com
Frédéric Bosché	UEDIN	f.bosche@ed.ac.uk
Martín Bueno Esposito	UEDIN	martin.bueno@ed.ac.uk
Jochen Teizer	DTU	teizerj@dtu.dk
Karsten Johansen	DTU	kawj@dtu.dk

Reviewers

Name	Beneficiary	Email
Giorgos Giannakis	Hypertech	g.giannakis@hypertech.gr
Panos Andriopoulos	QUE	panos@que-tech.com

Version History

Version	Editors	Date	Comment
0.1	UCL, UPM	20.09.2022	ToC
0.3	UCL	10.10.2022	Draft version of section 2,3,4
0.6	UCL, UPM	15.10.2022	Draft version of sections 5,6,7,8,9
0.8	Hypertech, QUE	27.10.2022	Internal review

0.9	UCL, UPM	29.10.2022	Internal review comments addressed
1.0	Hypertech, UCL	01.11.2022	Submission to EC

Disclaimer

Executive Summary

The COGITO deliverable “D7.2 Digital Twin Platform Design & Interface Specification v2” aims to document the COGITO Digital Twin Platform detailed architecture and report the outcomes of work performed in “T7.1 Digital Twin Platform Design & Interface Specification”. Overall, the COGITO Digital Twin Platform lies at the core of the COGITO system and is responsible for implementing an information management solution that aims to enable interoperability with existing standards and ontologies covering different domains.

To ensure the COGITO functional requirements are met using a generic set of reusable components, the COGITO Digital Twin Platform (DTP) is implemented using a multi-layer architecture. The **Authentication Layer** ensures that user access is restricted to specific user roles and groups. The **Data Ingestion Layer** is responsible for loading new datasets and orchestrating the execution of the Extract, Transform and Load (ETL) services for generating the knowledge graph and populating the corresponding databases. At the same time, the **Data Persistence Layer** provides a cloud-based data storage solution, including graph, relational and time-series databases. The **Data Management Layer** satisfies the data needs of the other COGITO applications by managing DTP’s data-driven modules and dynamic endpoints. It offers a runtime system responsible for handling the various requests and delivering the data responses to the COGITO applications using the message broker provided by the **Messaging Layer**. The **Data Post-Processing Layer** provides software components responsible for checking and processing BIM models that conform to the Industry Foundation Classes (IFC) standard.

The document mainly focuses on presenting the operational blocks of the DTP architecture, including the layers with their subcomponents and their interconnections. In this deliverable, we describe the architecture following a hierarchical top-down approach starting from the high-level description of each layer and then drilling down with a detailed description of its components. In addition, we present the detailed technology stack used for the final implementation, we define the data flows and we analyse their interfaces: i) among the different layers of the platform and ii) the ones interacting with the other COGITO applications to ensure transparent interoperability of the COGITO solution.

Contents

List of Figures

List of Tables

List of Acronyms

Term	Description
AAI	Authentication and Authorisation Infrastructure
AMQP	Advanced Message Queueing Protocol
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BIM	Building Information Model
CoAP	Constrained Application Protocol
COGITO	Construction Phase diGItal Twin mOdel
CRUD	Create, Read Update and Delete
DAO	Data Access Object
DB	Database
DI	Dependency Injection
DT	Digital Twin
DTP	Digital-Twin Platform
DTV	Design Transfer View
ERP	Enterprise Resource Planning
ESB	Enterprise Service Bus
ETL	Extract, Transform and Load
glTF	GL Transmission Format
GUI	Graphical User Interface
HSE	Health, Safety and Environment
IFC	Industry Foundation Classes
IoC	Inversion of Control
IoT	Internet of Things
JDBC	Java Database Connectivity
JMS	Java Messaging System
JNI	Java Native Interface
JPA	Java Persistence Layer
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LoRa	Long Range
MC	Model Checking
MQTT	Message Queue Telemetry Transport
MVD	Model View Definition
OPC UA	OLE for Process Control Unified Architecture

ORM	Object-Relational Mapping
OWL	Web Ontology Language
PaaS	Platform as a Service
RDF	Resource Description Framework
RDMS	Relational Database Management Systems
REST	Representational State Transfer
SaaS	Software as a Service
SHACL	SHApes Constraint Language
SOA	Service Oriented Architecture
SQL	Structured Query Language
SSO	Single-Sign On
STEP	Standard for the Exchange of Product Data
STOMP	Streaming Text-Oriented Messaging Protocol
TD	WoT Thing Description
UDI	User-Driver Innovation
UML	Unified Modeling Language
VM	Virtual Machine
WODM	Word Order Definition and Monitoring Tool
WoT	Web of Things
XML	Extensible Markup Language

1 Introduction

This deliverable reports on the final version of the COGITO Digital Twin Platform (DTP) architecture reflecting the outcomes of "T7.1 Digital Twin Platform Design & Interface Specification". The deliverable builds upon the identified stakeholder requirements of "T2.1 Elicitation of Stakeholder Requirements", the overall COGITO system architecture of "T2.4 COGITO System Architecture Design", and the definition of the COGITO ontology network of "T3.2 COGITO Data Model, Ontology Definition and Interoperability Design". This work delivers the detailed multi-layered architecture of the COGITO DTP, the specifications of its core components, and a detailed definition of the interfaces.

1.1 Scope and Objectives of the Deliverable

The DTP follows a multi-layered software architecture comprising six core layers. Each layer contains a set of software components implementing the Service-Oriented Architecture (SOA) design pattern and performing various business logic operations to support the main objectives of the COGITO solution. The main scope of this deliverable is to present the final architecture of the DTP and analyse the layers and their roles in supporting the user-facing COGITO software applications. The objective is to describe: i) the high-level software architecture of the DTP; ii) decompose each layer into distinct software components; and iii) explore the technological stack and interfaces required for the implementation.

A centrally integrated solution appears in the proposed ICT framework of several construction-related H2020 projects and other commercial organisations. In the **SPHERE** project, the raw data produced by machines, systems and products are linked, captured, and managed using a central cloud-based collaborative Platform as a Service (PaaS) ICT solution [1]. In **BIMprove**, the core component is a cloud-based data integration service using modular APIs for information exchange and data processing. These APIs can add/remove and update information in the different layers of the BIMprove solution [2]. Finally, in **ASHVIN,** a microservices-based messaging IoT middleware is proposed [3]. The middleware abstracts the most common IoT protocols specifications, such as LoRa, MQTT, OPC UA, CoAP to establish a unified communication interface between devices and software applications.

COGITO's DTP is a cloud-based and semantically enabled data integration middleware that includes a comprehensive suite of services to guarantee scalability, reliability, and enhanced security as it is responsible for: a) handling data from various input sources such as point clouds, 4D BIM, and IoT devices, b) populating the internal data models and knowledge graphs, and c) responding to data requests of the various COGITO applications. The semantically linked knowledge graph and the COGITO ontologies are defined in WP3.

The DTP architecture design considers the variable computation data needs for providing synchronous and asynchronous interfaces to increase the performance by minimising the query response time as much as possible. Additionally, the functional components will be deployed as microservices, enabling flexibility and high-availability capabilities. This renders COGITO's DTP into a federated 'loose' system of interoperable tools instead of a rigid, tightly integrated system.

1.2 Relation to other Tasks and Deliverables

This deliverable is the outcome of the "T7.1 Digital Twin Platform Design & Interface Specification", which falls under "WP7 COGITO Digital Twin Platform" activities. There are several dependencies of this work on other deliverables and tasks:

- The configuration of the user roles in the Authentication Layer is based on the work performed in "T2.1 Elicitation of Stakeholder Requirements" and the corresponding deliverable "D2.1 Stakeholder requirements for the COGITO system".
- The layered architecture design of the platform is primarily based on the work performed in "T2.4 COGITO System Architecture Design" and the corresponding deliverable "D2.5 COGITO System Architecture v2".
- Furthermore, the design of the Data Ingestion and Data Persistence layers is based on the work performed in "T3.2 COGITO Data Model, Ontology Definition and Interoperability Design".

1.3 Structure of the Deliverable

The rest of the deliverable is organised according to the structure of the COGITO DTP:

- Section 2 presents the overall architecture of the DTP, introducing its layers and their interfaces.
- Section 3 presents the architecture of the *Authentication Layer*, which provides a central identity and access management solution, and ensures that access is restricted to specific users and applications with appropriate permissions.
- Section 4 presents the architecture of the *Data Ingestion Layer*, which provides a web-based application for loading the input data and orchestrating the execution of the Model Checking (MC) and Extract, Transform and Load (ETL) services. These services are responsible for validating the input data and populating the corresponding databases of the Data Persistence Layer.
- Section 5 presents the architecture of the *Data Persistence Layer*, which provides a cloud-based data storage solution including graph, relational and time-series databases.
- Section 6 presents the architecture of the *Data Management Layer*, which manages the execution of the data-driven modules used for handling the requests of the various COGITO applications.
- Section 7 presents the *Messaging Layer* and the proposed technologies for handling asynchronous messages and notifications.
- Section 8 presents the architecture of the *Data Post-Processing Layer*, which provides a comprehensive set of microservices used for preforming time-consuming processes implementing asynchronous communication patterns.
- The document concludes with Section 9, where some possible future improvements are discussed.

1.4 Updates to the first version of the DT Platform Design & Interface Specification

The first version, "D7.1 - Digital Twin Platform Design & Interface Specification v1", presented the design and specification of a multi-layered platform, including the layers with their software components and interconnections. Since the submission of the first version, some software components have been added or changed to meet the functional and non-functional requirements of the COGITO solution. This deliverable comprises the following changes:

- The Messaging Layer is responsible for transmitting data and notifications between the DTP and the other COGITO applications in an asynchronous manner. The first version included an Enterprise Service Bus (ESB) solution that provided the message broker and the integration framework for configuring the necessary routing operations. In the final version, these services are managed by the Data Management Layer. Thus, the Messaging Layer now contains the ActiveMQ Artemis message broker supporting multiple messaging protocols such as AMQP, STOMP and MQTT.

- The Data Management Layer is responsible for processing the various requests performed by the other COGITO applications and sending back the requested data. The first version contained three core components: i) Runtime Environment, ii) API Wrappers, and iii) Web Services. In the final design, the Data Management Layer has been redesigned to maximise flexibility and scalability. The new design consists of the following core components:
    - The DT Runtime component is responsible for configuring and supervising the execution of the configured modules, handling their responses, and sending them back to the COGITO applications.
    - The DT Library component is responsible for providing a set of ready-made coding blocks implementing various data-processing operations required for the smooth running of the COGITO system.
- The Data Ingestion Layer is responsible for loading the input data in the DTP and orchestrating the execution of the ETL and MC services for generating the knowledge graph and populating the corresponding databases. The Data Ingestion Layer contains three core components: i) Input Data Management (previously named Project Management), responsible for project creation and project management, ii) BIM Management, responsible for handling data conforming to openBIM standards and iii) Knowledge Graph Generator, responsible for generating and validating the unified knowledge graph and for populating the corresponding databases of the Data Persistence Layer. In the final design, the architecture of the provided core components has been updated to maximise reusability and flexibility.
- The Data Pro-Processing Layer is responsible for hosting time-consuming MC and ETL processes utilising asynchronous communication methods. In the final design, the Geometric Clash Checker has been added. This component performs clash detection and semantic enrichment when relationships between building element and spatial zones are missing.

2 Overall Architecture

The DTP is a cloud-based and semantically enabled data-integration middleware that includes a comprehensive suite of services for loading, populating, and managing data used by the various COGITO applications. This section describes the multi-layered architecture of a cloud-based middleware that enables interoperability between construction-related applications developed within the COGITO project and a set of interfaces that define the high-level data interactions among the provided software components.

2.1 Multi-layered Architecture

The DTP is responsible for loading the as-designed and as-built data, populating the corresponding knowledge graphs and databases, and handling the data queries from the various COGITO applications. The platform architecture follows a multi-layer approach implementing the Service-Oriented Architecture (SOA) design pattern to guarantee horizontal scalability, reliability, and enhanced security. Figure 1 displays an overview of the architecture of the DTP, which includes the defined layers with high-level interactions and the information flow among the layers.

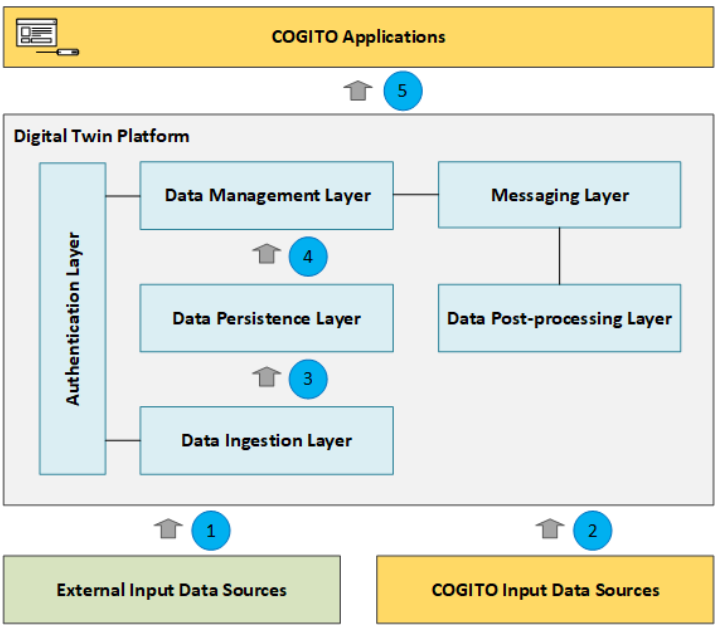


Figure 1 DTP high-level architecture

As shown in Figure 1, the final architecture includes six core layers. Each layer contains a set of software components performing various business logic operations to meet the final functional and non-functional requirements of the DTP.

- The **Authentication Layer** ensures that access to resources and services is restricted to specific users with the appropriate permissions. It is based on the open-source project Keycloak, which serves as a central identity and access management provider.
- The **Data Ingestion Layer** loads the as-designed data and real-time IoT location tracking data to the DTP and orchestrates the execution of the included Extract, Transform and Load (ETL) services to generate the unified knowledge graph and to populate the corresponding databases.
- The **Data Persistence Layer** provides an integrated cloud-based storage solution for COGITO data, providing datastores such as file storage and relational, key-value, time-series, and triplestore databases. Additionally, it provides the Thing Description Directory (TDD) which is a searchable registry of Thing Descriptions. A Thing Description (TD) is an object that follows the WoT Thing Description[1]

[1]WoT Thing Description https://www.w3.org/TR/wot-thing-description/

specification and provides metadata and interfaces of Things, where a Thing is a digital abstraction of physical or virtual entities.

