

BuildON

D3.1 - Technical requirements and architecture design

WP3, T3.1

Authors: UCL, CARTIF, NTUA, HOL, Rochelle, CORDIA, YOD

DUE DATE: [March 2024](#)



Co-funded by
the European Union

Copyright Message

This report, if not confidential, is licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0). A copy is available here: <https://creativecommons.org/licenses/by/4.0/>. You are free to share (copy and redistribute the material in any medium or format) and adapt (remix, transform, and build upon the material for any purpose, even commercially) under the following terms: (i) attribution (you must give appropriate credit, provide a link to the license, and indicate if changes were made; you may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use); (ii) no additional restrictions (you may not apply legal terms or technological measures that legally restrict others from doing anything the license permits).

Disclaimer



Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or CINEA. Neither the European Union nor the granting authority can be held responsible for them.



Innovate
UK

This work was co-funded by UK Research and Innovation (UKRI) under the UK government's Horizon Europe funding guarantee grant number 10069441.



Co-funded by
the European Union

DOCUMENT INFORMATION

Project Acronym	BuildON
Project Title	Affordable and digital solutions to Build the next generation of smart EU buildings
Project Coordinator	FUNDACIÓN CARTIF
Project Duration	May 2023 – October 2026 (42 months)
GA Number	101104141
Work Package	WP3 - IoT Edge-Cloud interoperable framework for building data exchange and operation
Task (s)	T3.1 - Technical requirements and architecture design
Deliverable	D3.1
Lead beneficiary	20 - UCL
Other Beneficiaries	1 - CARTIF, 2 - NTUA, 8 - HOL, 11 - Rochelle, 14 - CORDIA, 18 - YOD
Author (s)	20 - UCL
Reviewer (s)	3 - EURAC, 16 - ONE
Dissemination level*	PU-Public
Due date of deliverable	31 October 2023
Actual submission date	27 March 2024



DOCUMENT HISTORY

Version	Date	Contributor(s)	Description
0.1	11/07/2023	ENG	First proposal of ToC
0.2	08/09/2023	ENG	Draft contributions from ENG
0.3	05/10/2023	NTUA	Draft contributions for NTUA
0.4	03/01/2024	UCL	Draft restructuring proposal from UCL
0.5	02/02/2024	UCL	Draft contributions from UCL
0.6	23/02/2024	NTUA, YOD, HOL	Draft contributions from partners
0.7	04/03/2024	UCL	Reviewed version shared with partners
0.8	07/03/2024	NTUA, YOD, HOL, CARTIF	Internal review
0.9	04/03/2024	UCL	Reviewed version shared with reviewers
0.10	12/03/2024	EURAC, ONE	External review
0.11	19/03/2024	UCL	Final version
1.0	27/03/2024	CARTIF	Submitted deliverable

List of organisations

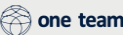
No	Participant Name	Short name	Country	Logo
1	FUNDACIÓN CARTIF	CARTIF	ES	
2	NATIONAL TECHNICAL UNIVERSITY OF ATHENS	NTUA	GR	
3	ACCADEMIA EUROPEA DI BOLZANO	EURAC	IT	
4	ENGINEERING S.p.A.	ENG	IT	
5	VTT TECHNICAL RESEARCH CENTER	VTT	FI	
6	RINA CONSULTING S.P.A.	RINA-C	IT	
7	FUNDAZIONE ICONS	ICONS	IT	
8	HOLISTIC IKE	HOLISTIC	GR	
9	PRZEDSIĘBIORSTWO ROBÓT ELEWACYJNYCH FASADA SP.ZO.O.	FAS	PL	
10	ELECTRICITÉ DE FRANCE	EDF	FR	
11	LA ROCHELLE UNIVERSITY	ROCHELLE	FR	
12	HELSINKI CITY APARTMENT	HEKA	FI	
13	VEOLIA SERVICIOS LECAM SOCIEDAD ANONIMA UNIPERSONAL	VEOLIA	ES	
14	CORDIA	CORDIA	GR	
15	CITY OF GDYNIA	GDYNIA	PL	
16	One team SRL	ONE	IT	
17	EUROHEAT & POWER	EHP	BE	
18	YODIWO	YOD	SE	
19	BEEPLANET FACTORY	BEE	ES	
20	UNIVERSITY COLLEGE LONDON	UCL	UK	

Table of contents

1	Introduction	1
1.1	Purpose of the document	1
1.2	Relation to other activities	1
1.3	Structure of the document	2
2	Methodology	3
3	Technical Use Cases (TUCs).....	5
3.1	TUC-1 – Monitor and Assessment services	7
3.2	TUC-2 – Monitor, Assessment and Prediction services	7
3.3	TUC-3 – Monitor, Assessment, Prediction and Optimisation services	8
3.4	TUC-4 – Digital Building Logbook	9
4	Core Functionalities	10
4.1	F-1 – Internal functionalities	10
4.2	F-2 – External functionalities	11
5	Conceptual Architecture.....	13
5.1	Edge Domain Controller (DC)	13
5.2	Cloud Controller (CC)	14
5.3	Data Access and Control Manager (DACM).....	14
5.4	BuildON Data Repository	14
5.5	Universal API.....	15
6	Functionalities Sequence Diagrams.....	16
6.1	F-1 – Internal functionalities	16
6.2	F-2 – External functionalities	28
7	Functional & Non-Functional Requirements.....	32
7.1	BuildON non-functional requirements	32
7.1.1	Modularity	32
7.1.2	Efficiency.....	32
7.1.3	Interoperability	32
7.1.4	Security	33
7.1.5	Reliability and availability	33
7.2	Core components functional requirements	33
7.2.1	Edge Domain Controller (DC)	33

7.2.2	Cloud Controller (CC)	35
7.2.3	Data Access and Control Manager (DACM)	36
7.2.4	BuildON Data Repository	37
7.2.5	Universal API	38
8	Software Architecture	40
8.1	Components	40
8.1.1	Edge Domain Controller (DC)	40
8.1.2	Cloud Controller (CC)	46
8.1.3	Data Access and Control Manager (DACM)	48
8.1.4	BuildON Data Repository	49
8.1.5	Universal Building API	52
8.2	Interfaces specification	54
8.2.1	DC and Field devices	55
8.2.2	DC and DACM	55
8.2.3	CC and Web Services	56
8.2.4	CC and DACM	56
8.2.5	DACM and Repository	57
8.2.6	DACM and Universal API	58
8.2.7	Repository and Universal API	58
8.2.8	Universal API and Services	59
8.2.9	All core components and Users	60
9	Deployment Plan Procedure	63
9.1	Technologies	63
9.2	Security mechanisms	64
10	Conclusions	66

Figures

Figure 1 – Relation of D3.1 within WP3 with other BuildON activities	2
Figure 2 – Methodology for the definition of BuildON technical requirements and architecture	4
Figure 3 – BuildON internal and external functionalities	10
Figure 4 – BuildON conceptual architecture	13
Figure 5 – Sequence diagram of F-1.1 (register users and connect components)	18



Figure 6 – Sequence diagram of F-1.2 (configure field devices)	20
Figure 7 – Sequence diagram of F-1.3 (configure web services).....	21
Figure 8 – Sequence diagram of F-1.4 (generate and enrich graphs)	23
Figure 9 – Sequence diagram of F-1.5 (register services requirements and validate graphs)	25
Figure 10 – Sequence diagram of F-1.6 (read, preprocess and store raw data)	27
Figure 11 – Sequence diagram of F-2.1 (read dynamic data).....	29
Figure 12 – Sequence diagram of F-2.2 (write commands)	31
Figure 13 – BuildON software architecture.....	40
Figure 14 – Edge Domain Controller (Holistic DC) subcomponents diagram.....	42
Figure 15 – Edge Domain Controller (Yodiwo DC) subcomponents diagram	44
Figure 16 – Edge Domain Controller (Yodiwo DC) internal procedures.....	44
Figure 17 – Cloud Controller (Holistic DC) subcomponents diagram.....	47
Figure 18 – Data Access and Control Manager (DACM) subcomponents diagram.....	49
Figure 19 – Data Repository subcomponents diagram	51
Figure 20 – Universal API subcomponents diagram.....	54

Tables

Table 1. Technical UCs mapping to the Business UCs from WP2 D2.1	5
Table 2. Functional Requirements of the Edge Domain Controller	34
Table 3. Functional Requirements of the Cloud Controller	35
Table 4. Functional Requirements of the Data Access and Control Manager	36
Table 5. Functional Requirements of the BuildON Data Repository	37
Table 6. Functional requirements of the Universal API.....	38
Table 7. Libraries, frameworks and tools to support the Edge Domain Controller for Holistic DC	45
Table 8. Libraries, frameworks and tools to support the Edge Domain Controller for Yodiwo DC	45
Table 9. Libraries, frameworks and tools to support the Cloud Controller.....	47
Table 10. Libraries, frameworks and tools to support the Data Access and Control Manager	49
Table 11. Libraries, frameworks and tools to support the BuildON Data Repository	51
Table 12. Libraries, frameworks and tools to support the Universal API.....	54
Table 13. Interfaces between DC and Field devices.....	55
Table 14. Interfaces between DC and DACM	55
Table 15. Interfaces between CC and Web Services	56

Table 16. Interfaces between CC and DACM	56
Table 17. Interfaces between DACM and Repository	57
Table 18. Interfaces between DACM and Universal API	58
Table 19. Interfaces between Repository and Universal API	58
Table 20. Interfaces between Universal API and Services.....	59
Table 21. Interfaces between all core components and Users	60