- The **Data Management Layer** manages the requests of the various COGITO applications by providing i) configurable endpoints and ii) a runtime system for hosting the various modules which implement the business-logic operations for the preparation of the responses.
- The **Messaging Layer** facilitates the smooth transfer of messages between the various COGITO applications and ensures that the notifications produced by the DTP are delivered to the correct destinations. It is based on the open-source project Apache ActiveMQ Artemis offering a message broker which enables asynchronous data transmission utilising the publish/subscribe pattern using queues and topics.
- The **Data Post-Processing Layer** provides reusable data integration and quality checking services to ensure end-user COGITO applications are provided with quality data. As COGITO solution complies with openBIM data (IFC), the following services have been developed and deployed: i) Model-View Definition (MVD) checking, ii) Geometric clash detection checking and enrichment, iii) IFC optimisation and v) B-rep geometry generation.

A quick overview of Figure 1 shows that the starting point of the information flow is the input data sources. Within COGITO, the input data come from two different sources: a) external applications **(1)** such as BIM authoring tools, construction project management tools and Enterprise Resource Planning (ERP) tools, providing the as-designed BIM model along with the corresponding construction schedules and resources data, and b) the COGITO data pre-processing tools **(2)** such as the Visual Data Pre-processing tool and the IoT Data Pre-processing tool, providing imagery and location tracking data along with their meta-data.

When external sources send data to the DTP, checking, enrichment and optimisation operations are initially performed before loading the data into the Data Persistence Layer. For instance, in the case of the BIM model, an ETL service of the Data Ingestion Layer process the IFC data and performs transformations based on a pre-defined set of mapping rules. Regarding the transformations of contextual data (construction schedule, as-planned resources) that conform to machine-readable formats, such as JSON, CSV and XML, various ETL services of the Data Ingestion Layer generate RDF data and the Thing Descriptions in line with the ontologies defined in "WP3 COGITO Data Model and Reality Capture Data Tools" [4].

The Data Ingestion Layer is responsible for orchestrating the transformation processes to generate the knowledge graph and populating **(3)** the various databases of the Data Persistence Layer.

After the ingestion process is complete, the unified knowledge graph is available, and databases of the Data Persistence Layer databases are populated. The Data Management Layer can then respond to data requests from upstream COGITO applications. It comprises the DT Runtime component which uses a set of configurable modules for retrieving data **(4)** stored in the Data Persistence Layer. In some queries, the corresponding responses require data arising from different types of databases. For instance, in the case of a 4D BIM query, two requests are performed: one to the knowledge graph for retrieving the tasks and another to the BIM database for retrieving the corresponding IFC objects. The Data Management Layer is responsible for orchestrating the internal data exchanges, harmonising the returned data, and sending responses **(5)** to the COGITO applications.

2.2 Components and High-Level Interfaces

The DTP follows a multi-layered architecture comprising six core layers. Each layer contains a set of software components implementing various business logic operations to support the main objectives of the DTP. These components are packaged and deployed as microservices in a cloud-computing infrastructure, providing flexibility, availability, and scalability. The main components included in DTP layers are the following:

The *Authentication Layer* contains COGITO's **Identity Provider** which offers a central identity and access management solution.

The *Data Ingestion Layer* contains a) the **Input Data Management** component, which is responsible for project creation, user assignment, and loading input data, b) the **BIM Management** component, which is responsible for

parsing, validating and versioning IFC data, and c) **Knowledge Graph Generator** which is in charge for generating and validating the unified knowledge graph and populating the databases of the Data Persistence Layer.

The *Data Persistence Layer* contains a) a **File Storage** system for storing files, b) a **Relational DB** for storing project and user data, c) a **Key-value DB** for storing the IFC objects, d) a **Timeseries DB** for storing IoT data, e) a **Triplestore** for storing the knowledge graph, and f) **Thing Descriptions Directory** for storing the Thing Descriptions.

The *Data Management Layer* contains a) the **DT Runtime** component, responsible for creating and executing the various data-driven modules, handling their responses, and sending them to the COGITO applications and b) the **DT Library** component, responsible for providing a set of ready-made coding blocks facilitating the COGITO developers to create their own data-driven modules using a graphical environment.

The *Data Post-Processing Layer* contains a) the **MVD Checker** component for performing model checking on the BIM model in terms of data completeness, b) the **IFC Optimiser** component for de-duplication and lossless compression of the IFC file, c) the **B-rep Generator** for generating triangulated B-rep solids of the IFC objects, and d) the **Geometric Clash Checker** component for detecting clash and containment errors and for creating additional semantic links between existing entities of the unified knowledge graph. In general, the Data Post-Processing Layer provides an extensible mechanism to host components providing additional functionalities if needed.

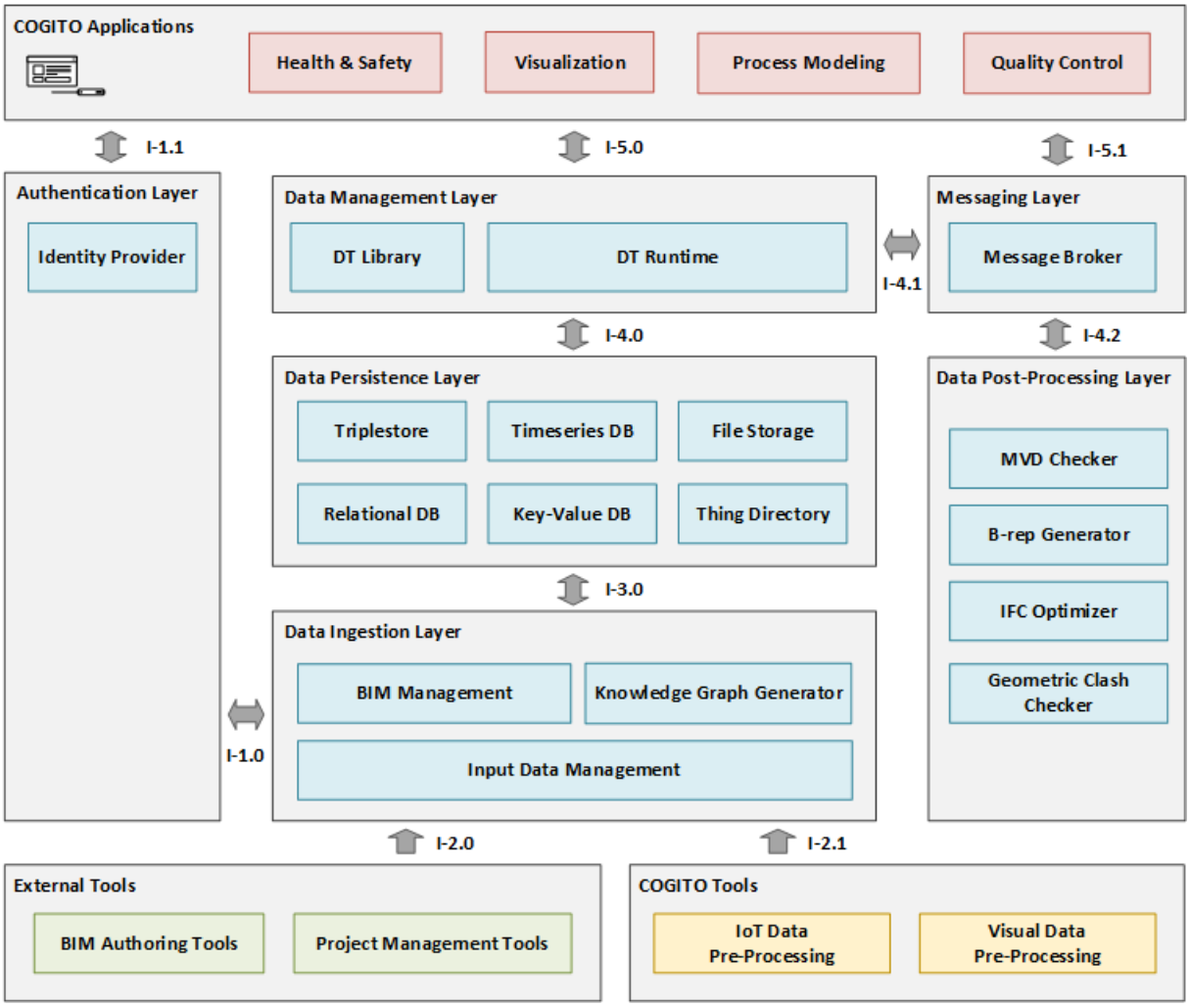


Figure 2 DTP's components and high-level interfaces

The primary interfaces have been identified and shown in Figure 2 to illustrate the data exchanges among the DTP layers' components. The main internal and external interfaces of the DTP layers and their main uses are the following:

- **I-1.0:** The Input Data Management component uses I-1.0 to register, assign roles and authenticate COGITO's users and developers.
- **I-1.1:** The various COGITO applications use I-1.1 for authenticating the registered COGITO users.
- **I-2.0**: COGITO's registered users use I-2.0 to load the input data into the DTP. The users upload data such as BIM models, construction schedules and the as-planned resources using the rich Graphical User Interface (GUI) of the Input Data Management component.
- **I-2.1**: COGITO's data pre-processing applications use I-2.1 to load processed data into the DTP. The corresponding COGITO applications use REST endpoints and KAFKA channels provided by the Input Data Management component to load various data, such as point clouds, imagery, and real-time location tracking data.
- **I-3.0:** This interface facilitates the communication between the Data Ingestion and Data Persistence layers. The Input Data Management component uses the file storage system and the relational database to store the uploaded files with their assigned permissions and meta-data. In addition, the BIM Management component uses the file storage system to store temporary files and the key-value database to store the IFC objects. In conclusion, the Knowledge Graph Generator component uses the Triplestore to store the generated RDF data and the Thing Descriptions Directory to store the Thing Descriptions.
- **I-4.0:** This interface enables the communication between the DT Runtime component of the Data Management Layer and the corresponding databases of the Data Persistence Layer. The main protocols of these interactions are JDBC and HTTPS.
- **I-4.1:** The DT Runtime component uses I-4.1 to interact with the Messaging Layer allowing the running data-driven modules of the DT Runtime component to exchange data and notifications asynchronously.
- **I-4.2:** The Data Post-Processing Layer uses the I-4.2 for triggering the execution of the post-processing software components. The business logic operations of these components often require more time, depending on the complexity of the input data files. The Data Post-Processing Layer implementation will use the Service-Oriented Architecture (SOA) design pattern, with software components provided as containerised applications. These services utilise the publish/subscribe pattern to interact with the Data Management Layer via the Messaging Layer asynchronously. Before the knowledge graph generation, the loaded IFC data are checked for schema consistency and completeness to ensure that each structural building element has semantic links with the tasks included in the construction schedule. Then, the IFC is optimised, and the geometric information is exported and converted into OBJ format.
- **I-5.0:** This interface establishes the communication between the DTP and the various COGITO applications. Offers a comprehensive and dynamic configurable REST API for exchanging data and invoking the data-driven modules of the Data Management Layer.
- **I-5.1:** Finally, I-5.1 allows the various COGITO applications to interact asynchronously with the DTP via the Messaging Layer. In this case, the Data Management Layer sends notifications to the various COGITO applications based on the business-logic implementation of the data-driven modules. Additionally, the Messaging Layer is responsible for forwarding the location tracking data streams via the I-5.1 to various COGITO applications such as the ProActiveSafety and Digital Command Centre (DCC).

The DTP supports both synchronous and asynchronous communication protocols in a unified manner. Some components can process the data requests in real-time, while others require more time depending on the data processing load. For instance, some modules of the Data Management Layer often use asynchronous communication protocols for exchanging data with the components of the Data Post-processing Layer to perform MVD-based model-checking, IFC optimisation and B-rep model generation. The software components in DTP's various layers and the detailed interface specifications are presented in the following sections of this document.

2.3 Service Orchestration

The Service-Oriented Architecture (SOA) design pattern is used to implement the DTP. The proposed multi-layered architecture requires service orchestration and an efficient messaging system that supports loose coupling between the various components of the DTP to achieve flexibility, scalability, and reliability. The Messaging Layer contains the ActiveMQ Artemis, offering a native integration environment using multiple protocols for asynchronous communication between the DTP and the various COGITO applications. The support of asynchronous messaging protocols such as AMQP, STOMP and MQTT offers many benefits and brings challenges, such as concurrency and synchronisation issues. The service orchestration is needed to ensure that the business logic operations are running smoothly and that the available computing resources are allocated correctly. As shown in Figure 3, we use Docker and Docker Swarm to provide basic capabilities such as flexibility, scalability, and reliability.

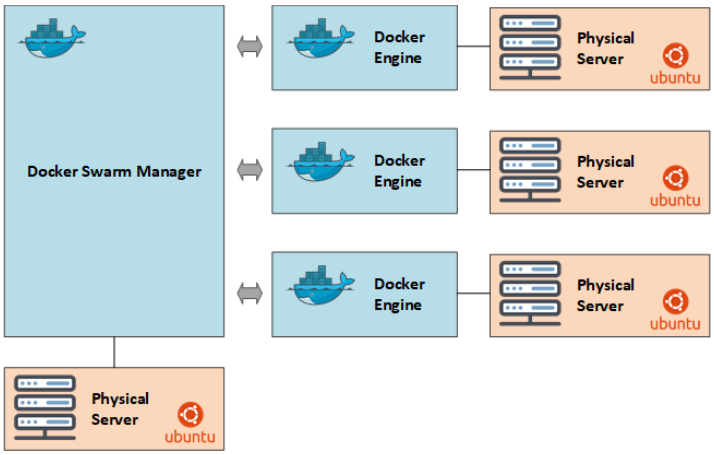


Figure 3 Use of Docker Swarm in COGITO's DTP

The primary responsibility of the orchestration layer is to deploy the components on the available physical computational nodes efficiently. This deployment method uses Docker, a lightweight virtualisation system that does not require Virtual Machine (VM) hypervisors running on hardware. The Docker images facilitate the portability and distribution of workloads in a standardised manner and allow developers to package the software components and dependencies into reusable and scalable units.

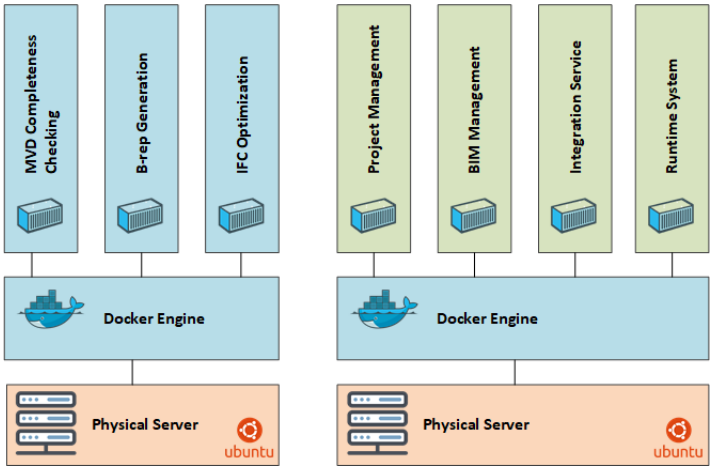


Figure 4 DTP's service orchestration

The distribution model of the DTP follows the Software as a Service (SaaS) approach, deployed on a private and containerised cloud computing environment hosted on dedicated physical servers. This approach ensures that the allocation of hardware resources can be centrally managed using simple instructions. As shown in Figure 4,

the Data Post-Processing Layer components require multiple running instances, while the Data Ingestion Layer and Data Management Layer components have fewer requirements.