Abbreviations and Acronyms

Acronym	Description
AI	Artificial Intelligence
API	Application Programming Interface
BACN	Building Automation and Control Network
BEMS	Building Energy Management System
BIM	Building Information Modelling
CC	Cloud Controller
DACM	Data Access Control Manager
DBL	Digital Building Logbook
DC	Domain Controller
DoA	Description of the Action
DSPP	Data Security, Privacy and Protection
ESS	Energy Storage System
ETL	Extract, Transform, and Load
EV	Electric Vehicle
GPIO	General Purpose Input/Output
GUI	Graphical User Interface
H&C	Heating and Cooling
HTTPS	Hypertext Transfer Protocol Secure
HVAC	Heating, Ventilation and Air Conditioning
IEQ	Indoor Environment Quality

IFC	Industry Foundation Classes
IoT	Internet of Things
KPIs	Key Performance Indicators
MAPO	Monitor, Assessment, Prediction and Optimisation services
ML	Machine Learning
MoSCoW	Must-haves, Should-haves, Could-haves and Won't-haves – this time
MQTT	Message Queue Telemetry Transport
MVD	Model View Definition
NFRs	Non Functional Requirements
O&M	Operation & Maintenance
PaaS	Platform as a Service
PV	Photovoltaic system
RES	Renewable Energy Sources
SHACL	Shape Constraint Language
SOA	Service-Oriented Architecture (SOA)
TUCs	Technical Use Cases
UCs	Use Cases
UML	Unified Modelling Language

Executive summary

This report outlines the methodology used to define the requirements and design procedures for the BuildON software architecture. It also details the resulting architecture, which will serve as the foundation for all future technical development within BuildON. The goal is to support various usage scenarios (referred to as technical use cases) in both residential and non-residential contexts, ensuring a highly replicable solution that deliver diverse functionalities while minimising the need for customisation efforts. The resulting components, which aid in achieving this goal, are built upon the conceptual architecture outlined in the Description of Action, and include edge domain controllers, cloud controllers, data access and control manager elements, a common data repository, and a universal API. This report explores each of these components, detailing their associated functional and non-functional requirements, along with their interfaces. Throughout the project demonstration phase, the proposed solution will be tested, refined, and validated to ensure its effectiveness and reliability.

Keywords

Software architecture, technical requirements, technical use cases, functionalities, core components, replicability

1 Introduction

BuildON focuses on supporting the development of the next generation of Building Energy Management Systems (BEMS). Its aim is to offer a widely replicable, generic solution for streamlining onboarding procedures of advanced data-driven services across diverse buildings, allowing them to seamlessly gather data and transmit control signals. Among these categories are Monitoring, Assessment, Prediction, and Optimisation (MAPO) services, which rely on Artificial Intelligence (AI) and Machine Learning (ML) solutions that provide insights into consumption patterns of building occupants, forecast on-site energy generation, and implement automated controls. Such controls involve improvements for indoor environmental quality and energy efficiency, as well as building-to-grid integration supported by Renewable Energy Sources (RES) and energy storage systems.

To accomplish its aim, BuildON promotes data integration processes using non-proprietary, interoperable, and semantically driven components. This approach ensures easier replicability of data-driven smart building solutions compared to current ad hoc methods while reducing vendor lock-in and siloed data. At the core of this effort is the BuildON software architecture, designed to align with the most recent European initiatives and reference architectures, including the International Data Spaces Association¹, GAIA-X federated secure data infrastructure², FIWARE framework of open source platform³, and Bridge's Data Exchange Reference Architecture 3.0⁴. Based on their key design elements, such as interoperability layers (technical, semantics, organisational), security mechanisms (especially identity and access management), and marketplace paradigms, BuildON software architecture provides a technology stack to depart from a static perspective of buildings as assets to a more flexible Platform-as-a-Service (PaaS) delivery approach. Such an approach will significantly accelerate the smart transformation of existing and new buildings.

1.1 Purpose of the document

This deliverable (D3.1) presents the main outcomes of Task 3.1 - "Technical requirements and architecture design". It aims to define the BuildON software architecture and high-level project requirements to support the goals mentioned above and defined usage scenarios, here referred to as Technical Use Cases (TUCs). The document outlines the core functionalities, requirements, components and their interfaces within the architecture, guiding all future technical advancements within the project.

1.2 Relation to other activities

Figure 1 below depicts the main relation of this deliverable with other Work Packages (WPs) and their deliverables developed within the BuildON project.

¹ <https://internationaldataspaces.org/>

² <https://gaia-x.eu/>

³ <https://www.fiware.org/>

⁴ <https://data.europa.eu/doi/10.2833/815043>

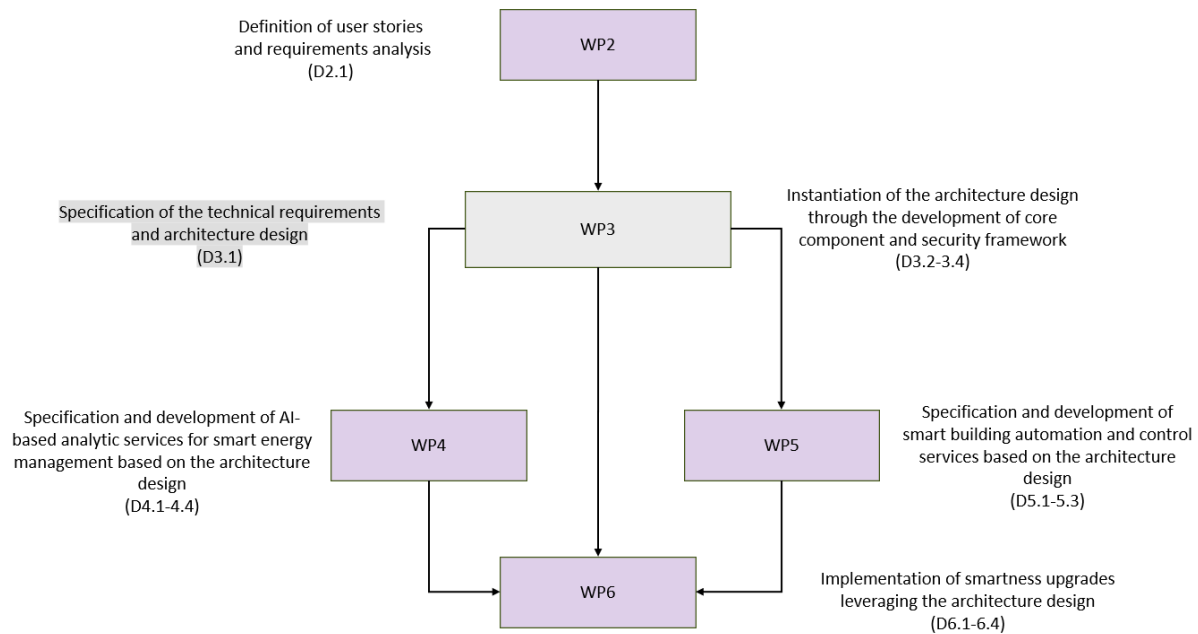


Figure 1 – Relation of D3.1 within WP3 with other BuildON activities.

1.3 Structure of the document

The remainder structure of the document consists of the following sections:

- **Section 2** outlines the methodology for establishing BuildON's technical requirements and software architecture.
- **Section 3** introduces the TUCs extracted from WP2 D2.1, titled “Human-smart assets interaction tools: analysis of user stories and requirements.” These TUCs inform the spectrum of services the software architecture must support.
- **Section 4** showcases the fundamental functionalities necessary to uphold BuildON's TUCs.
- **Section 5** presents the conceptual architecture proposed in the Description of the Action (DoA) and amends BuildON's core components and their primary dependencies to align with the identified functionalities.
- **Section 6** illustrates the flow of operations and interactions between BuildON's core components based on the identified functionalities.
- **Section 7** identifies functional and non-functional requirements linked with the core components.
- **Section 8** devises the BuildON software architecture by delineating the subcomponents included within the core components.
- **Section 9** elaborates on the deployment procedures for BuildON.
- **Section 10** outlines concluding remarks, summarising key findings and implications.

The work presented in this deliverable presents a detailed proposal for the BuildON reference architecture and forms the basis for implementing the various software components identified.

2 Methodology

This section outlines the methodology devised to establish the technical requirements and software architecture of BuildON as part of WP3, an IoT edge-cloud interoperability framework for building data exchange and operation. This architecture serves as the blueprint for all future technical advancements within the project. Its goal is to be universally applicable, offering a highly transferable solution not tailored to the pilots and adaptable to diverse building typologies with similar or differing energy ecosystems. To achieve this, the technical requirements and software architecture of BuildON were defined following seven steps outlined in the subsequent sections as described below and illustrated in Figure 2:

- **Step 1:** review the content of WP2 D2.1, which collected use cases and user requirements, to extract pertinent (TUCs). These TUCs are used to inform the range of services that the software architecture must accommodate.
- **Step 2:** identify the core functionalities needed to support the TUCs outlined in Step 1. Distinguish between internal functionalities, which pertain to configuring interactions among components within the architecture (e.g., user registration and component connection), and external functionalities, focused on exchanging data among the architecture components and the Monitor, Assessment, Prediction and Optimisation (MAPO) services and Digital Building Logbook (DBL).
- **Step 3:** revise the DoA conceptual architecture, updating BuildON's core components and their high-level dependencies to support the functionalities identified in Step 2.
- **Step 4:** specify Unified Modelling Language (UML) sequence diagrams to illustrate the flow of operations and interactions between BuildON's core components based on the functionalities identified in Step 2.
- **Step 5:** identify each component's functional and non-functional requirements. Articulate the capabilities and constraints that arise from the design or implementation, as needed, to ensure the seamless execution of the operations and interactions outlined in the sequence diagrams from Step 4. Each requirement should possess a distinct identification and be prioritised according to the criteria of the MoSCoW (Must-haves, Should-haves, Could-haves and Won't-haves – this time).
- **Step 6:** design the BuildON software architecture by outlining the subcomponents encapsulated within the core components. Develop a UML components diagram that aligns with the specified component requirements in Step 5. Additionally, specify the required interfaces for facilitating seamless interactions among the components.
- **Step 7:** describe deployment procedures, including technologies used, data sharing procedures, backup and recovery processes, and security measures.