3 Authentication Layer

The Authentication Layer is responsible for storing and managing user accounts and their roles, enabling the DTP to register and authenticate the users of the COGITO solution. It provides an Authentication and Authorisation Infrastructure (AAI), allowing the DTP to manage the access of the different stakeholders/users of the various COGITO software applications by assigning various functionalities such as registration, authentication, and authorisation to an external open-source identity provider.

3.1 Identity and Access Management

The AAI solution used in the DTP relies on the open-source identity and access management solution named Keycloak[2]. This solution is an industry-standard implementation for identity and access management supporting various protocols such as OpenID Connect and SAML 2.0. The OpenID Connect protocol enables Single Sign-On (SSO) and cross-domain identity management. Keycloak offers a REST API to handle the authentication protocol requests and a Graphical User Interface (GUI) to facilitate user registration, login, profile management and administrative operations.

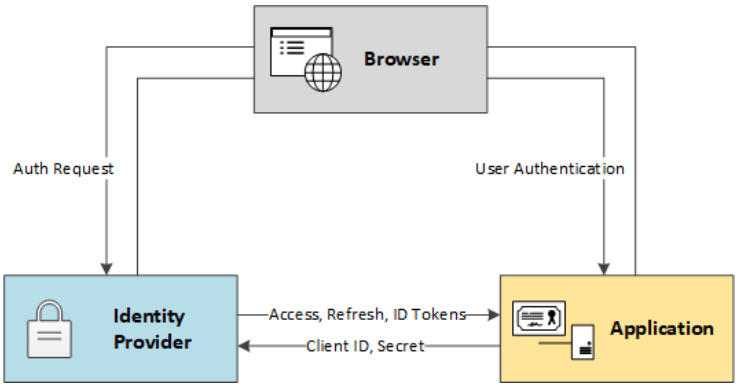


Figure 5 User authentication process in COGITO solution

Figure 5 shows the authentication process using the OpenID Connect protocol. Keycloak is the central system for managing profiles and the overall authentication process. The browser is used for the communication between COGITO's web-enabled applications and Keycloak. In case of a protected resource or endpoint request, the COGITO application forwards the user session to Keycloak's login page for the authentication request. Then, the COGITO application identifies itself using the Client ID and a secret key is generated from the Keycloak Admin Console. After successful authentication, Keycloak provides the Access, Refresh, and ID tokens to the COGITO application. The application can use these tokens to retrieve the assigned roles of the authenticated user and apply access policies.

3.2 User Roles

Within Task “T2.1 Elicitation of Stakeholder Requirements” and its primary outcome “D2.1 Analysis of Digital Tools Market and Prevailing Regulatory Frameworks,” the main stakeholders of the COGITO project have been identified following a User-Driven Innovation (UDI) methodology [5]. These stakeholders are the end-users of the various COGITO applications. Thus, the identified roles have been introduced in the Authentication Layer using the Keycloak Admin Console. Core roles that have been added to the Keycloak database are the following: Project Manager, Site Manager, Quantity Surveyor, Foreman, Worker, Quality Manager, Surveyor, Health Safety and Environment (HSE) Manager, HSE Supervisor and HSE Trainer.

When a new user registers to COGITO`s DTP, the Input Data Management component provides access to the GUI for project creation and assignment of the user into existing projects.

[2] Keycloak Identity and Access Management https://www.keycloak.org

3.3 Interface Specification

As mentioned in the previous section, Keycloak is a central management system for managing user-profiles and the user authentication process. It provides two primary endpoints to interact with the COGITO applications: a) the Keycloak Authentication API granting access to users based on user credentials such as username and password and b) the Keycloak Admin REST API allowing administrators to access all features provided by the Admin Console GUI.

The COGITO applications use the Authentication API for authenticating users. The authentication process has the following steps: First, the COGITO application redirects the user to Keycloak for performing the authentication process. If the authentication is successful, the Keycloak redirects the user back to the COGITO application. Next, the COGITO application performs a second request for retrieving the Access, ID, and Refresh Token. The endpoints along with their descriptions of the Authentication API are listed in Table 1.

Table 1 Identity Provider's Authentication API specification

Endpoint Name	Protocol	Method	Endpoint Description
OpenID	HTTPS	GET	It provides the main configuration parameters of the authentication server. The response is a JSON object which includes all available endpoints, scopes, and signing algorithms.
Authorisation	HTTPS	GET	It is the endpoint of the authorisation server and is used to retrieve an authorization code which is included as a parameter in the redirected link after a successful login.
Authentication	HTTPS	GET	It is the endpoint of the authentication server and is used by the COGITO application to request the Access, ID and Refresh Tokens.

On the other hand, the Keycloak Admin REST API is used by some COGITO applications and admin users for interacting with the Keycloak backend. It supports various requests such as getting roles, getting users, getting users with a specific role, and assigning a role to a user. The endpoints along with their descriptions of the Admin API are listed in Table 2.

Table 2 Identity Provider's Admin API specification

Name	Protocol	Method	Endpoint Description
Get users	HTTPS	GET	Get all users of the COGITO realm
Get roles	HTTPS	GET	Get all roles of the COGITO realm
Get user's roles	HTTPS	GET	Get COGITO's role mappings
Assign a role to a user	HTTPS	POST	Add COGITO's role mappings to a user
Remove a role from a user	HTTPS	DELETE	Remove COGITO's role mappings from a user

4 Data Ingestion Layer

The Data Ingestion Layer includes software components responsible for project creation, BIM data consistency validation, knowledge graph generation and database population. The software components can be grouped into three operational blocks based on their characteristics and functionalities: i) Input Data Management, ii) BIM Management, and iii) Knowledge Graph Generator. These blocks include components, which can act as standalone services. Figure 6 displays a generalised view of the architecture of the Data Ingestion Layer with its three blocks and their members.

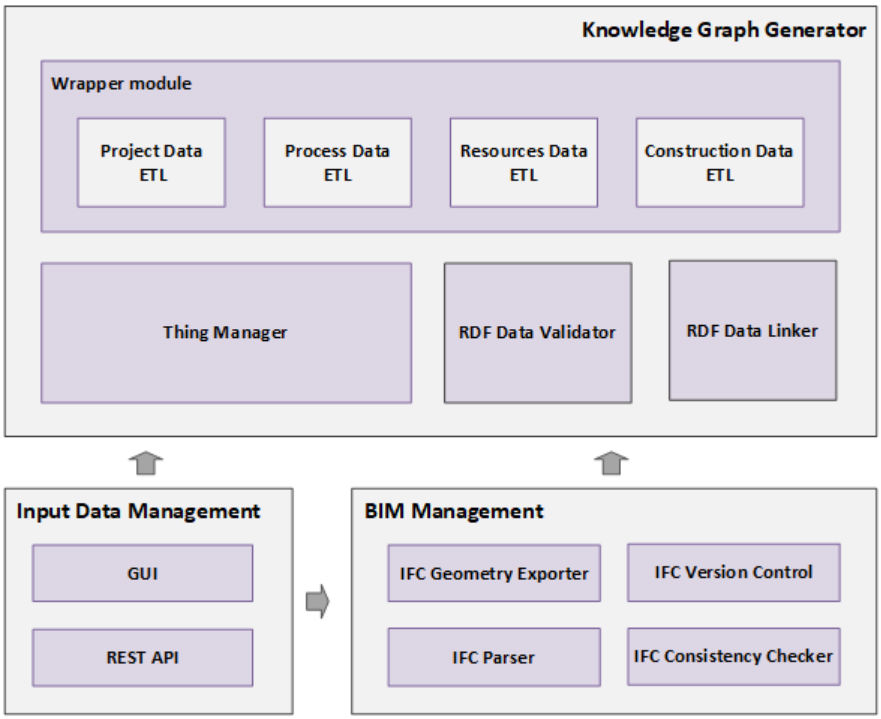


Figure 6 Data Ingestion Layer architecture

These three operational blocks and their respective components are described in detail in the following subsections.

4.1 Input Data Management

The Input Data Management component provides a standalone web-based application responsible for creating a new project, assigning users, and loading data from the other COGITO applications. It is responsible for supervising the running services and informing the COGITO users about the progress and the corresponding results.

4.1.1 Architecture

The Input Data Management component is deployed as a Spring Boot application and is the primary input interface of the DTP. Spring Boot is built on top of the Spring Framework, enabling an easy way to set up, configure and run services and web-based applications. It provides an API and includes several indicators to inspect the health of the running processes, memory usage, error logging and more.

The proposed software implementation uses modern web technologies to deliver a rich Graphical User Interface (GUI) for user interaction and configuration, including web-sockets to synchronise the backend services with the front-end elements. It uses the Apache Tomcat web server and software packages provided by the BIM Management component to load, handle, and visualise IFC data.

The Input Data Management component provides authorised access to COGITO users and other COGITO applications through the Authentication Layer. The web application is based on the Model-View-Control (MVC) design pattern and uses web-sockets implementing the Streaming Text Oriented Messaging Protocol (STOMP). Figure 7 shows the interactions of the main components involved in the GUI and REST API backend for the element.

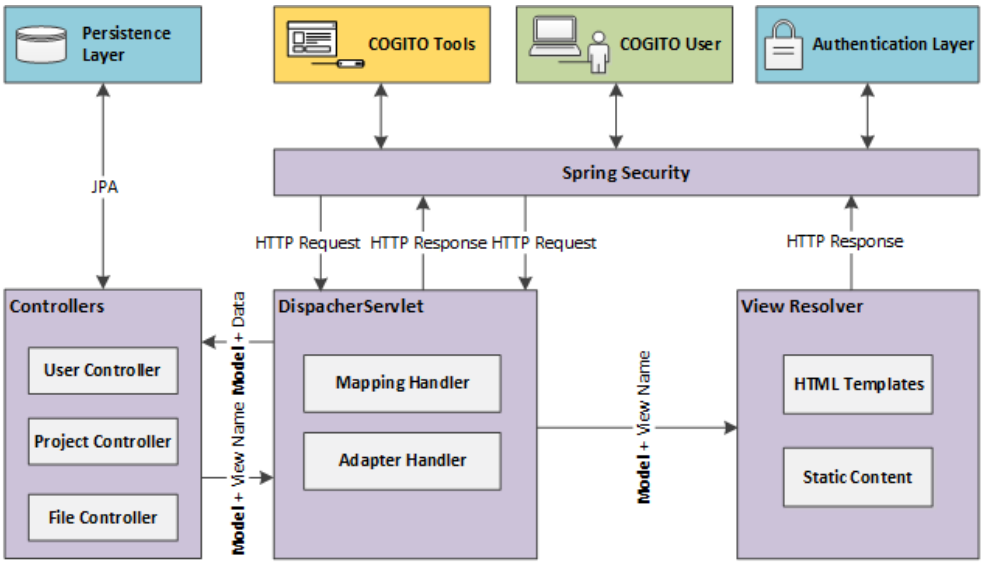


Figure 7 Backend architecture of Input Data Management component

As shown in Figure 7, the Spring MVC framework comprises four main parts:

- The **Dispatcher Servlet** forwards incoming client requests to specific controllers based on the URL pattern configuration of the Mapping Handler.
- The **Model** contains the data of the request; the data can be a single object or a collection of objects.
- The **Controller** collects through an API or using Dependency Injection (DI) the data from the Persistence Layer and stores them in the Model object; and
- The **View Resolver** combines the Model object with the corresponding HTML template to render the page and then forwards the response back to the client.

The Input Data Management component uses the Spring Security Framework with the embedded Spring Keycloak Adapter to manage the access policies of the COGITO users. Spring Security uses the tokens provided by the Authentication Layer to grant access to users and other COGITO applications for accessing protected data through the REST API.

As mentioned previously, the backend implementation of the Input Data Management component uses a relational database located in the Persistence Layer for storing all project data and meta-data. Moreover, it includes an AMQP adapter for enabling asynchronous communication with the Data Post-Processing Layer through the Messaging Layer. Figure 8 illustrates the stack diagram of the Input Data Management component that contains the following subcomponents:

- **Spring Core** is the foundation component of the Spring Framework. It supports application development using Dependency Injection (DI) and Inversion of Control (IoC).
- **Spring JDBC** is a component of the Spring Framework responsible for connecting to the Project Database and executing SQL queries. Spring JDBC provides an abstraction for handling database connections, preparing SQL statements, and handling potential exceptions.
- **Spring Java Persistence API (JPA)** is a component of Spring Framework that automates the creation and population of relational databases using the Object-Relational Mapping (ORM) specification. It supports the Create, Read Update and Delete (CRUD) operations of the Data Access Objects (DAO) of the Input Data Management component.

- **Spring Security** is a component of Spring Framework that manages the authentication and authorisation operations of the Input Data Management component. The Keycloak Spring Adapter uses Spring Security to grant access to registered users and other COGITO applications.
- **Spring Model View Control (MVC)** provides a framework for creating web applications using the Model View Control design pattern. It supports the main concepts of modern web applications and offers extensibility through the integration of external frameworks.
- **Thymeleaf** is a template engine used for implementing the front-end components, which constitutes the View part of the MVC design pattern.
- **Spring Java Messaging Service (JMS)** provides a framework for the asynchronous communication of the Input Data Management component with the message broker of the Messaging Layer using the AMQP protocol.

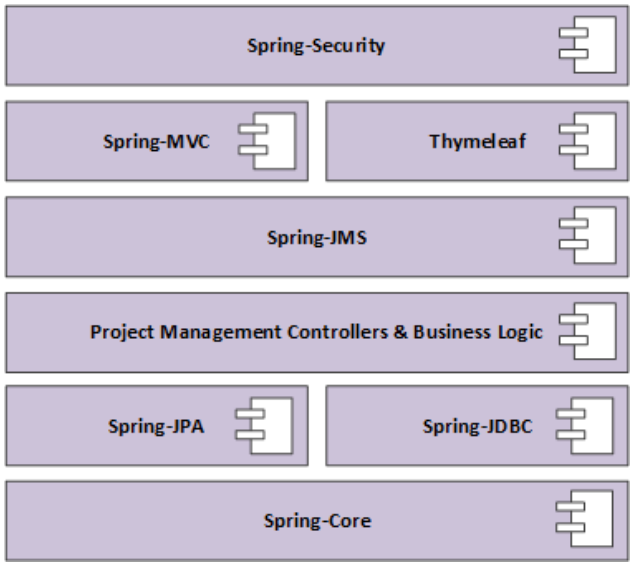


Figure 8 Stack diagram of Input Data Management component

It is worth mentioning that all involved technologies and frameworks to develop the Input Data Management component are based on open-source projects. The deliverable "D7.10 - Digital Twin Platform v2" reports the complete list of software frameworks used for the implementation with their version and licences.