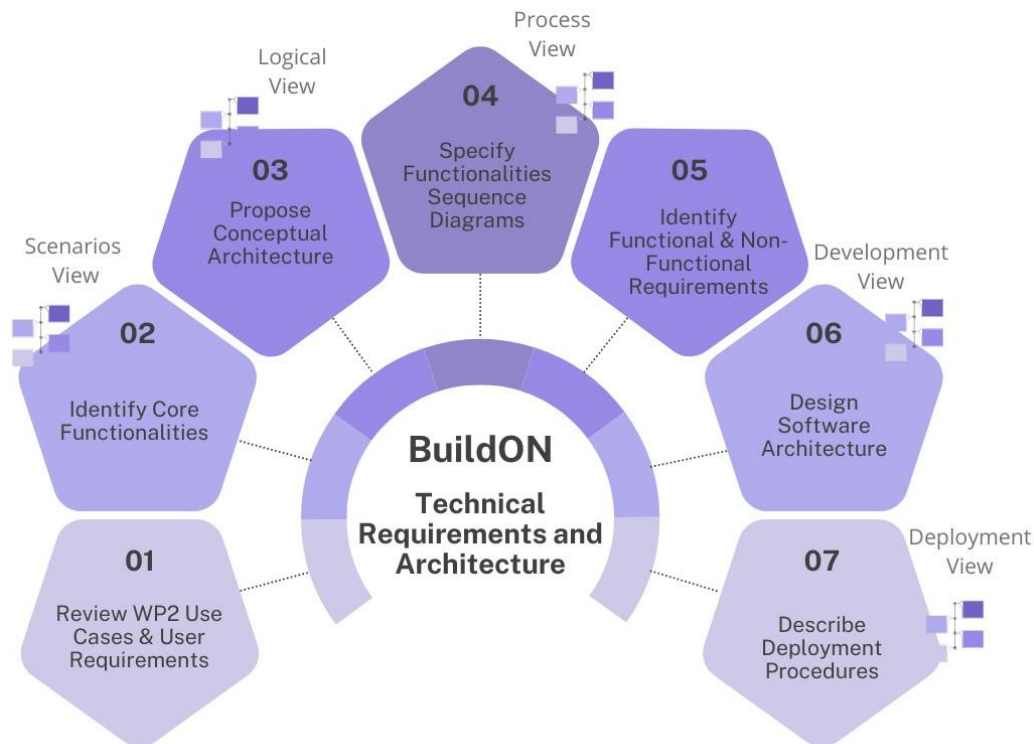


Figure 2 – Methodology for the definition of BuildON technical requirements and architecture.

Several steps in the proposed methodology align with the "4+1" View Model of Software Architecture^{5 6}, a framework that enhances architectural description by incorporating multiple views that can address different features and concerns during development. This approach facilitates comprehension for various stakeholders, including end-users, developers, project managers, and testers. The 4+1 View Model encompasses vital perspectives: scenario, logical, process, development and deployment views. These perspectives are addressed by the proposed methodology in the following manner:

- The Scenario View is shaped by the core functionalities necessary for deploying the TUCs.
- The Logical View is captured through a high-level conceptual architecture, outlining its core components.
- The Process View is illustrated using sequence diagrams, showing case concurrent processes that support the TUCs' core functionalities. It aids in deriving both functional and non-functional requirements.
- The Development View is constructed based on component diagrams, offering insights into the software modules and subcomponents.
- The Deployment View is represented by mapping software onto hardware, including non-functional requirements such as availability, security, reliability, and scalability.

The following sections describe each step of the methodology in detail.

⁵ Kruchten, Philippe, 1995, Architectural Blueprints - The "4+1" View Model of Software Architecture

⁶ Grgec, Miroslav; Muzar, Robert, 2007, Role of UML Sequence Diagram constructs in object lifecycle concept

3 Technical Use Cases (TUCs)

This section outlines a list of TUCs that classify the range of higher-layer BuildON services proposed in the DoA. Identifying these TUCs and their requirements is essential for guiding the design process of the BuildON software architecture, which will support their data access for monitoring and controlling the pilot buildings' resources.

The proposed TUCs have been derived from the business Use Cases (UCs) outlined in WP2 D2.1, titled "Human-smart assets interaction tools: analysis of user stories and requirements". These business UCs are centred around the user empowerment tools, MAPO services, DBL and their related user stories specific to each pilot. The TUCs specifically focus on the business UCs related to the MAPO services and the DBL. User empowerment tools are out of this scope because they do not necessarily rely on the dynamic data exchange capability provided by the BuildON architecture. Based on the scope descriptions of MAPO services and the DBL, the TUCs are organised into four distinct groups:

- TUC 1 – Monitoring and Assessment services: AI-based analytic services for smart energy management, including monitoring building energy assets, benchmarking, and fingerprinting services.
- TUC 2 – Monitoring, Assessment and Prediction services: AI-based analytic services for smart energy management, including detecting and proactively alarming faults and asset profiling for providing energy reduction recommendations.
- TUC 3 – Monitoring, Assessment, Prediction and Optimisation services: smart building automation and control services, including user comfort optimisation, cross-domain energy assets control, and grid-optimised energy flexibility services.
- TUC-4 – Digital Building Logbook: building data management and integration leveraging a common data repository.

Following an incremental approach, the MAPO-services-related TUCs are built on top of the previous ones. Table 1 depicts the mapping between the TUCs and their corresponding business UCs.

Table 1. Technical UCs mapping to the Business UCs from WP2 D2.1

Technical UC Group	Technical UC name	Business UC name	Business UC scope	Involved Services
TUC-1 – Monitoring and Assessment services	TUC-1.1 - Energy consumption and generation monitoring, analysis and reporting	UC4 - Building energy assets monitoring	Monitor, analyse, and optimise energy consumption across the facility's devices and processes.	s1.1
	TUC-1.2 - Building energy performance benchmarking	UC5 - Building automation benchmarking	Benchmark buildings for energy performance	s1.2
TUC-2 – Monitoring, Assessment and Prediction services	TUC-2.1 - Monitoring, detection and proactive alarming of faults	UC8 - Optimisation of O&M activities in tertiary buildings	Monitor the condition of facility-level equipment, reduce maintenance costs, and prevent equipment breakdowns.	s1.5

TUC-3 – Monitoring, Assessment, Prediction and Optimisation services	TUC-2.2 - Building energy reduction recommendations	UC6 - Building energy assets profiling	To review and recommend best solutions and strategies tailored to buildings' energy profile to reduce energy consumption	s1.3
	TUC-2.3 – Forecasting models for energy load predictions	UC7 - Building systems behaviour for optimal occupants' comfort	Monitor systems' devices and processes, performance and consumption, leading to optimised usage habits	s1.4
	TUC-3.1 – Automated lighting control for improved visual comfort	UC9 - Visual comfort control applications	Improve the indoor environment lighting conditions within the building, enhancing productivity and well-being.	s2.1
	TUC-3.2 - Automated control for optimal IEQ (thermal/visual comfort and air quality)	UC10 - Smart controls for building internal environmental quality	Optimise the building automation system, ensuring efficiency in operation and optimal indoor conditions	s2.2
	TUC-3.3 - Automated control for optimal RES generation and distribution	UC11 - Optimising heating & cooling with increased RES integration	To establish an optimisation control strategy for H&C, ensuring optimal RES integration and operation	s2.3
	TUC-3.4 - Automated heating system control for optimal thermal comfort	UC12 - Smart building configurations for optimised thermal comfort	To maximise thermal comfort by enhancing building performance to reduce heating demand and enable remote monitoring of energy levels in the building	s2.4
	TUC-3.5 - Automated control for building flexibility management	UC13 - Demand Response applications for optimal energy performance	To optimise energy distribution efficiently via demand response	s2.5
		UC14 - Building flexibility management for optimal environmental performance	To efficiently schedule the energy systems operation to maximise low-carbon energy consumption and efficiency	s2.6
TUC-4 – Digital Building Logbook	TUC-4.1 - Building data management and integration	UC3 - Building information via digital logbooks	To provide general, administrative, construction, operational and smart information about the building in a structured	All services

3.1 TUC-1 – Monitor and Assessment services

- **TUC-1.1 - Energy consumption and generation monitoring, analysis and reporting**

This TUC supports the Energy Assets Flow Monitor service (s1.1), which aims to model and provide visualisation of energy data from various building energy systems (e.g., heating, cooling, energy production and consumption, battery state of charge, etc.). This TUC must support a stand-alone dashboard capable of seamlessly processing and visualising historical and real-time data. The dashboard should contain multiple visualisation options, ranging from simple bar charts to more complex representations such as scatter plots, heat maps, box plots, pie charts, and histograms.

- **TUC-1.2 – Building energy performance benchmarking**

This TUC supports the Automation Benchmarks service (s1.2), which aims to assess energy efficiency improvement in buildings by using energy-related Key Performance Indicators (KPIs) derived from building typology and usage patterns. These KPIs will enable benchmarking of the building's energy performance, facilitating evaluation of the project's services across different energy efficiency scenarios. For that, this TUC needs to support the analysis of various data types, including energy consumption and generation per energy carrier, second-life battery measurements (state of charge, charging/discharging power, and setpoints), external weather conditions (temperature, solar irradiation), indoor conditions (temperature, air quality, etc.), occupancy periods, and system settings (e.g., temperature settings, operation schedules, etc.).

3.2 TUC-2 – Monitor, Assessment and Prediction services

- **TUC-2.1 – Monitoring, detection and proactive alarming of faults**

This TUC supports the Tertiary Building Management for Operation & Maintenance (O&M) service (s1.5), which aims to minimise service downtime by detecting anomalies and errors in predefined operating conditions in real-time. To achieve this, the TUC should provide alerts based on predefined signals in building subsystems and allow automatic reconfigurations based on user preferences. This TUC should leverage the outcomes of TUCs 1 and 2, incorporating their KPIs, benchmarking, and real-time monitoring functionalities.