4.1.2 Interface Specification

The Input Data Management component provides a REST API allowing the various COGITO applications to interact with the DTP for project creation, user management and more. Some of the fundamental functionalities provided are listed in Table 3. The detailed documentation of the Input Data Management REST API is provided in the deliverable “D7.10 - Digital Twin Platform v2”.

Table 3 Input Data Management’s API specification

Functionality	Protocol	Method	Endpoint Description
Get all projects	HTTPS	GET	Returns the projects of the DTP
Get a project	HTTPS	GET	Returns the meta-data of a project. The response includes attributes such as project id, name, description, location, etc
Get all users of a project	HTTPS	GET	Returns the users assigned to a project
Get all properties of a project	HTTPS	GET	Returns the properties of a project
Get a user	HTTPS	GET	Returns the meta-data of a user. The response includes attributes such as user id, first name, last name, email, etc
Get all users	HTTPS	GET	Returns the users registered in the DTP
Get all roles of a user	HTTPS	GET	Returns the roles assigned to a user
Get all projects for a user	HTTPS	GET	Returns the projects of a user

4.2 BIM Management

The BIM Management component is responsible for handling BIM data that conform to the Industry Foundation Classes (IFC) standard. It contains low-level software packages for serialising/deserialising, querying, updating, and merging IFC data used by the DTP facilitating various data processing operations. For instance, the Persistence Layer uses the BIM Management component for loading the IFC objects into the key-value database. Furthermore, the Knowledge Graph Generator of the Data Ingestion Layer uses the BIM Management component for transforming the IFC objects to RDF instances. Finally, the Data Post-Processing Layer uses the BIM Management component to extract geometric information and execute application-specific algorithms that use the IFC objects as input.

4.2.1 Architecture

The BIM Management component is a Software Development Kit (SDK) providing a set of software libraries for serialising/deserialising, validating, querying, updating, and merging IFC data. These software libraries are widely used within the various layers of the DTP. For the implementation, we use Java with a minimum number of external dependencies.

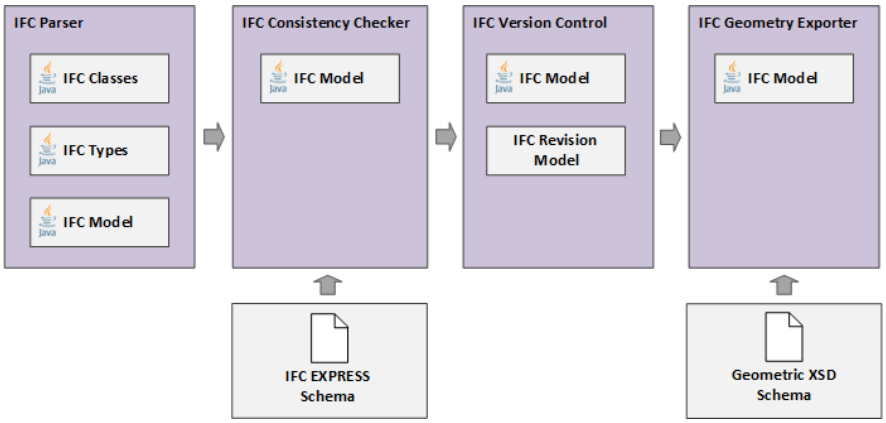


Figure 9 High-level architecture of BIM Management component

As shown in Figure 9, the BIM Management component contains four low-level software packages: i) **IFC Library** used for serialising/deserialising and querying IFC data, ii) **IFC Consistency Checker** used for validating the STEP data against the IFC schema, iii) **IFC Version Control** used for tracking the changes that might have occurred to the IFC data, and iv) **IFC Geometry Exporter** used for generating an XML file that contains the geometric representation of the building elements and their related coordinate system data.

4.2.2 EXPRESS Schema Compiler for Java

The EXPRESS Schema Compiler is a library developed in Java EE using the Java Code Model framework for generating the IFC Java classes and IFC types directly from the EXPRESS schema. The compiler can parse all available IFC releases successfully, from IFC2x3 to IFC4x3. Figure 10 illustrates how IFC classes and IFC types are generated automatically from the IFC EXPRESS schema.

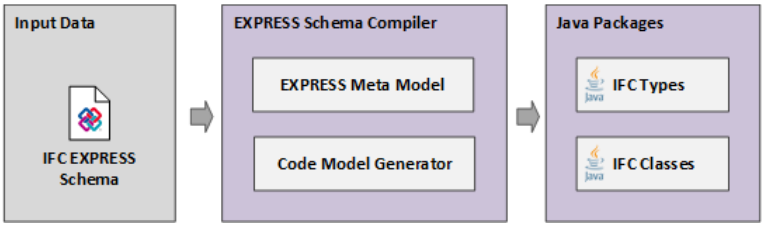


Figure 10 IFC Java Classes Generation using the EXPRESS Schema Compiler

Firstly, the compiler transforms the EXPRESS data into in-memory objects using an internal data model representation. Then, it applies a set of transformation rules to instantiate the corresponding Java Code Model objects. In the end, the Java Code Model framework generates the IFC classes and IFC types and classifies them based on the name of the IFC schema in different packages.

4.2.3 IFC Java Library

The IFC Java Library, aptly named IFC Library, uses the IFC classes and IFC types generated from the EXPRESS Schema Compiler to efficiently parse the STEP data and instantiate the representation of the BIM model in-memory. The current version of the implementation can handle the most frequently used IFC releases, from IFC2x3 to IFC4x3.

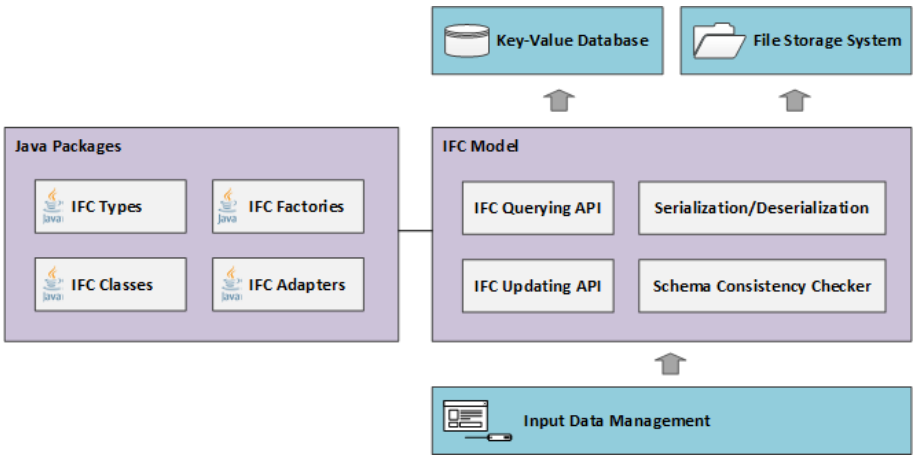


Figure 11 IFC Implementation for Java

As shown in Figure 11, the IFC Library provides an API that offers useful functionalities for handling the loaded objects. It supports an API for querying and updating the IFC objects and some advanced features, such as initialising the inverse relations by adding an object to the corresponding collection of the inverse connected instance of another entity. The initialisation of the inverse relations in the objects is automatically archived by calling the *inverse()* method, as shown in Table 4.

Table 4 Example of inverse relations provided by the IFC Library

IfcElement

```
public IfcElement(){
   ...
   this.hasOpenings = new ArrayList<IfcRelVoidsElement>();
}

public void inverse(){
}

public List<IfcRelVoidsElement> getHasOpenings() {
   return this.hasOpenings;
}
```

IfcRelVoidsElement

```
public IfcRelVoidsElement(){
}

public void inverse(){
   if(this.element != null && this.element.getHasOpenings() != null){
      this.element.getHasOpenings().add(this);
   }
}
```

In the above example, using the *inverse()* method, it is possible to retrieve the *IfcRelVoidsElement* object from the inverse method *getHasOpenings()* of the *IfcElement* object. This functionality is widely used by the MVD Completeness Checking component.

4.2.4 IFC Consistency Checker

Within the COGITO solution, building information exchange among the different components is based on the openBIM standard IFC ISO 16739:2018. In IFC standard, the data are stored in ASCII form using the STEP format, the structure of which is defined according to the corresponding EXPRESS schema provided by buildingSMART International. The IFC Library includes a schema compliance checker for validating the STEP data against the EXPRESS schema of the standard. The schema compliance can perform validation across various datatypes, classes and restrictions in numerical values and collections.

4.2.5 IFC Geometry Exporter

The efficient processing of the IFC data consists of separate operations performed by individual libraries written in different programming languages. The deserialisation of IFC is easier using high-level programming languages such as Java or C#, while geometric operations are more efficient in low-level languages such as C or C++. As shown in Figure 12, the IFC Geometry Exporter component uses the IFC Library to parse the IFC data and generate an XML file that contains the geometric representation of the building elements and their related local coordinate system data.

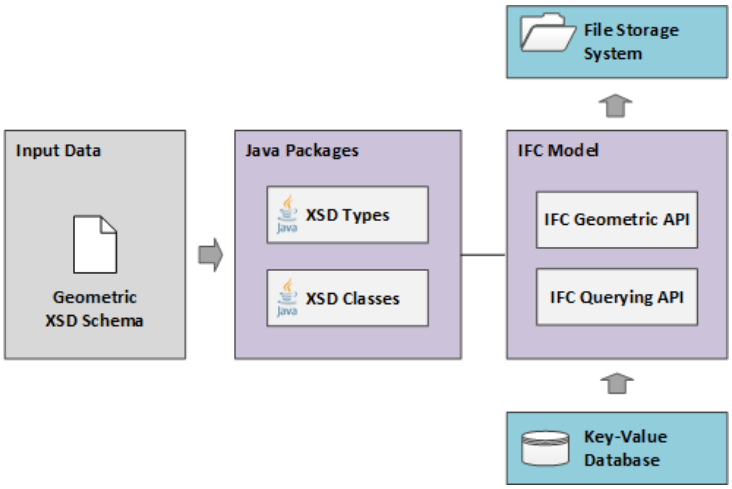


Figure 12 IFC Geometry Exporter component

The generated XML conforms to an XSD schema which follows the geometric subset of the IFC4 Design Transfer View (DTV) specification, including the geometric extensions for IFC4x3.

4.2.6 IFC Revision Control

The BIM Management component can handle both the geometric and the semantic information included in the IFC data. In the IFC schema specification, the objects may reflect a final state, but they also may reflect a transient state. For instance, the SafeConAI application identifies zones in the BIM model where specific types of hazards can occur [6]. After identifying the zones, the tool enhances the BIM model with safety information by revising the existing IFC objects.

In a scenario where multiple applications update the IFC model simultaneously, the IFC schema supports local copies of the modified objects. The included revision scheme identifies changes declared on a per-object basis instead of identifying changes in the text. An IFC object is considered as modified when: a) any of the direct attribute changes; b) any referenced resources change; and c) items are added or removed from any collection. Each IFC object is marked with a change action within this revision scheme, indicating if the IFC object was added, modified, deleted, or not changed.

The COGITO applications can use this revision scheme to track the changes that might have occurred to the IFC objects during the last active session. In this case, the BIM Management component will know how various COGITO applications affect an IFC object. Figure 13 illustrates a scenario where various external tools update the original IFC. The BIM Management component is responsible for filtering and merging the updated IFC objects according to the latest changes.

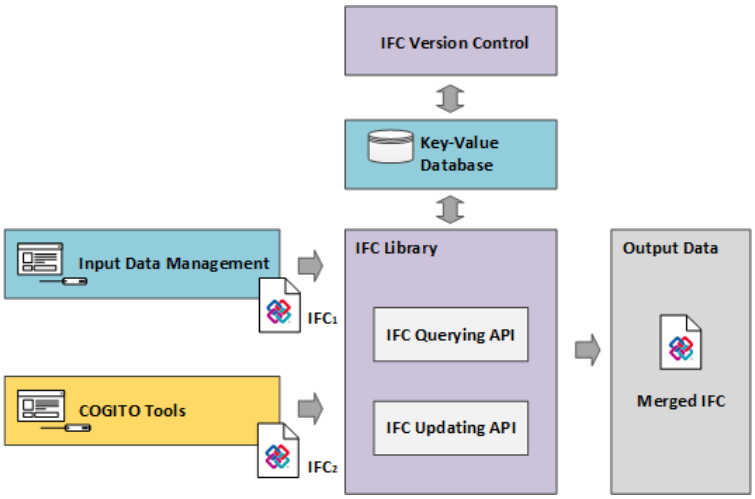


Figure 13 IFC Revision Control component

The process is as follows: First, the BIM authoring tool creates an IFC open for modifications. The IFC exporter should set the ChangeAction attribute of the IfcOwnerHistory to NOCHANGE to establish a baseline. With this annotation model, the BIM Management component can identify the upcoming modifications. Next, when a COGITO application updates the information of an existing IFC object, it should set the ChangeAction to MODIFIED and the OwningApplication to the application identifier. On the other hand, when it adds or deletes an IFC object, it should set the ChangeAction to ADDED or DELETED accordingly.

Moreover, the COGITO applications are responsible for updating the LastModifiedDate attribute to the time of modification. Thus, when the BIM Management component receives modified IFC data, it can determine which objects have been added, modified, and deleted and either merge or reject these changes, as necessary.

4.3 Knowledge Graph Generator

The Knowledge Graph Generator (KGG) is responsible for generating, validating, and storing the RDF data and the corresponding Thing Descriptions. As shown in Figure 14, it supports the transformation of heterogeneous data payloads coming from different domains used by the COGITO project. The implementation of the various ETL tools is based on the type of the input data, the mapping rules, and the corresponding COGITO ontologies. For instance, the Construction ETL consumes the BIM model (IFC) and generates the corresponding RDF data based on the COGITO Construction ontology. Similarly, the Process ETL consumes the construction schedule data (XML, CSV) and generates the corresponding RDF data based on the COGITO Process ontology. The generated RDF data will be validated and merged into a unified knowledge graph which will conform to the semantic representation of the construction digital twin. This semantic representation will be the main provider of information from which the different COGITO applications can consume and exchange data.

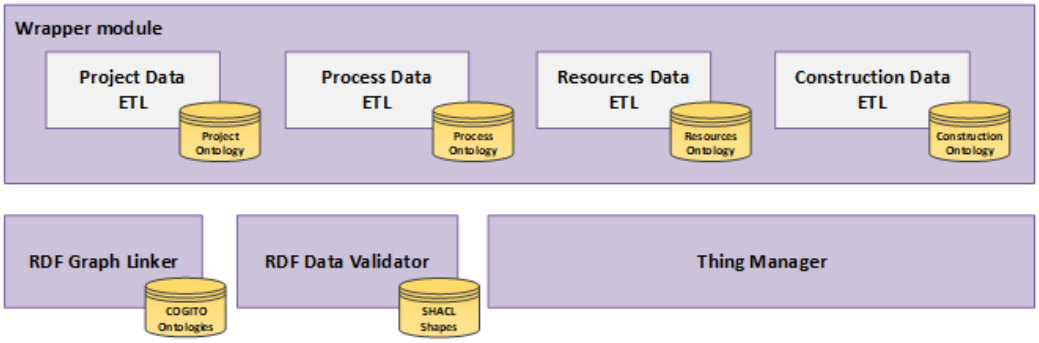


Figure 14 Knowledge Graph Generator's core components

4.3.1 Architecture

As shown in Figure 15, the high-level architecture of the KGG component consists of the following components: i) the **Thing Manager** is responsible for orchestrating the execution of the various data transformation processes, validating their outputs, and storing the RDF data and the corresponding Thing Descriptions in the Persistence Layer, ii) the **Wrapper module** is responsible for performing data pre-processing operations and invoking the various ETL tools for generating the corresponding RDF data, iii) the **RDF Graph Linker** is responsible for creating additional semantics on the RDF data, and iv) the **RDF Data Validator** is responsible for validating the individual RDF outputs and the unified knowledge graph.

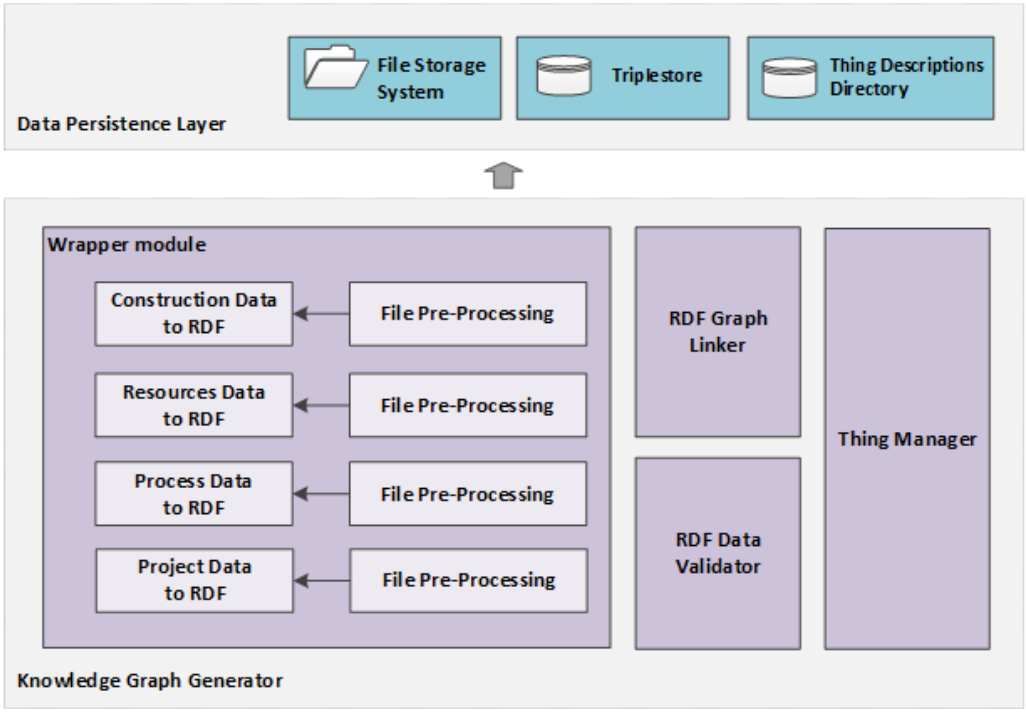


Figure 15 High-level architecture of the Knowledge Graph Generator

As mentioned previously, the main objectives of the KGG component are the creation and validation of the knowledge graph and the generation of the corresponding Thing Descriptions. Figure 16 shows the interactions of the involved components. Once the input files are loaded and stored to the Persistence Layer, the Input Data Management component invokes the KGG to orchestrate the execution of various components. Next, the process carried out is as follows:

- The Wrapper module retrieves the input data files from the Persistence Layer **(1)**, manages the executions the various ETL tools for generating **(2)** and validating **(3)** the RDF files.

- The Thing Manager receives the individual RDF files **(4)** and invokes the RDF Graph Linker component **(5)** to perform the linking process.
- Once the RDF data are linked, the merged dataset is forwarded to the RDF Data Validator **(6)** for validating the additional semantics.
- Finally, the Thing Manager stores the RDF data in the Triplestore **(7)** and the corresponding Thing Descriptions in the TDD **(8)**.

The communication between the components will be established utilising a dedicated network and internal APIs. The Thing Manager will provide the only external interface from the KGG component to the other layers of the DTP.

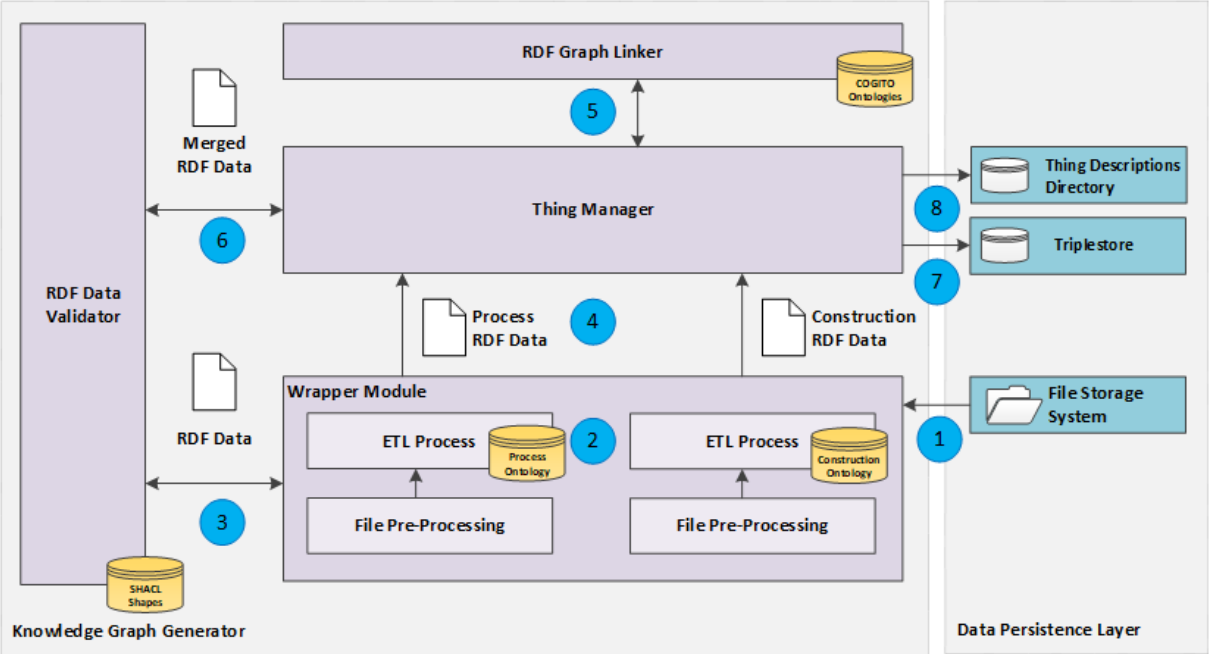


Figure 16 Knowledge graph generation and validation process

The services are deployed independently as containerised backend applications using the Docker compose technology. The Flask micro framework with the Gunicorn HTTP WSGI server for handling the requests in production. The main programming language used for the development of the component is Python. In the following sub sections, we will provide further details for each subcomponent.

4.3.2 Thing Manager

The Thing Manager is responsible for orchestrating the flow of information inside the KGG component. The KGG component gets invoked externally when the RDF graph needs to be generated. Predefined configurations or the request parameters will activate the corresponding services to store or retrieve data correctly. The Thing Manager is also in charge of creating the Thing Descriptions for the different data resources needed. Those data resources represent the various components of the construction site, such as workers, equipment, machinery, spaces, building elements, etc. We follow the WoT Thing Description specification for their representation, which provides a set of standard metadata for defining those objects. Figure 17 provides a small example of the Thing Description used to model a location tracking device.

```
{
      "id": ""id": "urn:dev:wot:com:example:servient:tracking-device",
      "name": ""name": "MyTrackingDevice",
      "security": [{"scheme": "basic"}],"security": [{"scheme": "basic"}],
      "properties": {"properties": {
             "status": {"status": {":{
                    "type": "string",
                    "forms": [{""forms": [{"href": "coaps://dt.cogito.io/device/status"}]
```

```
                }},
                "battery": {"battery": {":{
                        "type": "integer",
                        "forms": [{"forms": [{"href": "coaps://dt.cogito.io/device/battery"}]
                }},
                "altitude": {"altitude": {":{
                        "type": "real",
                        "forms": [{"forms": [{"href": "coaps://dt.cogito.io/device/altitude"}]
                }},
                "longitude": {"longitude": {":{
                        "type": "real",
                        "forms": [{"forms": [{"href": "coaps://dt.cogito.io/device/longitude"}]
                }},
                "latitude": {"latitude": {":{
                        "type": "real",
                        "forms": [{"forms": [{"href": "coaps://dt.cogito.io/device/latitude”}]
                }}
        },},
        "actions": {"actions": {
                "alarm-on": {"alarm-on" : {
                "forms": [{"forms": [{"href": "coaps://dt.cogito.io/device/alarm/on"}]
                }},
                "alarm-off": {"alarm-off" : {
                "forms": [{"forms": [{href": "coaps://dt.cogito.io/device/alarm/off"}]
                }}
        },},
        "events":{"events":{": {
                "low-battery":{"low-battery":{
                        "type": "boolean",
                        "forms": [{"forms": [{
                                "href": "coaps://dt.cogito.io/device/low-battery",
                                "subProtocol": "LongPoll"
                        }]}]
                }}
        }}
}
```

Figure 17 Location tracking Thing Description example

The Web of Things (WoT) Thing Description (TD) gives information about the different ways we can interact with the resources. Those interactions, called in the specification interaction affordances, are classified into three categories:

- **Property Affordance:** Provides information about certain internal states or properties of the thing we are modelling. The states can be read-only or writable. For instance, we can retrieve the device's current location by reading the properties of longitude, latitude, and altitude.
- **Action Affordance:** This allows the manipulation of the state of the thing or the triggering of an internal process. Examples of this type of interaction include the on or off actions of a sound alarm of a wearable location tracking device.
- **Event Affordance:** This type of interaction affordance pushes subscribers' event data, such as low battery notification.

For each type of interaction, the TD provides information concerning the endpoints needed to interact with the real resource, the access method, the protocol, etc. For the specific case of the resources in a construction project, we could have the thing description of the workers on the construction site. Those TDs can provide endpoints to access the notification devices of the workers and alert them in case they are near a dangerous zone.

The Thing Manager will generate those TDs automatically from the data upload by the COGITO application based on the ontologies developed for each domain. The Thing Descriptions created will be stored in a special component called the Thing Descriptions Directory, which will perform the persistence of the TDs.

The Thing Manager is implemented as a backend service, provided as a containerised application using the Docker technology for its easy deployment. The main framework used for the implementation of the service has been the Flask micro-framework.

4.3.3 Wrapper Module

The Wrapper module is responsible for transforming heterogeneous data provided by the different COGITO applications into RDF data. The contained ETL tools ensure that the RDF data generated are aligned with the ontologies developed for each domain. The COGITO developers define the mappings files for each domain or application before proceeding with the execution of the respective transformation. We define the mapping rules to perform the data transformation using the RDF Mapping Language (RML). The processed data is then sent to the other services of the KGG to validate the data, perform the appropriate links to the existing graph, and store the data in the Triplestore of the Data Persistence Layer.

In particular, the Wrapper module offers the following capabilities:

- Mapping of the ingested data to the respective COGITO ontologies: Mapping files will be created to align the concepts from the data provided by the applications and concepts from ontologies developed for the different domains of the project.
- Transformation of the ingested data to the respective COGITO ontology: Using the previously created RML mapping files, the ETL service derives the transformation rules needed to convert the original data into RDF graph data automatically.

The ETL tools will be implemented as backend services with two layers:

- The business logic layer containing the logic behind the execution of the transformation rules. The Flask micro web framework and Helio will be used for the development. Helio is a framework that allows the translation of data into RDF and the publication of RDF data as a Linked Data Service.
- The data access layer storing the mapping files and additional internal configuration files. This layer will utilise a Postgress database.

4.3.4 RDF Graph Linker

The RDF Graph Linker component will be responsible for linking different resources belonging to different RDF datasets. When two RDF resources refer to the same real-world entity, such as the same individual, or identical items, it is possible to establish a link reflecting this identity relationship. This way, heterogeneous data provided by the various COGITO applications can be linked, obtaining greater effectiveness. Potential advantages of the linked data approach include improved *precision* and *robustness* by cancelling possible errors in the data; *efficiency* in terms of time and space, minimising the search time between different resources; and *versatility*, applying to various datasets and domains. The positive benefits of this linked data approach will be evaluated in the context of the COGITO applications.

4.3.5 RDF Data Validator

The RDF Data Validator aims to detect, through the validation process, the different possible errors that may exist within the previously generated knowledge graph. A recent development, the Shapes Constraint Language (SHACL) can help achieve this.

SHACL is a language for validating RDF graphs against a set of requirements provided as shapes and other constructs expressed in the form of an RDF graph (shapes graphs). SHACL shape graphs are used to validate the data graph, and they can be a description of the data graphs that satisfy a set of conditions or requirements. RDF graphs that pass validated against a shapes graph are called "Application Profiles".

5 Data Persistence Layer

The Data Persistence Layer of the DTP contains different types of datastores for storing structured data. Based on the architecture specification of the platform, the Persistence Layer consists of i) a file storage system for storing files generated by other COGITO applications, ii) a relational database for storing project and user data, iii) a key-value database for storing IFC objects, iv) a time-series database for storing IoT sensor data, v) a triplestore for storing COGITO's unified knowledge graph, and vi) a Things Description Directory for storing COGITO's Thing Descriptions. The following subsections describe these datastores and their functionalities.

5.1 File Storage System

The File Storage system provides a file storage solution enabling other COGITO applications to access shared files. It supports file system semantics and a permission model applying access policies such as role-based access control on the registered users. The proposed software implementation includes a REST API for accessing and managing files stored on a DTP cloud provider.

5.2 Project Database

The Project database includes various tables and relationships representing a relational data model which stores the information required by the Input Data Management component. The tables related to users and user roles are populated and synchronised by the Identity Provider of the Authentication Layer, as shown in Figure 18.