- **TUC-2.2 – Building energy reduction recommendations**

This TUC supports the Energy Profiling service (s1.3), which aims to create an energy profile of buildings to predict energy consumption patterns and behaviours, allowing for comparisons between different buildings based on their operational characteristics. For that, this TUC must consider the buildings' energy use, generation and storage capabilities, heating & cooling systems and subsystems, overall and per-space consumption, and occupancy levels. Analysing this integrated information will allow building energy reduction recommendations to be provided.

- **TUC-2.3 – Forecasting models for energy load predictions**

This TUC supports the Optimal System Conditions via Occupant Behaviour service (s1.4), which aims to develop short-term, medium-term, and long-term forecasting models to predict energy loads

accurately. This TUC should consider factors such as seasonality trends, indoor conditions, occupancy levels, weather conditions, and other relevant data. To ensure accuracy and flexibility, data of varying granularities will be used, enabling the creation of models capable of delivering precise predictions across different forecasting horizons, including day-ahead, week-ahead, and month-ahead scenarios. Various forecasting techniques will be employed, including traditional methods like auto-regression and advanced AI methods such as ML (Decision Trees, Random Forests, Support Vector Machines, Linear Regression) and deep learning (Artificial Neural Networks, LSTMs, RNNs). This service will empower system conditions optimisation, such as the s2.4 Indoor Environment Quality (IEQ) Thermal Comfort Enhanced Facility Heating Management service, enhancing energy efficiency and enabling more effective building management for operational and maintenance activities. These models will be tailored to specific contexts, such as forecasting onsite energy production for the next day's energy supply or weekly heating demand for O&M activities.

3.3 TUC-3 – Monitor, Assessment, Prediction and Optimisation services

- **TUC-3.1 – Automated lighting control for improved visual comfort**

This TUC supports the Smart Lighting Control for Enhanced Visual Comfort service (s2.1), which aims to enhance lighting control by adapting usage to building and environmental conditions. For that, this TUC needs to enable automated management of lighting systems based on submetering devices tracking energy usage per zone and end-use, lighting levels in each zone, occupancy levels, preset lighting preferences and user feedback. Control options will be facilitated through dimmer switches for remote operation. The user feedback will be based on signals such as manual switching and satisfaction assessments via questionnaires. This ensures precise optimisation strategies tailored to building and user requirements.

- **TUC-3.2– Automated control for optimal IEQ (thermal/visual comfort and air quality)**

This TUC supports the IEQ Optimisation for Enhanced Comfort Service (s2.2), which aims to regulate the IEQ conditions in buildings to enhance occupants' comfort levels (including thermal comfort, well-being, etc.). For that, this TUC must consider individual and aggregated occupant behaviour models, energy profiling, and consumption demand patterns to align comfort improvement strategies with energy efficiency goals. Through this, it will learn user preferences implicitly to maximise comfort. Parameters such as pollutant concentration, humidity, and internal/external temperatures will inform the identification of internal conditions, along with user preference criteria and predefined comfort targets. This TUC will recommend optimising user comfort, suggesting actions such as adjusting windows, blinds, lighting, temperature, Heating, Ventilation and Air Conditioning (HVAC) settings, and managing air pollutant sources. Additionally, it should consider cost-effectiveness, weighing the deployment costs against energy and cost savings, along with potential flexibility remuneration.

- **TUC-3.3 – Automated control for optimal RES generation and distribution**

This TUC supports the RES-enhanced heating and cooling energy generation and distribution service (s2.3), which aims to optimise and prioritise the use of RES in managing buildings' heating and cooling systems. This TUC must regulate various energy assets, including heat pumps, heat recovery units, and smart thermostats, to minimise overall energy consumption while ensuring occupants' thermal

comfort. It will use real-time data, consumption patterns, user feedback, and optimisation objectives to offer tailored solutions. Data sources shall include energy consumption metrics (e.g., district heating, domestic hot water, ground source heat pump), energy generation data (e.g., solar photovoltaic), environmental conditions (temperature, humidity), Electric Vehicle (EV) charging data, occupancy levels and user feedback via switch activations or adjustments to energy assets.

- **TUC-3.4 – Automated heating system control for optimal thermal comfort**

This TUC supports the IEQ Thermal Comfort Enhanced Facility Heating Management service (s2.4), which aims to manage heating facilities to implement optimal strategies for maintaining occupants' comfort levels and meeting heating demand. For that, this TUC must assist in integrating weather forecasts and thermal comfort parameters to identify areas for minimising energy consumption and improving energy efficiency. It also should support detecting abnormal electricity consumption to prevent outages or abnormal operation of these assets, allowing for optimal electrical or thermal load management, among other benefits.

- **TUC-3.5 – Automated control for building flexibility management**

This TUC supports two services: the Flexibility Services for Building Occupants (s2.5) and the Building Flexibility Management for Grid Congestion through Peak Shaving and Load Shifting (s2.6). The Flexibility Services for Building Occupants (s2.5) aims to provide flexibility solutions by incorporating RES and storage capabilities into existing building infrastructure. For that, this TUC should encompass strategies for integrating Energy Storage Systems (ESS) (including thermal and electrical storage using second-life batteries), EVs, and Photovoltaic (PV) systems, along with the flexibility potential from electrical appliances within buildings. The outcomes of s2.5 will offer energy flexibility strategies and management at the building level, facilitating the effective integration of RES and EV capabilities during operational periods.

The Building Flexibility Management for Grid Congestion service (s2.6) aims to deliver models for demand flexibility management to flatten load curves by mitigating peaks and shifting them to off-peak hours within residential buildings and grids. This TUC should leverage data granularities to develop optimal models that provide accurate predictions for different forecasting horizons (including day-ahead, week-ahead, and month-ahead). In instances of RES and EV availability, control strategies will be devised to identify peak loads in advance, enabling the full use of ESS for charging during low-demand periods and discharging during high-demand periods, as well as leveraging EVs as an energy source during peak demand.

3.4 TUC-4 – Digital Building Logbook

- **TUC-4.1 – Building data management and integration**

This TUC supports the DBL, which serves as a central repository to store and display various types of information regarding the buildings, including general, administrative, construction, energy performance, operational, and intelligent data. Acting as both an input provider and a data consumer, this TUC's main purpose is to supply static data from Building Information Models (BIM) and fulfil requests for dynamic data from buildings through BuildON's main software architecture. This should enable automated updates, integration with external databases and sources, and the management of notifications related to consumption, maintenance tasks, and alerts.

4 Core Functionalities

This section outlines the core functionalities required to support the specified TUCs, categorised into internal and external functionalities (Figure 3). The internal functionalities include the functions related to configuring interactions among components within the architecture. The external functionalities focus on the data exchange (reading and writing data) as required by the MAPO services and DBL (both referred to as “services” throughout the rest of this report) via the architecture.

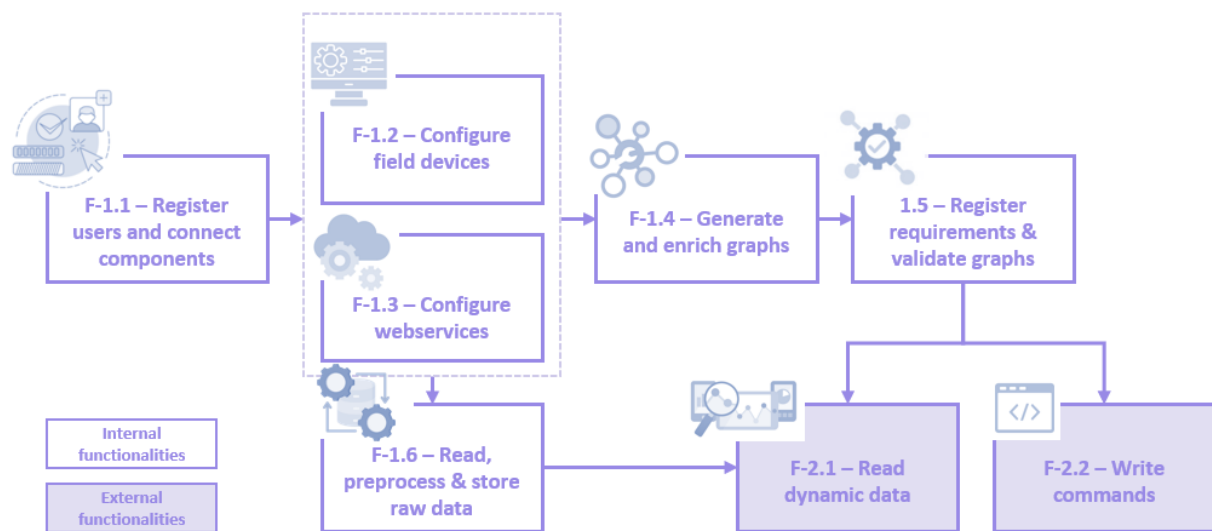


Figure 3 – BuildON internal and external functionalities

4.1 F-1 – Internal functionalities

- **F-1.1 – Register users and connect components**

Description: the architecture needs to include access control mechanisms, which assign roles, permissions, and credentials to each registered user and client (architecture’s core component and services). Additionally, it should integrate all components to ensure a seamless data and command exchange within the platform.

Supported TUCs: All

- **F-1.2 – Configure field devices**

Description: Once devices are configured internally and capable of reading and writing data, their representation, characteristics, and methods for accessing and processing their information should be defined in a standardised manner. This will streamline their discovery and use by higher-level components and services. To achieve this, devices’ information, including data point types, properties, units, overridden capabilities, and context (such as location and related equipment), should be specified using configuration files (or annotated point lists) that adhere to schemas such as the Web of Things and Haystack.

Supported TUCs: All

- **F-1.3 – Configure web services**

Description: All relevant web service interfaces for internal and external communication should be configured and made available to process requests. To facilitate their discovery and use by higher-level components and services, they should also follow the procedures described in F1.2 to specify their information and access means in a standardised way.

Supported TUCs: All

- **F-1.4 – Generate and enrich graphs**

Description: the architecture shall be designed to incorporate components and processes that facilitate the creation of knowledge graphs using standard ontologies to uniformly represent the buildings and systems of the project's pilots. These components and processes should enable the population of the graphs using metadata sourced from standardised configuration files coming from functionalities F-1.3 and F-1.4 or other static data. Available static data sources may include BIM/IFC models, DBLs, and control flow diagrams.

Supported TUCs: All

- **F-1.5 – Register services requirements and validate graphs**

Description: once the knowledge graphs are instantiated or updated, modelling the diverse systems and devices within the buildings along with their context, data completeness tools are used to verify the availability of all required data for each service. To facilitate this, each service is required to document its data needs. These requirements should be translated using annotations compliant with the Shape Constraint Language (SHACL)⁷, enabling architecture components to attest their validation results automatically. Once users validate and select a service, its endpoints and messaging mechanisms must be configured based on specified data and other requirements, such as execution frequency/schedule. Initialisation will involve obtaining external references through common semantic queries from the knowledge graphs.

Supported TUCs: All

- **F-1.6 – Read, pre-process and store raw data**

Description: the platform shall be able to read the raw data coming from field devices and web services, and perform preprocessing and data quality checks, including missing value handling, outlier detection, duplicate removal and transmission frequency adjustment. This should ensure the data quality necessary to support lightweight services hosted at the edge as well as minimise the volume of transmitted data for storage and streaming purposes. The architecture shall also be able to communicate with the project's database and store the data as needed.

Supported TUCs: All

4.2 F-2 – External functionalities

- **F-2.1 – Read dynamic data**

Description: every registered service within the project must be able to access the required dynamic data, whether real-time or historical data, from field devices or web services. The architecture should

⁷ <https://www.w3.org/TR/shacl/>

be designed to offer a unified access point for all services through a universal Application Programming Interface (API). This API, supported by the capabilities offered by the knowledge graphs (e.g., common query mechanisms), will serve as a standardised means for data access, effectively abstracting the complexities of communicating with heterogeneous data sources and ad-hoc metadata naming conventions.

Supported TUCs: All

- **F-2.2 – Write commands**

Description: the architecture must facilitate two-way communication mechanisms, empowering registered services to control field devices effectively. Similar to the read dynamic data functionality, this mechanism should be standardised, simplifying communication complexities through a well-defined and documented API.

Supported TUCs: Only TUC-3 – Monitoring, Assessment, Prediction and Optimisation services.

5 Conceptual Architecture

The BuildON conceptual architecture, as specified in the Description of Action (DoA), sets the point of departure for developing the BuildON reference architecture. In this section, we revisit this conceptual architecture and consider refinements in line with the Technical Use cases presented above. The BuildON IoT-Edge interoperability framework is a set of software components that guarantee the generation and exposition of a unified and abstracted representation of the building to higher-layer services. It comprises a set of core components deployed both on edge and in the cloud to access the diverse energy systems, subsystems and devices, as well as other IT systems (e.g., existing databases for occupancy, weather stations, etc.) available in the building and external data (e.g., weather forecast, energy prices, etc.). The architecture offers this information to higher-layer services using a common data representation and a standard communication procedure that can hide the heterogeneity and complexities associated with bi-directional communication with building assets. This architecture acts as a middleware, allowing seamless exchange between higher-layer services and the diverse assets at the field level. The core components within the BuildON architecture are illustrated in Figure 4 and include: i) the Edge Domain Controller (DC), ii) the Cloud Component (CC), iii) the Data Access and Control Manager (DACM), iv) the BuildON Data Repository, and v) the Universal Building API.

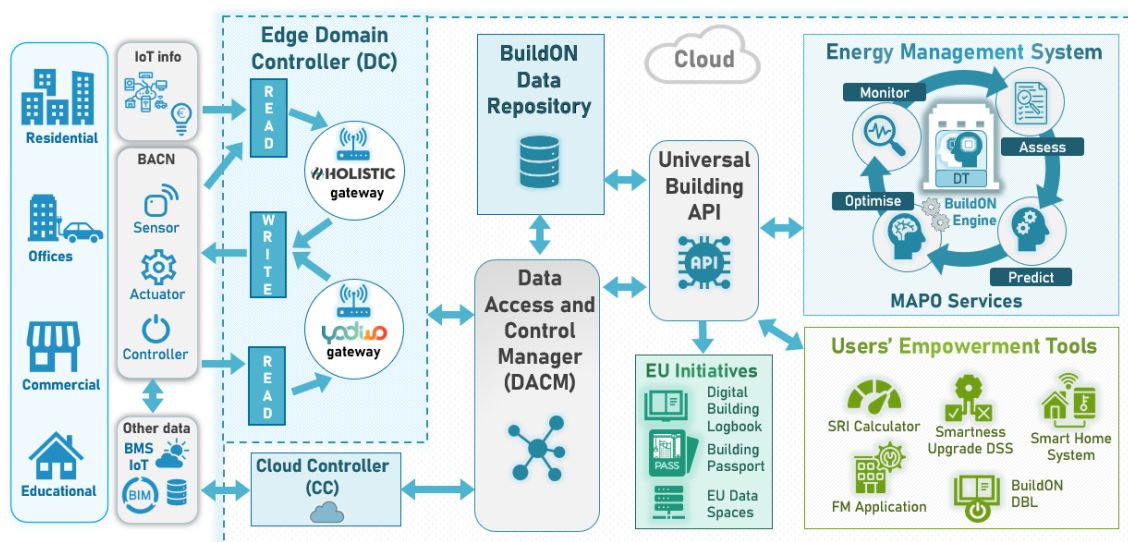


Figure 4 – BuildON conceptual architecture

5.1 Edge Domain Controller (DC)

The edge Domain Controller (DC) is deployed on-site and near the monitored assets. This component provides low-level access to the physical assets and devices, e.g., by connecting to the BEMS (or Building Automation and Control Network - BACN) for data retrieval, actuation triggering, local data buffering, or BEMS discovery (i.e., scanning for sensors and actuators). In the case of very large or distributed assets (e.g., operating sites of a company spread over a large geographical area, as in the CORDIA pilot) or different management/ownership domains, multiple DCs can be deployed. In this case, all DCs are managed by a single DACM instance, allowing simultaneous monitoring and control of assets under different DCs. In addition, the DC supports auto-configuration capabilities to discover

and interface with BEMS and communicate with legacy building systems. The edge DC provides such functionalities through an extensible driver architecture which makes it possible to use several field-level communication protocols (BACnet⁸, Modbus⁹, LonWorks¹⁰, open platform communication¹¹, KNX¹²) and support providing to the DACM a common data points specification via configuration files that are generated at the DC level. The DCs enable bidirectional communication (read-write) with building assets and provide a first abstraction layer for interacting with these components.

5.2 Cloud Controller (CC)

The Cloud Controller (CC) component is deployed in the cloud, providing access components to interface with IT web services, such as third-party APIs. The APIs can provide access to resources such as electricity prices, weather data, and IoT platforms. Similarly to the DC, multiple CCs (or a combination of DCs and CCs) can be deployed per pilot while simultaneously monitored and managed by a single DACM instance. In addition, the CC supports auto-configuration capabilities to discover and interface with IoT devices and external data sources communicated via diverse API endpoints (where relevant APIs support this). This will be supported by data connectors and a common data points specification via configuration files that are generated at the CC level and sent to the DACM.

5.3 Data Access and Control Manager (DACM)

The Data Access and Control Manager (DACM) is deployed on the cloud, supervising one or multiple edge DCs and CCs. In addition to assuring two-way communication with the DCs and CCs (to obtain/read data from the DCs and CCs and to send commands/write to the DCs and CCs), the role of this component is to provide centralised high-level functionalities of the architecture. For example, providing connectors to elements outside the IoT frameworks, such as the MAPO services (through the Universal Building API), interfaces with the Data Repository or connectors to external cloud-based data sources (e.g., weather forecast through the CCs). In the same way that one DACM can control multiple DCs in a building, several services can be connected to a DACM via the API to get historical and real-time data and send control actions. The DACM supports the discovery of DCs and CCs, auto-configuration, and synchronisation of field-level devices, as well as the security and privacy aspects related to data access and storage.

5.4 BuildON Data Repository

The BuildON Data Repository offers a Common Data Environment for static and historical dynamic data and other available building information (e.g., from DBLs, Renovation Passports or elsewhere). This component ensures the persistence and security of building data and exposes them through the Universal Building API to higher-layer services. Historical data is particularly useful to AI-based training

⁸ <https://bacnet.org/>

⁹ <https://modbus.org/>

¹⁰ <https://www.lonmark.org/>

¹¹ <https://opcfoundation.org/>

¹² <https://www.knx.org/>

services or applications, such as benchmarking. In addition, this component uses knowledge graphs describing the buildings and their assets uniformly to minimise the effort and costs required to onboard new higher-layer services in different buildings. This should support a more automated mapping to both historical and real-time data access offered through the common queries of the knowledge graphs. Data quality is essential for both dynamic and static data. To ensure that required data are available, especially given the complex and multiple ETL process to integrate heterogeneous data in forming the completed knowledge graph, the Data Repository implements a generic and user-configurable model-checking framework.

5.5 Universal API

The Universal Building API offers a standard and well-defined interface to access all data, "read" data, and "write" commands through an abstraction of the building – that is, offering the whole Building as a Service. The high-level services should use this API to access historical, linked, enriched, and federated information in the Data Repository and to read real-time and send, through the DACM, control commands. A detailed open API specification will be made available (using Swagger¹³ or Postman¹⁴) for the API's methods. This specification will be published as an openly available public specification for further development and reuse by third parties, attracting service developers to provide value-adding applications. This way, analytics, services, and third-party applications can be developed once and re-used on multiple sites, offering the path for creating an application ecosystem and "app store" for MAPO services.