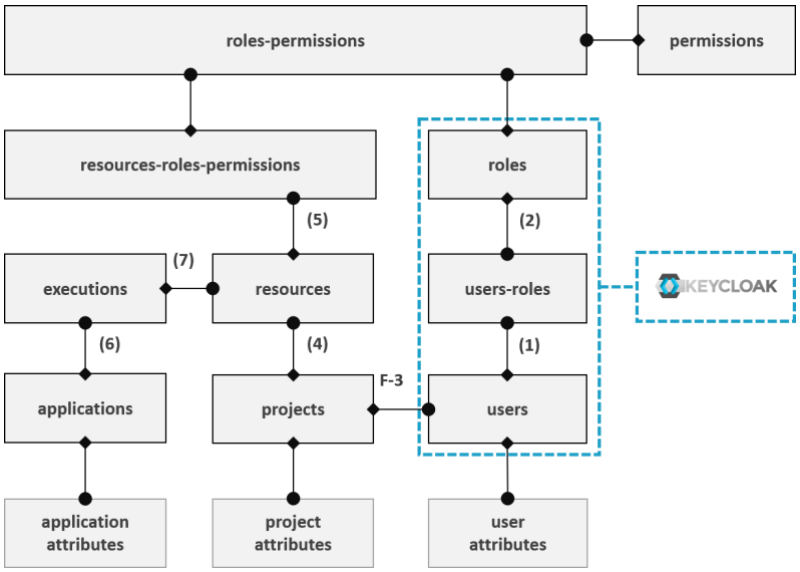


Figure 18 Relational data-model of the Input Data Management component

The GUI and the REST API use the information stored in the Project database and the relationships among the different tables to support core functionalities of the Data Ingestion Layer, which are described as follows:

1. Storing user accounts defined by Keycloak in the Authentication Layer.
2. Storing user roles defined by Keycloak in the Authentication Layer.
3. Assigning user accounts to a specific COGITO project.
4. Storing meta-data of the uploaded files and assigning them to a specific COGITO project.
5. Applying access policies to the uploaded files using a permission scheme connected with the user roles.
6. Monitoring the various COGITO applications and the progress of their executions.
7. Applying access-policies on the output resources of the executions of the various COGITO applications.

The Input Data Management component uses the Spring Java Persistence API (JPA) and the Hibernate framework for automatic deployment of the Project database. In this context, Java classes represent the tables, while the fields inside the classes represent properties and the relations between different tables. The Spring JPA framework supports all types of relations, such as one-to-one, many-to-one, one-to-many and many-to-many. When using this approach, the relational database can bet transparently managed from Java, increasing the abstraction level of the persistence layer.

The Input Data Management requires a connection to a MySQL server. MySQL server is a Relational Database Management Systems (RDBMS) that supports multi-tenancy. The Hibernate framework utilises the MySQL dialect to access the Project database for performing transactional operations and queries. The Spring JPA entirely manages the generation and execution of the SQL queries used mainly for providing the necessary information to the GUI and the REST API.

5.3 Key-Value Database

As mentioned in a previous section, the IFC Library provides an object-oriented representation of the IFC model. The STEP-data are parsed and loaded in memory using the IFC Java classes generated by the EXPRESS Schema Compiler. Based on their scope, the generated IFC Java classes implement various interfaces to achieve the desired level of abstraction. The IFC EXPRESS schema contains a set of different data types for storing IFC data described as follows:

- **ENTITY** is equivalent to the Java class. It can be defined as an abstract or concrete class, including attributes and implementing various interfaces.
- **ENUMERATED** is equivalent to the Java Enum. It represents a group of unchangeable STRING values.
- **SELECT** is equivalent to the Java interface. It defines a choice or an alternative between different options.
- **SIMPLE** is equivalent to the Java basic data types such as Integer, Double, Boolean and String.

The IFC Library defines a set of high-level interfaces to handle the above data types, as shown in Figure 19.

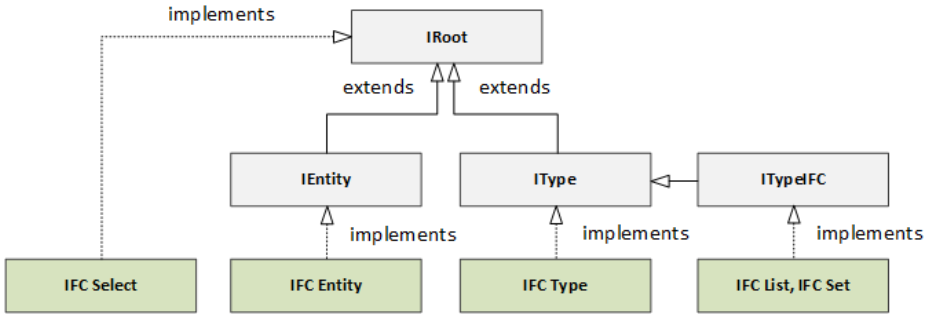


Figure 19 High-level interfaces defined in the IFC Library

The IFC parser of the BIM Management component includes a HashMap collection to store the key-value pairs of the loaded IFC data in memory. Each key contains the EXPRESS Id, and the corresponding value contains the IFC object that implements the IFC Library's high-level interfaces. After parsing the IFC, the BIM Management component stores the collection of the key-value pairs in Redis DB. Redis DB provides a distributed and high-performance key-value database system that offers additional functionalities than the Java HashMap collection, such as remote instances, persistence, concurrent read/write and more.

5.4 Timeseries Database

For storing location tracking data coming from sensorial IoT devices installed on workers and machinery in the construction site, the Persistence Layer of the DTP contains a time-series database based on InfluxDB. This database is optimised for storing high-volume data produced from various IoT devices. The IoT Data Pre-Processing tool feeds the DTP with timestamped location tracking data identified by a unique Tag ID. On the

other hand, the Data Management Layer contains the DT Runtime component responsible for hosting various data-driven modules. Among others, it hosts the IoT Data Logger module, which populates the time-series database using event-based data compression, and the IoT Data Retriever module, which queries the time-series database and applies linear interpolation on the returned time-series data. For instance, when the Process Modeling and Simulation (PMS) requests time-series data for specific resource instances, the DT Runtime component performs SPARQL queries to the knowledge graph to retrieve the corresponding Tags' IDs. Then, it performs a second request using the IoT Data Retriever module to retrieve the interpolated time-series data.

5.5 Graph Database

The Graph Database will store the RDF data generated by the Knowledge Graph Generation component. The RDF data will be stored under different namespaces (graphs) on different domains. The graph database will include the graph network representing the various aspects of the construction digital twin. The Graph database will support read, update, write and delete operations through its SPARQL endpoint. This endpoint will not be accessed directly by the applications or by the Thing Manager performing predefined SPARQL queries. However, this data will be linked utilising common concepts in the models. The triple store is deployed independently as a Docker container and will be implemented using the GraphDB[3] database.

5.6 Thing Description Directory

The Thing Description Directory is a persistence service that contains the Thing Descriptions (TD) created by the Thing Manager component in the Knowledge Graph Generator. The directory will support the discovery, creation, retrieval, update, and deletion of TD's. The Thing directory uses the WoT Hive implementation, compliant with the W3C Web of Things Directory standard specification. The RDF4J triplestore will be used.

[3]GraphDB https://graphdb.ontotext.com/

6 Data Management Layer

The Data Management Layer satisfies the data needs of the COGITO applications by offering an actor-based runtime system and a web-based application for configuring the endpoints and the various data-driven modules. The final design of the Data Management Layer contains the following components:

- The **DT Runtime** component facilitates the COGITO developers to configure: i) dynamic endpoints allowing COGITO applications to interact synchronously or asynchronously with the DTP and ii) data-driven modules for performing various data processing and routing operations. The DT Runtime component is responsible for hosting and executing the data-driven modules. It contains a few message adapters for sending notifications to the COGITO applications through the Messaging Layer utilising the AMQL, MQTT and STOMP protocols.
- The **DT Library** component provides a set of ready-made actors allowing COGITO's developers to create custom data-driven modules using the available actors.

Figure 20 shows the main interactions between the core components included in the Data Management Layer and the COGITO applications.

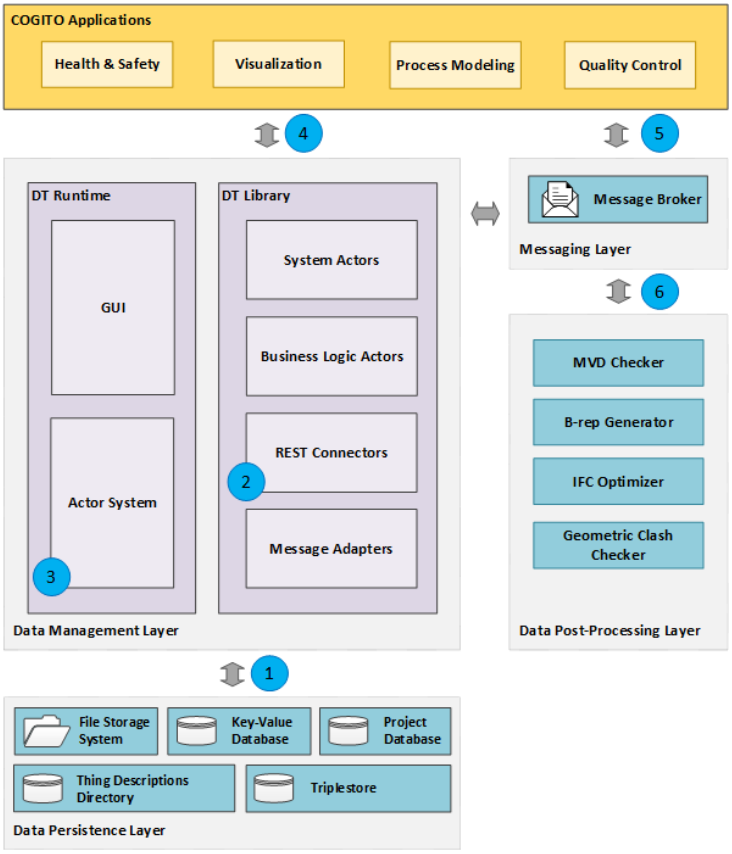


Figure 20 Data Management Layer's core components interactions

The Data Management Layer performs and supervises data query operations, ensuring that data have been correctly retrieved from the Data Persistence Layer **(1)** and efficiently delivered to their destinations. It provides a set of ready-made actors for abstracting the interfaces between the databases of the Persistence Layer and the COGITO applications **(2).** The actor-based runtime system **(3)** supervises the execution of the configured data-driven modules, which perform various data processing operations, ensuring that the data coming from the different databases are synchronised correctly before being forwarded to the data consumers through the available interfaces **(4,5)**. In addition, some data-driven modules often use the Messaging Layer **(6)** for invoking various components deployed in the Data Post-Processing Layer to perform MVD checking, IFC optimisation, B-rep geometry exportation and geometric clash detection.

6.1 DT Library

For interacting with the Persistence Layer and the Data Post-Processing Layer, the Data Management Layer contains a set of ready-made actors which encapsulate routines to facilitate the reusability and minimise the relevance of the expertise in the development of the required modules. The final list of DT Library's packages is presented in Table 5.

Table 5 DT Library's actor packages

Name	Description
DTP's system actors	This package contains actors implementing the message translations, message routers, data filters, etc.
DTP's business logic actors	This package contains actors implementing various data processing operations for supporting the smooth running of the COGITO system.
REST connectors	This package contains actors implementing various clients for interacting with the databases of the Data Persistence Layer.
Message adapters	This package contains actors implementing the various subscribers and producers for interacting with the massage broker of the Messaging Layer.

6.2 DT Runtime

The DT Runtime component offers a web-based application to configure DTP's modules and the dynamic endpoints used to interact with the other COGITO tools. The web-based application provides three main functionalities: i) registration of the COGITO tools and configuration of their access policies, endpoints, and notification channels, ii) installation of new actors using external JAR libraries, and iii) creation of the data-driven modules and configuration of their parameters. In addition, the DT Runtime component provides a REST API allowing the COGITO tools to manage their data collections for each configured endpoint.

The actor-based runtime system is a lightweight container for hosting data-driven modules deployed to synchronise data responses from the Persistence Layer and harmonise the data before being delivered to the final destinations. It is based on the open-source project Akka[4] which provides a toolkit and a runtime environment for simplifying the construction of concurrent and distributed applications. In other words, Akka is a powerful reactive high-performance framework optimised for running on the Java Virtual Machine (JVM) that can handle multiple queries simultaneously and respond to COGITO applications through the available adapters.

6.2.1 Architecture

The proposed architecture includes several packages of ready-made coding blocks which implement COGITO's business logic operations using a system abstraction provided by the DT Library. This solution implements a conceptual model for concurrent asynchronous data requests and enables communication between primitive software units named actors. An actor is an extensible program-code template that uses an abstraction for interacting with the Persistence Layer, executes its business-logic operations and forwards the results to other actors by producing asynchronous messages.

[4] Akka Actor Model https://www.akka.io

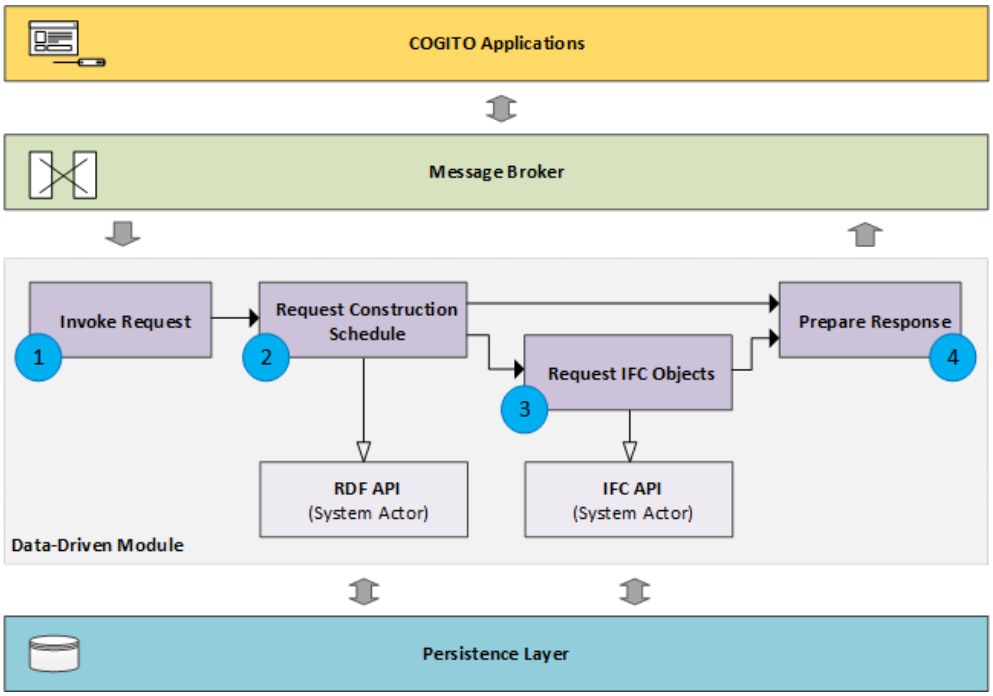


Figure 21 Example of a module deployment in the DT Runtime component

By way of example, see Figure 21. The involved actors are wired together in a sequence for handling the complex request of querying 4D BIM data from the DTP. First, a system actor **(1)** handles the incoming request and triggers the executions of the following actors **(2,3)** for retrieving the construction schedule and the corresponding IFC objects. These actors use the abstraction provided by the provided system actors for interacting with the Data Persistence Layer. Next, the last actor **(4)** is responsible for synchronising, merging, and delivering the results to the correct destination through the Messaging Layer. This is one way of creating responses to complex queries. Such actors can be created and re-used depending on application needs and information requirements.

6.3 Interface Specification

Based on the work performed in "T2.4 COGITO System Architecture Design" and the corresponding deliverable "D2.5 COGITO System Architecture v2", the main data exchange requirements between the DTP and the various COGITO applications have been identified [6]. Within COGITO, the multiple tools can be categorised as follows:

- **Data Pre-Processing** tools are responsible for a) pre-processing raw visual and location tracking data, b) annotating the processed data, and c) storing the data into DTP's Persistence Layer.
- **Health and Safety** tools are responsible for generating hazards mitigation measures and producing warning notifications to the on-site crew for their proximity to hazardous areas.
- **Workflow Modeling and Simulation** tools are responsible for monitoring and optimising the construction processes.
- **Quality Control** tools are responsible for comparing the as-designed and as-built data and detecting potential defects.
- **Visualisation** tools are responsible for retrieving data from the DTP and visualising it to support on-site and off-site activities of relevant stakeholders and training of the workers.

As shown in Table 6, the COGITO applications with their high-level data exchange requirements are grouped into the identified categories.

Table 6 Data exchange requirements of COGITO applications

Categories	COGITO Tools	Input Data	Output Data

Data Pre-processing tools	IoT Data Pre-processing	-	▪ Timestamped measurements of location tracking devices (**longitude, latitude, altitude, tags**)
	Visual Data Pre-processing	▪ Authentication Tokens ▪ Semantically linked data (**geometric representations of building elements, building element objects, task objects**) filtered by (**camera location, time range**)	▪ Annotated imagery datasets. (**building element objects, images**) ▪ Annotated point clouds. (**building element objects, point cloud datasets**)
Health and Safety	VirtualSafety	▪ Semantically linked data (**geometric representations of building elements, building element objects, geometric representations of H&S elements, H&S elements, tasks**) filtered by (**time range**)	▪ Training KPIs for workers (**actual resource objects, performance metrics objects**)
	SafeConAI	▪ Authentication Tokens ▪ Semantically linked data (**geometric representations of building elements, building element objects, task objects**) filtered by (**time range**)	▪ Semantically linked data (**geometric representations of H&S elements, H&S elements**)
	ProActiveSafety	▪ Semantically linked data (**geometric representations of building elements, building element objects, geometric representations of H&S elements, H&S elements, tasks**) filtered by (**time range**) ▪ Timestamped measurements of actual resources (**longitude, latitude, altitude, actual resource objects**)	▪ Notification (**actual resource objects, safety warning objects**)
	WODM	▪ Authentication Tokens ▪ Semantically linked (**actual resource objects, tags**) ▪ Semantically linked data (**as-planned resources ,task objects, building element objects**)	▪ Semantically linked data (**actual resource objects, task objects, building element objects**)
Workflow Modeling and Simulation	WOEA	▪ Authentication Tokens	-
	PMS	▪ Authentication Tokens ▪ Initial data (**as-planned resources**) ▪ Initial linked data (**building element objects, construction schedule**) ▪ Timestamped measurements of actual resources. (**longitude, latitude, altitude, actual resource objects**) ▪ Semantically linked data (**geometry quality control objects, building element objects**) ▪ Semantically linked data (**visual defect objects, building element objects, images**) ▪ Semantically linked data (**geometric representations of H&S elements, H&S elements**) ▪ Defect Notification (**building element objects, images, type of remedial works**) ▪ Hazard Notification (**building element objects, images, type of mitigation works**)	▪ Semantically linked data (**as-planned resources, task objects, tag objects, building element objects**) ▪ Semantically linked data (**actual resource objects, task objects, tag objects, building element objects**)
Quality Control	Geometry QC	▪ Semantically linked data (**geometric representations of building elements, annotated point clouds**) filtered by (**time range**)	▪ Semantically linked data (**geometry quality control objects, building element objects**)
	Visual QC	▪ Semantically linked data (**geometric representations of building elements, tasks**) filtered by (**camera location**)	▪ Semantically linked data (**visual defect objects, building element objects, images**)

	DigiTAR	▪ Authentication Tokens ▪ Semantically linked data (**geometric representations of building elements, building element objects, geometric representations of H&S elements, H&S elements**) filtered by (**AR glasses/tablets location**) ▪ Semantically linked data (**geometry quality control objects, building element object**) ▪ Semantically linked data (**visual defect objects, building element objects, images**)	▪ Defect Notification (**building element objects, images, type of remedial works**) ▪ Hazard Notification (**building element objects, images, type of mitigation works**)
Visualisation	DCC	▪ Authentication Tokens ▪ Semantically linked data (**geometric representations of building elements, building element objects, geometric representations of H&S elements, H&S elements, tasks**) filtered by (**time range**) ▪ Semantically linked data (**geometry quality control objects, building element object**) ▪ Semantically linked data (**visual defect objects, building element objects, images**) ▪ Timestamped measurements of actual resources. (**longitude, latitude, altitude, actual resource objects**)	-

To meet the diverse data needs of the various COGITO applications, the Data Management Layer asynchronously bridges the different data stored in the Persistence Layer via a set of dedicated interfaces. Table 7 shows the identified interfaces of the DTP with their high-level specifications. These interfaces have been processed and merged based on the type and applied filters.

Table 7 DTP's external interfaces

Type	Data Exchange	Protocol	API	Data Type
Input Data	Timestamped measurements of location tracking devices (**longitude, latitude, altitude, tags**)	MQTT	-	JSON
	Annotated imagery datasets. (**building element objects, images**)	HTTPS	REST	JSON
	Annotated point clouds. (**building element objects, point cloud datasets**)	HTTPS	REST	JSON
	Training KPIs for workers (**actual resource objects, performance metrics objects**)	HTTPS	REST	JSON
	Semantically linked data (**geometric representations of H&S elements, H&S elements**)	HTTPS	REST	IFC
	Semantically linked data (**as-planned resource objects, task objects, building element objects**)	HTTPS	REST	JSON
	Semantically linked data (**actual resource objects, task objects, building element objects**)	HTTPS	REST	JSON
	Semantically linked data (**geometry quality control objects, building element objects**)	HTTPS	REST	JSON
	Semantically linked data (**visual defect objects, building element objects, images**)	HTTPS	REST	JSON
	Defect Notification (**building element objects, images, type of remedial works**)	AMQP	JMS	Images, JSON
	Hazard Notification (**building element objects, images, type of mitigation works**)	AMQP	JMS	Images, JSON
Output Data	Authentication Tokens	HTTPS	REST	JSON

	Semantically linked data (**geometric representations of building elements, building element objects, task objects**) filtered by (**location, time range**)	AMQP	JMS	OBJ, IFC, JSON
	Semantically linked data (**geometric representations of building elements, building element objects, geometric representations of H&S elements, H&S elements, task objects**) filtered by (**location, time range**)	AMQP	JMS	OBJ, IFC, JSON
	Timestamped measurements of actual resources (**longitude, latitude, altitude, actual resource objects**)	HTTPS	REST	JSON
	Semantically linked (**actual resource objects, tags**)	HTTPS	REST	JSON
	Semantically linked data (**as-planned resources ,task objects, building element objects**)	HTTPS	REST	JSON
	Initial data (**as-planned resources**)	HTTPS	REST	JSON
	Initial linked data (**building element objects, construction schedule**)	HTTPS	REST	JSON
	Semantically linked data (**geometry quality control objects, building element objects**)	HTTPS	REST	JSON, BCF
	Semantically linked data (**visual defect objects, building element objects, images**)	HTTPS	REST	JSON, BCF
	Semantically linked data (**geometric representations of H&S elements, H&S elements**)	HTTPS	REST	IFC
	Defect Notification (**building element objects, images, type of remedial works**)	AMQP	JMS	Images, JSON
	Hazard Notification (**building element objects, images, type of mitigation works**)	AMQP	JMS	Images, JSON
	Semantically linked data (**geometric representations of building elements, annotated point clouds**) filtered by (**time range**)	AMQP	JMS	Point Cloud, JSON

7 Messaging Layer

The Messaging Layer is responsible for transmitting data asynchronously between the various components of the Data Management Layer and the Data Post-Processing Layer. In addition, the Messaging Layer is responsible for delivering the notifications produced by the DTP to the COGITO applications. Since the COGITO applications will be implemented using different technologies and communication protocols, a system that provides the infrastructure compatible with these technologies is required. For this purpose, the Messaging Layer provides the Apache ActiveMQ Artemis message broker, which enables asynchronous data transmission utilising the publish/subscribe pattern and supports multiple messaging protocols such as Advanced Message Queueing Protocol (AMQP), Streaming Text Oriented Messaging Protocol (STOMP) and Message Queue Telemetry Transport (MQTT).

The primary reason for using a message broker is to increase the performance and reliability of the COGITO system. The message broker supports service-oriented communication approaches and offers a service allowing bi-directional communication between the various components with fault-tolerance capabilities. Supposing that any time consumer happens to be offline for a short period due to connectivity issues, the message broker will continue accepting messages by storing them in the embedded queues. On the other hand, the DT Runtime component of the Data Management Layer provides the integration framework needed for configuring inbound and outbound gateways and hosting the defined message routers, message translators and endpoints. Figure 22, shows the messaging functionalities provided by the Data Management Layer and the Messaging Layer.

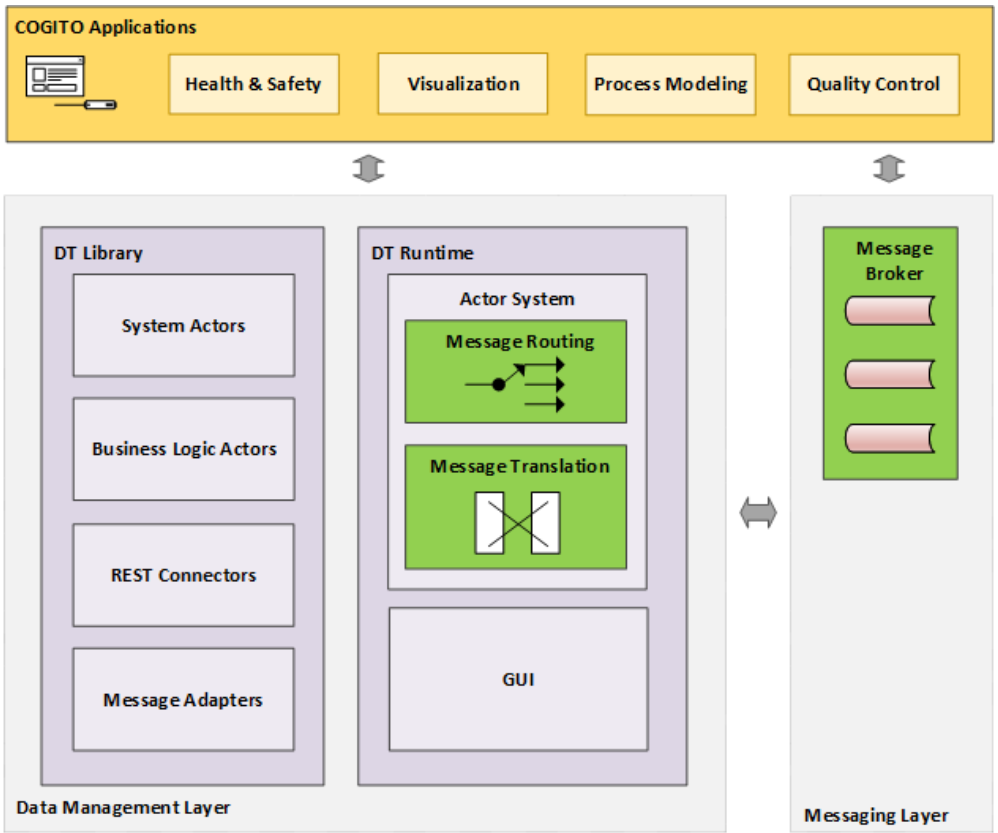


Figure 22 Messaging functionalities provided by the Data Management and Messaging layers

As shown in Figure 22, the DT Runtime component provides the integration framework, which offers dynamic message routing patterns. There are cases where the producers do not know the exact channel that the message to the consumer will get, but the producer sends the message to the DT Runtime, determining how to deliver the message to the consumer. In addition, if the COGITO applications do not agree on the format of the messages, DT Runtime offers a set of filters for converting message payloads that contain the same conceptual information from one form to another.

8 Data Post-Processing Layer

The Data Post-Processing Layer comprises components for handling time-consuming processes such as IFC optimisation, MVD model checking, B-rep geometry generation and geometric clash detection. These components are modular, and additional can be included in an extensible manner. They contain a set of low- and high-level libraries to perform their business logic operations. The low-level libraries support multithread processing and exchange information with the high-level libraries through the Java Native Interface (JNI) programming framework.

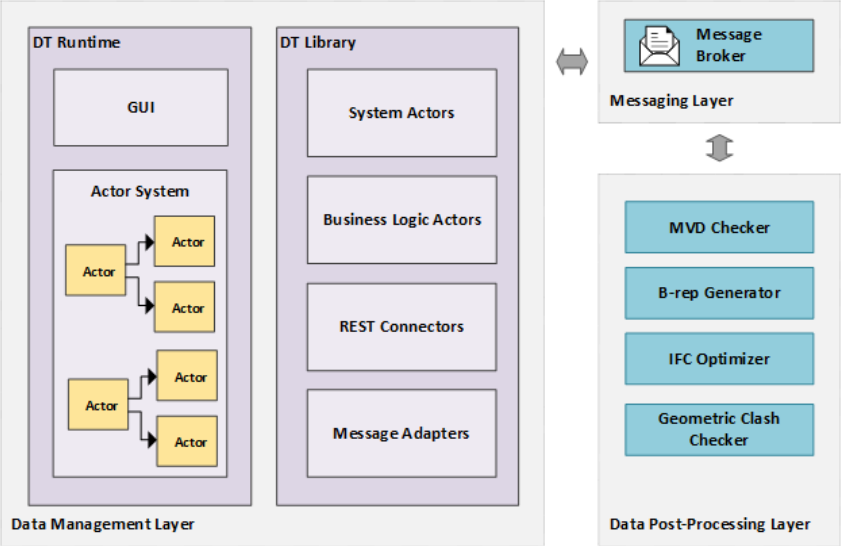


Figure 23 Data flow between Data Post-Processing and the Data Management layers

The components are packaged as Software as a Service (SaaS) applications and deployed in a private cloud computing infrastructure. In the proposed architecture design, we defined the following components as a basis of the Data Post-Processing Layer:

- **The Model-View Definition (MVD) Checker** helps the BIM Manager to validate IFC files in terms of data completeness and semantic consistency against predefined rules following the MVD specification. This component reports the detected issues using machine-readable data storage formats such as JSON and XML.
- **The B-rep Generator** reads the geometric information generated from the IFC Geometry Exporter to generate triangulated B-rep solids of the structural and non-structural elements. This component exports the geometric information using open formats such as OBJ and glTF.
- **The IFC Optimiser** performs lossless compression of an IFC file to speed up loading and data transformation processes. This component generates a new IFC with reduced file size.
- The **Geometric Clash Checker** detects clash and containment errors to create additional semantic relationships between existing entities of the unified knowledge graph.