¹³ <https://swagger.io/>

¹⁴ <https://modbus.org/>

6 Functionalities Sequence Diagrams

This section delineates the operational flow and interactions among BuildON's core components to enable the proposed functionalities (referred to as FX.X as in section 4). These flows are illustrated through UML sequence diagrams, which comprehensively depict the main actors and components, showcasing their interactions and internal functionalities. In the UMLs, each interface enabling seamless interactions among components is uniquely identified (IX.X) for reference in the following sections, which provide more detailed interface specifications.

6.1 F-1 – Internal functionalities

• F-1.1 – Register users and connect components

In this internal functionality, BuildON proposes a comprehensive workflow for managing user identities, client applications, roles, and permissions. This functionality relies on the Universal API component, incorporating an Identity Provider subcomponent to ensure robust identity and access management through role-based access control. Due to its open-source and extensive integration capabilities, Keycloak¹⁵ has been chosen for such a purpose.

The execution of this functionality follows a sequence of steps outlined in Figure 5:

1. **Administrator setup:** an administrator actor (BuildON Admin) defines other actors and their respective roles, each representing specific sets of permissions.
2. **User registration:** BuildON users are registered in Keycloak by the BuildON Admin actor through an administrative interface or programmatically via Keycloak's REST API.
3. **Role assignment:** users can be assigned one or more roles, granting them the associated permissions. The responsibility for assigning suitable roles and permissions lies with actors associated with each core component:
 - a. BuildON Admin manages the Repository component.
 - b. BuildON systems integrators handle the DACM component.
 - c. DC and CC providers oversee the DC and CC components, respectively.
 - d. Service providers are responsible for the MAPO services and DBL. Although MAPO services and DBL are not core components, they interact with other components involving related actors.
4. **Client registration:** clients in Keycloak represent BuildON core components and services that need the authorisation to interact with each other. As for the users, clients are registered in Keycloak by the BuildON Admin actor through an administrative interface or programmatically via Keycloak's REST API.
5. **Client roles:** similar to user roles, clients can also have roles defined within Keycloak. These roles represent specific permissions for the interaction among components and services.
6. **Credentials:** Keycloak supports various credential types for user authentication, including passwords, one-time passwords, and client certificates. These credentials are used during the authentication process to verify the identity and permissions of the user and clients.

¹⁵ <https://www.keycloak.org/>

7. **Connect clients:** Once clients (components and services) are registered and have their client certificates set up and roles assigned, they can communicate securely using their certificates for authentication. This could involve exchanging data over secure channels such as Hypertext Transfer Protocol Secure (HTTPS), where each client presents its certificate for authentication, and the server (or other clients) verify the certificate to ensure the authenticity of the connection. In addition to client certificates, clients can obtain access tokens from Keycloak, which they can then use to authenticate and access protected resources. This adds an extra layer of security to the authentication process.

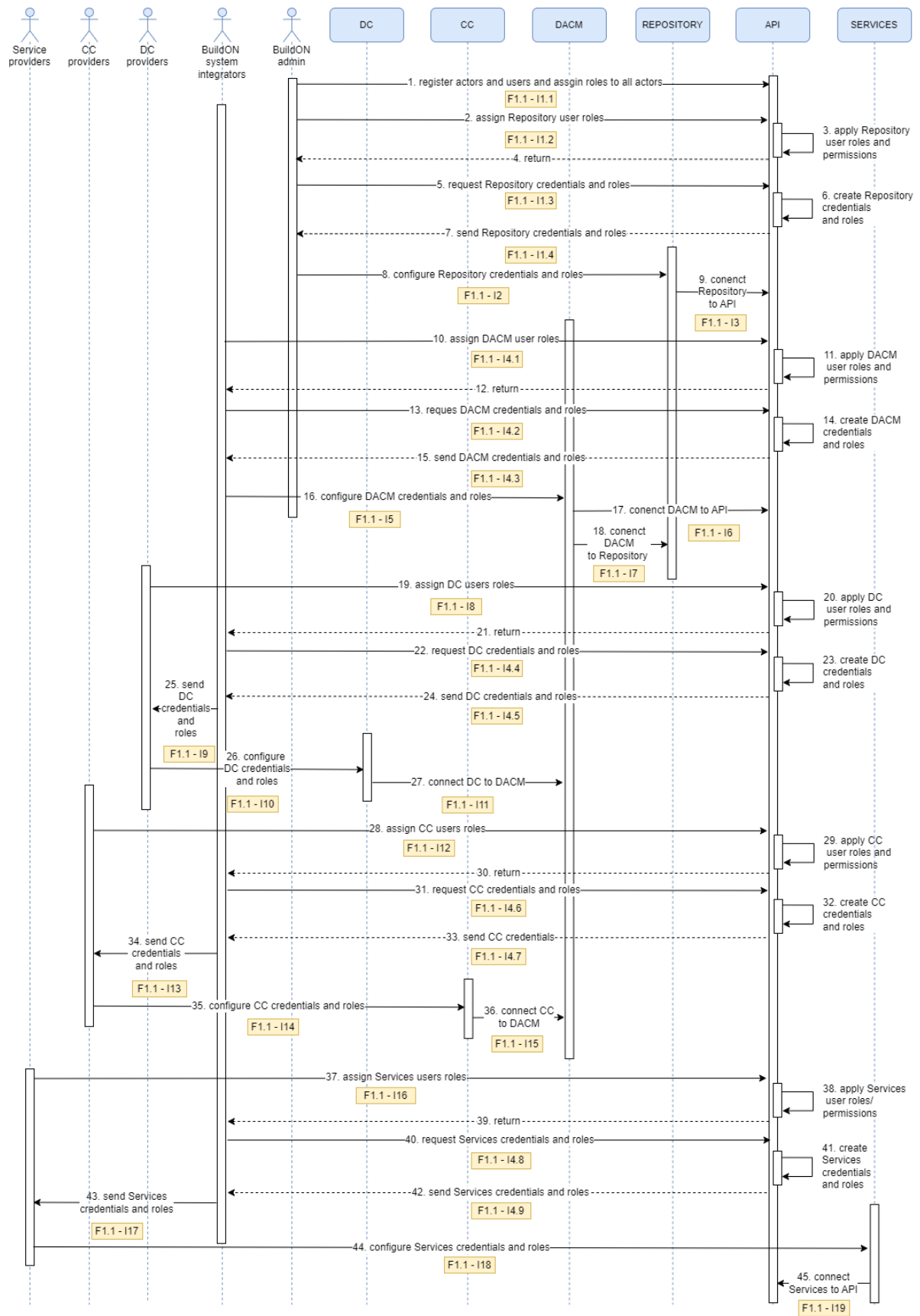


Figure 5 – Sequence diagram of F-1.1 (register users and connect components)

- **F-1.2 – Configure field devices**

This internal functionality involves setting up field devices within DC and DACM components. The core of this functionality is to provide a uniform description of devices' data points. This uniformity is crucial due to these descriptions' bespoke and often ambiguous nature, making it difficult to identify, map, and integrate them with higher-layer software architecture components and services.

The execution of this functionality follows a sequence of steps outlined in Figure 6:

1. **DC installation and configuration:** the BEMS contractors and DC provider actors initiate this functionality by installing and configuring new devices on the edge. The configuration includes tasks such as populating internal data models and setting up internal services (e.g., short-term data persistence to recover from loss of connectivity).
2. **Data uniformisation:** the DC implements an internal process to ensure consistent descriptions of devices and their data points across various gateways. This involves transforming internal data models into configuration files following a particular schema, such as Haystack or Web of Things. These configuration files resemble annotated lists of points. They include descriptions detailing various aspects such as data point types, properties, units, available actions, overridable points, information about their location, abstraction, topology, and conditions for writing a value (e.g., delay) to protect controllable assets.
3. **File validation:** the DACM receives configuration files from the DC and validates them against minimum requirements according to the schema they follow.
4. **File usability for graph generation:** once validated, these configuration files are shared with the Repository component, which uses them to generate knowledge graphs; this generation process happens in multiple steps, as DCs notify of creation or updated configurations. In the process, the whole part of the knowledge graph might need to be re-generated.
5. **DACM configuration:** using the configuration files containing devices' data point descriptions/specifications, field devices are configured within messaging mechanisms (broker and API server), along with other services such as logging and protective mechanisms for data writing.