As shown in Figure 23, each of these components exchanges information asynchronously through the Messaging Layer using data structures which conform to openBIM standards. Depending on the complexity of the BIM model, these services require more time to perform their business logic operations. For instance, the B-rep Generator generates triangulated B-rep solids of the structural and non-structural building elements, optimised for web-based graphic viewers, avoiding verbosity. The integration framework provided by the DT Runtime controls the biding between the components of the Data Post-Processing Layer and various data-driven modules which consume the processed data.

8.1 Architecture

The three main components of the Data Post-processing Layer are deployed as lightweight standalone Spring Boot applications, following the microservice architecture design pattern. Because they don't have GUI and persistence, some Spring Framework components, such as Spring MVC, Spring JPA, and Spring Security are not included in the final packages. Instead, a looser of coupling is used through the Spring JMS component, enabling asynchronous communication between the Data Post-processing Layer and the Data Management Layer. Security in message passing is assured through various encryption protocols and the connection credentials defined by the message broker.

A methodology for achieving a significant reduction in the executions times is the separation of their business logic operations into two groups a) high-level operations such as IFC data handling using high-level programming environments such as Java, and b) low-level geometric operations such as B-rep generation and mesh triangulation using low-level programming environments such as C/C++.

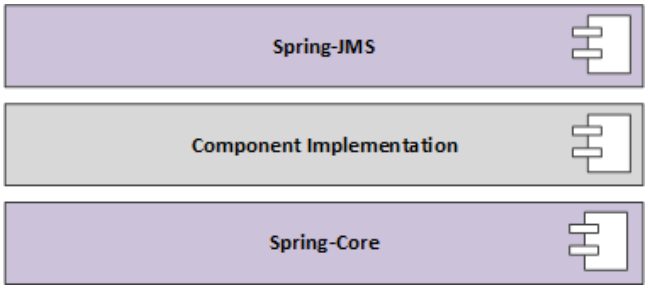


Figure 24 Stack-diagram of Data Post-Processing Layer components

Each component of the Data Post-Processing Layer provides an API including several indicators to inspect the health of running processes, memory usage, error logging and more. Figure 24 illustrates the stack diagram of these components.

8.2 Model View Definition (MVD) Checker

The IFC specification includes a multi-domain information model for capturing building data such as geometry, materials, components, properties and more. To support specific data exchange requirements between different tools and processes, only a subset of the IFC specification is required in terms of user entities and properties. The Model View Definition (MVD) specification allows the definition of reusable Concept Templates and Rules to describe the data exchange requirements precisely. Along with the maintenance of the IFC specification, buildingSMART has published the following general-purpose Model View Definition schemes:

- **IFC Reference View** is mainly used by tools and services that do not require geometry modifications. The geometric representation is optimised for analysis and display purposes but excludes the parametric geometry definitions.
- **IFC Design Transfer View** supports the editing of geometric representations of structural and non-structural building elements. It is the preferred MVD in COGITO solution because it enables the enrichment of the BIM model with new properties and geometric objects.

The DTP provides the MVD completeness checking component to help COGITO users validate the BIM model's completeness against predefined concepts using the mvdXML specification. For instance, completeness checking is essential to ensure that each structural building element of the IFC model has a property used to create the semantic link between the element and the construction schedule.

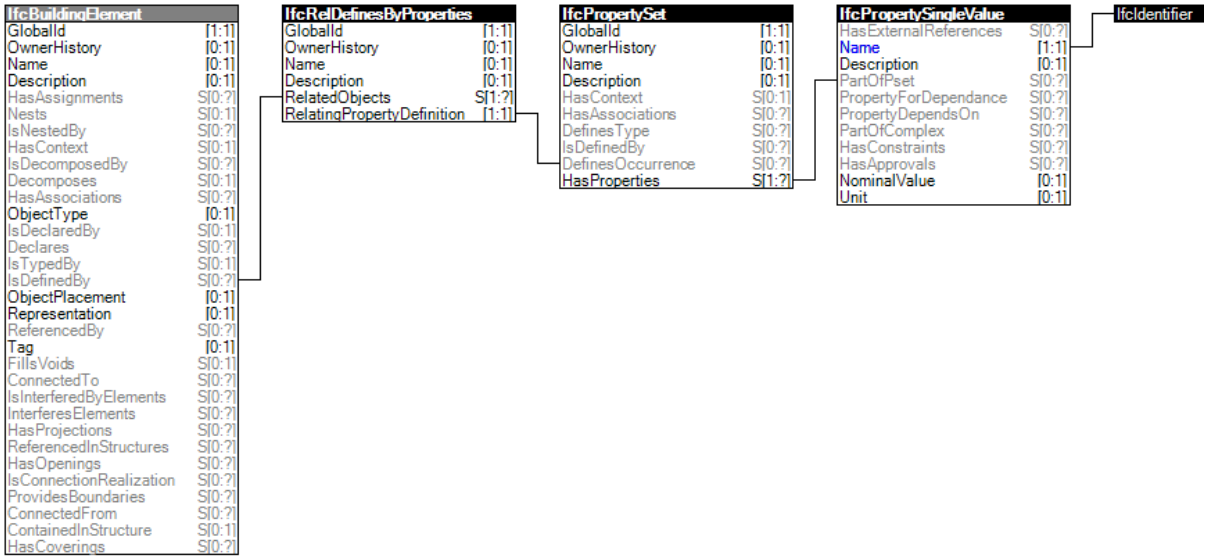


Figure 25 Example of a concept template for validating IFC properties

As shown in Figure 26, three steps are needed to achieve automatic MVD validation of a BIM model: a) The creation of the mvdXML file using the IfcDoc tool; b) The application of the concept rules on the IFC objects using the algorithms provided by the IFC Library, and c) The generation and the visualisation of the error reports.

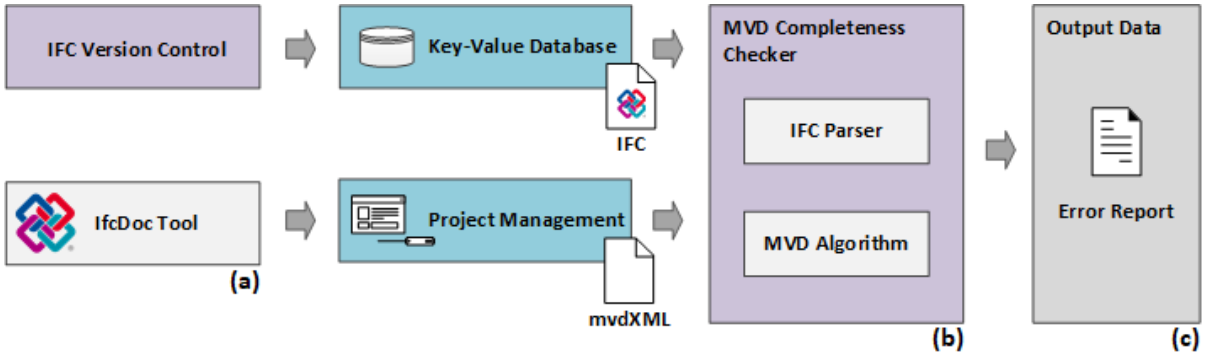


Figure 26 MVD model checking process

BuildingSMART International has developed IfcDoc to improve the computer-interpretable implementation of the IFC specification. The user can create custom definitions and assign new concepts to them. Each concept contains a) a connection to an IFC entity, b) an additional concept to filter the instances by validating the applicability of the relational diagram, and c) the rules along with their parameters and logical operations.

8.3 B-rep Generator

The IFC data have geometric representations which are not in a graphics-friendly format. Therefore, the B-rep Generation component of DTP transforms the IFC geometric data into triangulated Boundary representations, which are optimised for graphic viewers, avoiding verbosity and without losing critical information. In this transformation process, all geometric representations of the structural and non-structural building elements are transformed into 3D triangle surface sets in a two-step conversion process. In the first step, the IFC Geometry Exporter exports the geometric information in XML format. Then, in the second step, the geometry data are loaded into and processed by the B-rep generation component using the JNI programming interface. The output of the B-rep generation component (the generated B-rep objects), is exported in an optimised graphic data format such as OBJ and glTF, containing triangulated polygon surfaces.

The B-rep generation component transforms the geometric representation of every element in the input IFC data file into a triangulated surface set. Suppose the geometric representation of the element is parametric. In that case, the B-rep generation component applies all the necessary geometric operations to transform it to a boundary representation first (set of outward-oriented polygon surfaces) and then applies a triangulation process to every polygon surface to create a final triangle set. Multiple parametric geometric descriptions are supported for every element in the BIM file, including extruded area solids, half-space solids, and CSG boolean operations on these descriptions of finite depth. If the element has a non-parametric description, then only a triangulation process is applied on its boundary polygonal surfaces to transform them into a triangle set.

8.4 IFC Optimiser

The IFC exportation plugins of the BIM authoring tools perform the serialisation of the IFC objects and generate the final IFC using the STEP data format. The generated IFC data often contain duplications of the same information. The IFC Optimisation component performs lossless compression of IFC data to speed up loading and data transformation processes such as the B-rep Generation and Knowledge Graph Generation. It uses the IFC objects loaded by the BIM Management component and performs the following steps: a) converts the content of each object into a hash value, b) reduces the size of the generated hash table by removing the duplications, and c) updates the references of the removed entries in the remaining entries.

8.5 Geometric Clash Checker

Due to the fact that IFC files can be exported by various BIM authoring tools which are operated by users of variable levels of expertise, the relationship between construction elements and their associated construction zones, which is required for the complete formation of the knowledge graph of the COGITO data model, might be missing in these exported IFC files. To overcome this obstacle, the Geometric Clash Checker (GCC), is an ETL tool that is designed to perform enrichment of input IFC files with this required (construction element) – (construction zone) containment relationship. The enrichment is performed using pure geometric clash detection methods, without any prior knowledge. In short, GCC links a construction element A with a construction zone B only if their geometric boundary representations intersect.

9 Conclusions

This deliverable analysed in detail the final architecture of COGITO's DTP. The DTP plays a central role to COGITO's architecture, as it is responsible not only for performing user management tasks but also for handling and processing all incoming data traffic and for converting COGITO's input data to suitable data formats for all COGITO applications. A top-down approach was used to describe each component thoroughly, starting from an overview of the identified layers to a detailed description of the contained software components. The DTP has been divided into six layers each serving a specific functional purpose:

- The *Authentication Layer* performs user management tasks according to the open-source project Keycloak's specification.
- The *Data Ingestion Layer* includes operational blocks for transforming the incoming data into a unified knowledge graph and other necessary COGITO data models.
- The *Data Persistence Layer* provides data storage functionality.
- The *Data Management Layer* responds to COGITO applications data requests, synchronising the multiple parallel responses to data queries from these applications in a fast and efficient manner.
- The *Messaging Layer* coordinates the asynchronous data transmittance of data and notifications between the DTP and the other COGITO applications.
- The *Data Post-Processing Layer* to perform ETL and Model Checking operations on COGITO's BIM data.

We also described the interfaces and the data exchanges among these layers. In addition, we present the detailed technology stack used for the final implementation of the software components and define the data flows and the interface specifications. Many components of the above layers (Data Post-processing Layer, Data Ingestion Layer, Data Management Layer) can act as standalone services enabling parallel executions in a highly containerised environment following a Service-Oriented Architecture (SOA) pattern. This is an efficient design approach, achieving minimum response time to queries of variable processing load. Furthermore, the communication among the different subcomponents of the defined layers is performed efficiently, transparently, and well-structured using an appropriate message routing scheme provided by the Data Management and Messaging layers. The final design of the DTP is fully aligned with the requirements defined in the deliverables "D2.1 - Stakeholder requirements for the COGITO system" and "D2.5- COGITO System Architecture v2".

References

[1] “SPHERE project,” 2020. [Online]. Available: https://sphere-project.eu/technology/.

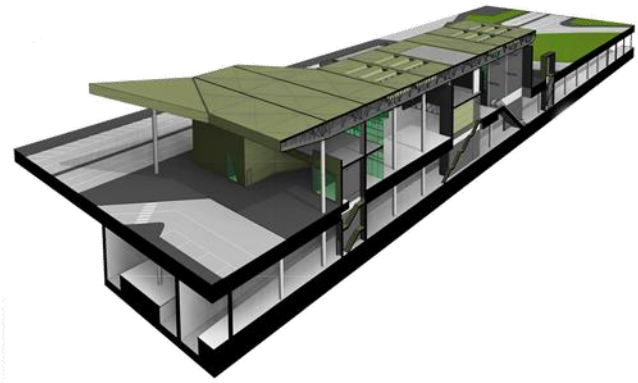
[2] “BIMprove project,” 2021. [Online]. Available: https://www.bimprove-h2020.eu/project/.

[3] “ASHVIN project: D1.1 Launch version of ASHVIN platform,” 2021. [Online]. Available: https://ashvin.eu.

[4] COGITO, “D3.2 - COGITO Data Model & Ontology,” 2021.

[5] COGITO, “D2.1 - Stakeholder requirements for the COGITO system,” 2021.

[6] COGITO, “D2.5 - COGITO System Architecture V2,” 2022.



COGITO

CONSTRUCTION PHASE DIGITAL TWIN MODEL

cogito-project.eu

HYPERTECH energy labs

UCL

AARHUS UNIVERSITET

THE UNIVERSITY of EDINBURGH

BoC www.boc-group.com

iti

UNIVERSIDAD POLITÉCNICA MADRID POLITÉCNICA

QUE TECHNOLOGIES

Novitech NEW INFORMATION TECHNOLOGIES

ASM

ferrovial construction

ΟΛΥΜΠΙΑ ΟΔΟΣ

RHOMBERG SERSA RAIL GROUP

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 958310