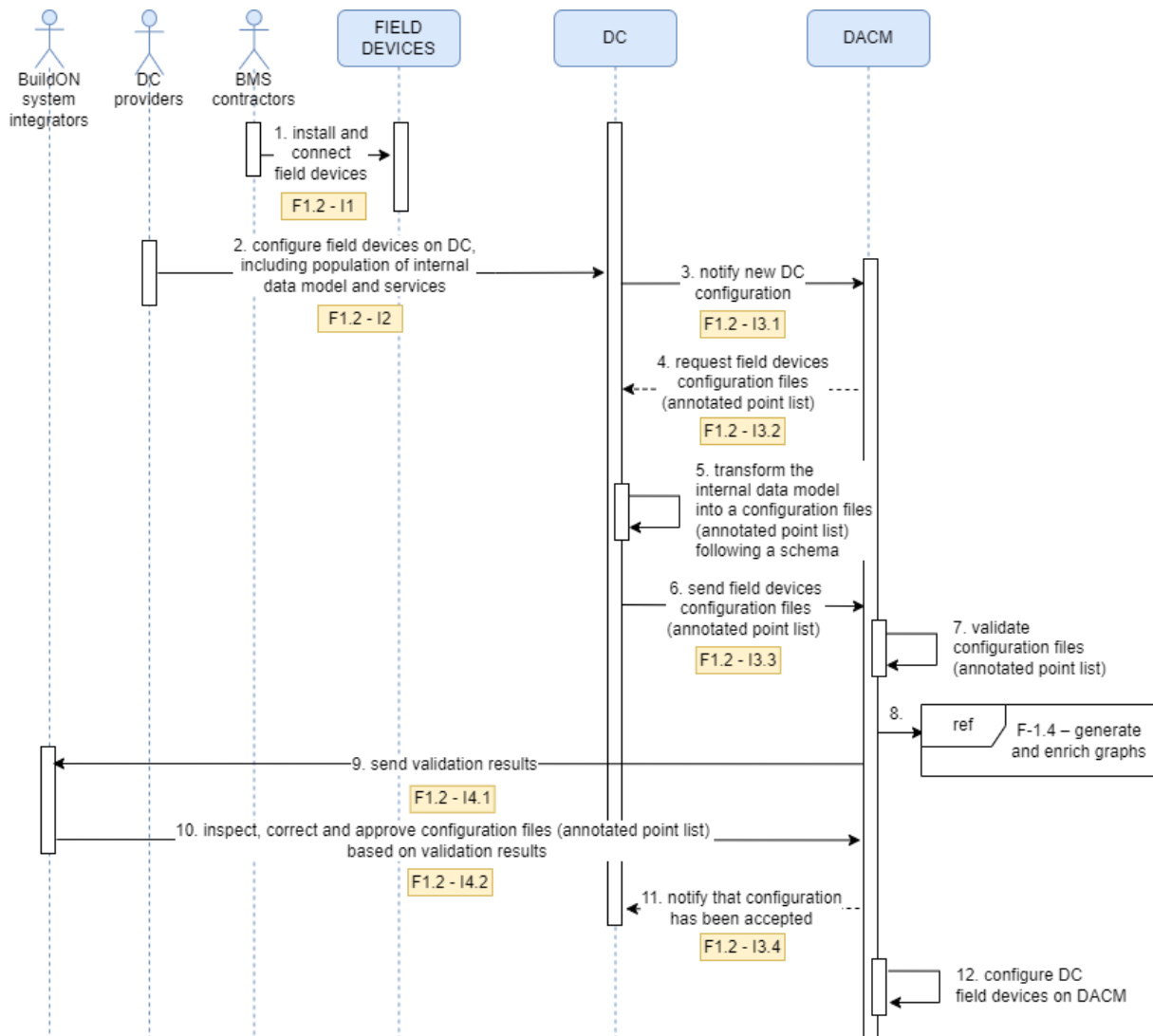


Figure 6 – Sequence diagram of F-1.2 (configure field devices)

• F-1.3 – Configure web services

This internal functionality involves setting up web services within the CC components. These web services may provide API endpoints for data access and exchange with IoT platforms, weather, and utility data providers. Similar to the configuration of the field devices, the core of this functionality is to provide a uniform description of the web services' data points. This consistency is also critical because these platforms and services often have bespoke and ambiguous descriptions, which hinders their identification, mapping, and integration with higher-layer components of the software architecture and the services.

The execution of this functionality follows a sequence of steps outlined in Figure 7:

1. **Connection and configuration:** CC provider actors initiate this functionality by connecting and configuring new web services on the cloud. The configuration includes tasks such as populating internal data models and setting up internal services.
2. **Data uniformisation:** the CC implements an internal process to ensure consistent descriptions of web services data points across various platforms. This also involves transforming internal data models into configuration files (annotated lists of points) following schemas, such as Haystack or Web of Things.

3. **File validation:** the DACM receives configuration files from the CC and validates them against minimum requirements according to the schema they follow (syntactic interoperability).
4. **File usability for graph generation:** once validated, these configuration files are shared with the Repository component, which uses them to generate knowledge graphs.
5. **DACM configuration:** using the configuration files containing web services data point descriptions/specifications, web services are configured within messaging mechanisms (API server) and other services such as logging and protective measures for data writing.

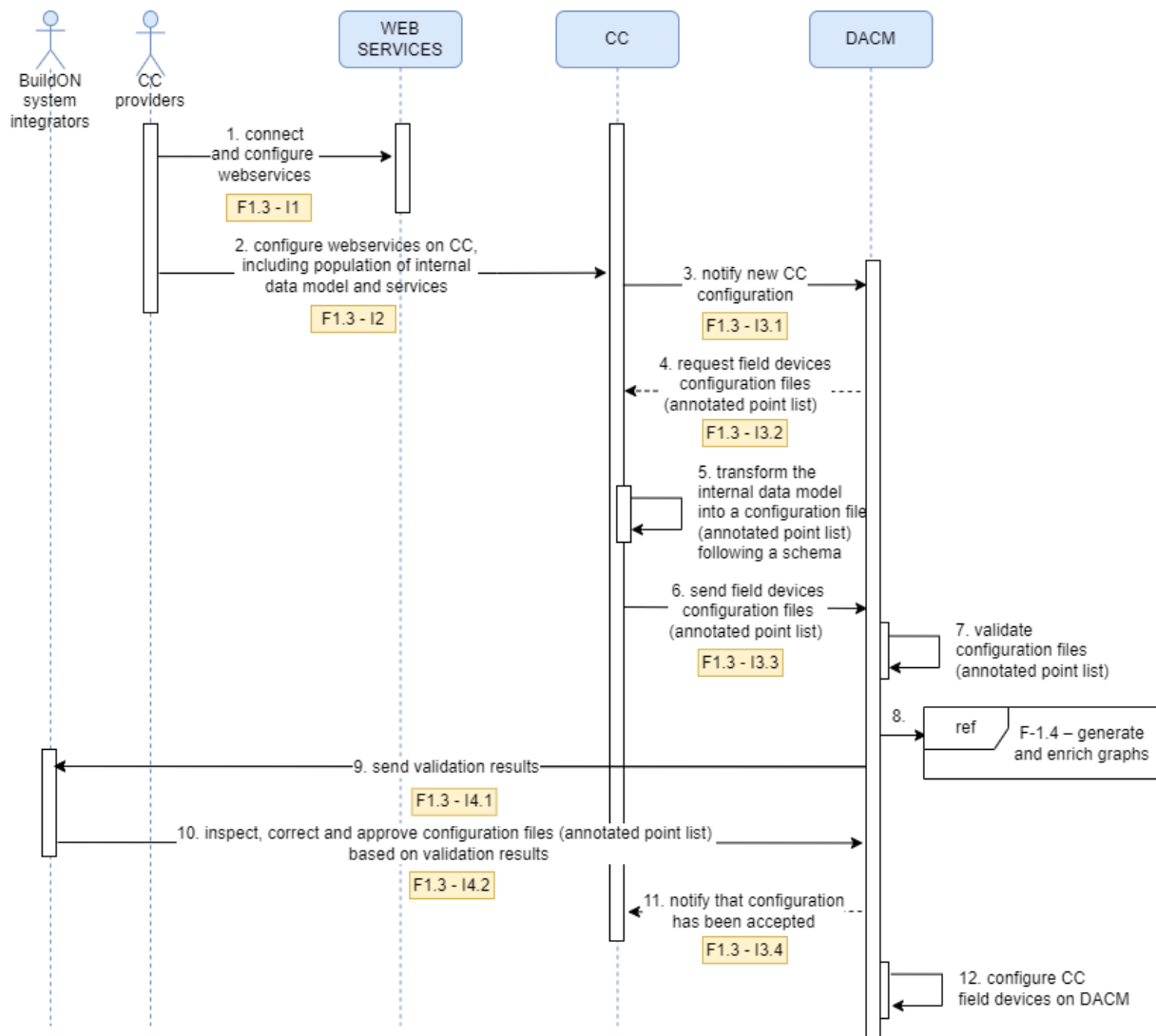


Figure 7 – Sequence diagram of F-1.3 (configure web services)

- **F-1.4 – Generate and enrich graphs**

This internal functionality captures metadata from static data files such as IFC models, BEMS technical specifications, and configuration files received through the DACM from the DCs and CCs. This metadata is used to instantiate knowledge graphs, which employ ontologies' logic notations and common concepts to enable data from various sources to share context and meaning, enhancing their interoperability and discoverability. This helps represent building systems uniformly, allowing different buildings to be abstracted to higher-layer components of the software architecture and the services.

The execution of this functionality follows a sequence of steps outlined in Figure 8:

1. **File uploads/exchange and processing:** this functionality begins with storing static files uploaded by users or configuration files sent from DC and CC via DACM. Due to their variability, the static files undergo checking and annotating tasks to facilitate their use for generating knowledge graphs.
2. **Isolated knowledge graph generation:** using Extract, Transform, and Load (ETL) tools, partial knowledge graphs are generated in parallel. These graphs are derived from annotated static files and configuration files. The ETL tools are tailored to handle specific data formats and structures/models (e.g., IFC, CSV, etc.). The creation of the graphs relies on the concepts outlined in the chosen underlying ontology. Selecting the appropriate ontology depends on the data needs of services and the extent of coverage provided by established ontologies such as Brick, BOT, REC, and the upcoming ASHRAE 223P standard.
3. **Knowledge graph validation:** validation processes are employed to ensure that the generated knowledge graphs comply with their underlying ontology (or set of them).
4. **Knowledge graph merger:** upon isolated generation of all graphs from these parallel processes, these graphs are merged into a unified graph.

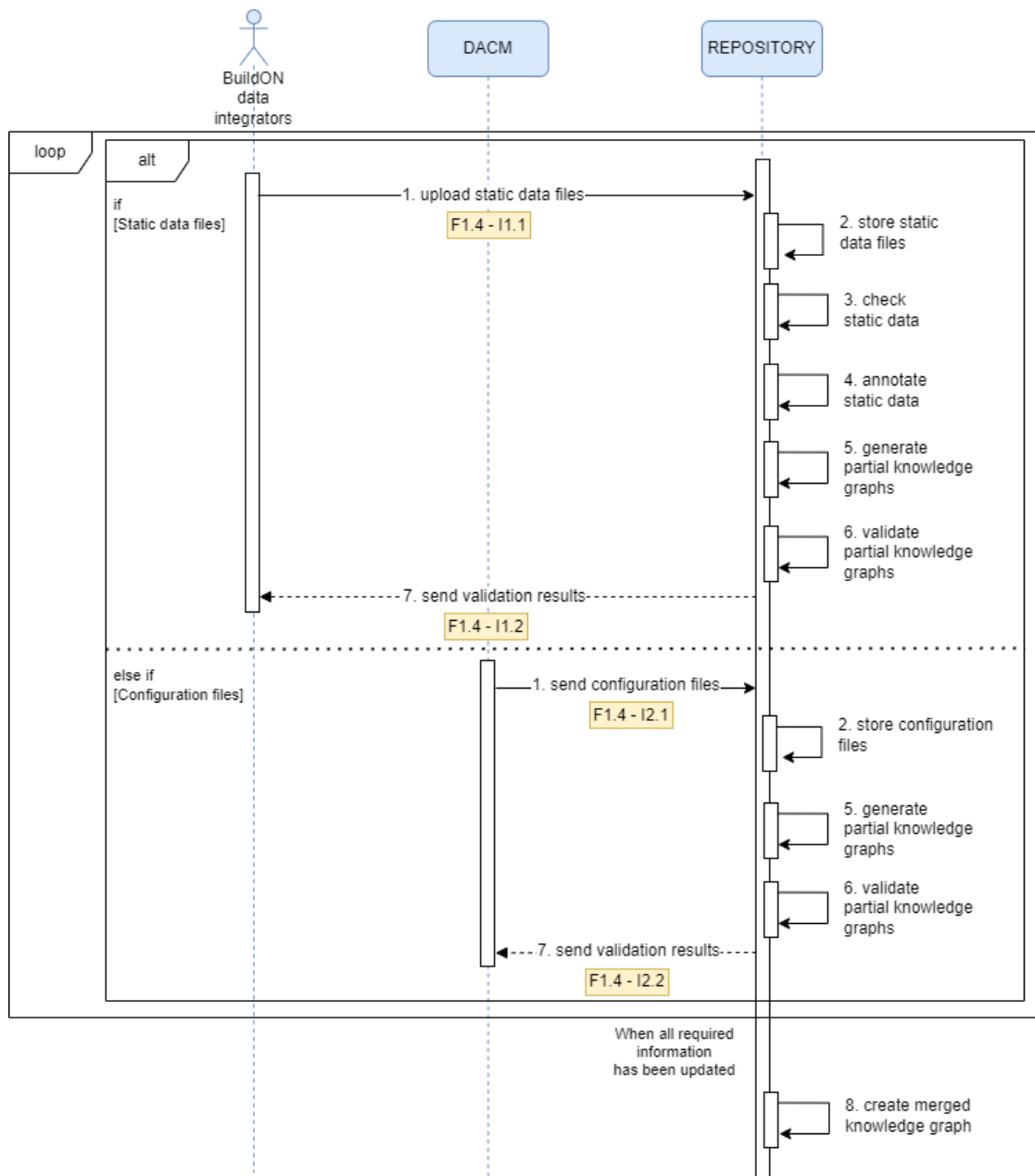


Figure 8 – Sequence diagram of F-1.4 (generate and enrich graphs)

- **F-1.5 – Register services requirements and validate graphs**

In this internal functionality, service requirements are registered for two primary purposes. First, to support data completeness mechanisms that verify the availability of all necessary data for each service in each pilot building. Service requirements are translated using appropriate annotations that enable automatic validation of data against the building's knowledge graphs. Second, to configure endpoints and data access methods for various valid services across different buildings.

The execution of this functionality follows a sequence of steps outlined in Figure 9:

1. **Requirements registration:** service providers initiate this functionality by identifying the data needs for their services. These include required data points, contextual metadata (e.g., location or equipment), and additional details such as execution schedule/frequency and whether real-time or historical data is needed.

2. **SHACL constraints specification:** once registered and compiled in the API, required data points and contextual metadata are translated into SHACL constraints. These constraints can be used to validate whether individual building graphs semantically support the services' data and metadata requirements.
3. **Selection of valid services:** validation results are sent to BuildON system integrators, who can choose from validated services available for each building.
4. **Data access configuration:** after services are selected, the API, along with internal repository ETLs, defines API endpoint requests and sub-pub topics that allow the services to access the data across different buildings. These configurations are sent to service providers for implementation. This involves obtaining external references from direct measurement and control points within buildings using semantic queries of their knowledge graphs and employing other methods to obtain data, such as indirect computations based on multiple and alternative observations.
5. **Priority logic:** once all services are registered and listed in the API, an internal prioritisation logic is defined to orchestrate their operation, mainly when multiple services are applied to the same buildings.

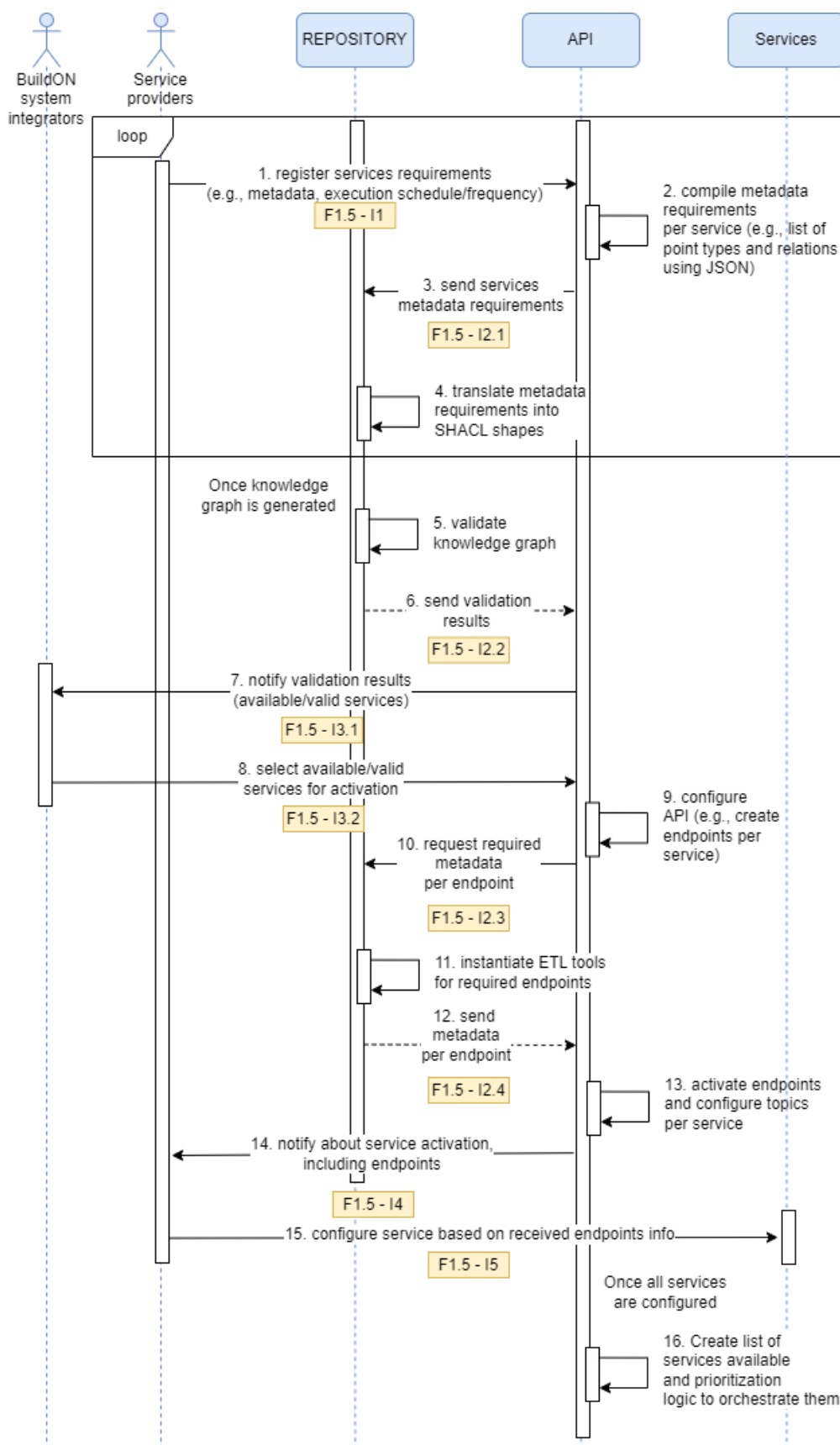


Figure 9 – Sequence diagram of F-1.5 (register services requirements and validate graphs)

- **F-1.6 – Read, preprocess and store raw data**

In this internal functionality, BuildON proposes a comprehensive workflow to manage the ingestion, processing, and storage of data from field devices and web services. This process entails internal configuration and, when needed, implements data quality mechanisms to support service deployment at the edge. Additionally, it incorporates temporary storage solutions within DC to ensure buildings' reliable and secure operation when services are deployed in cloud-based environments.

The execution of this functionality follows a sequence of steps outlined in Figure 10:

1. **DC and CC internal configuration:** this functionality is initiated by setting internal scheduling of tasks, event triggers administration, and alarm configurations in both DC and CC. Those are needed to enable streamlined routine activities and rapid detection of faults or abnormalities within DC and CC.
2. **DC and CC raw data retrieval:** depending on the supported communication protocol or messaging mechanism of field devices and web services, this step employs either a pull/send or push process to transfer data to the DC and CC.
3. **DC and CC data processing:** upon receiving the data, DC and CC execute preprocessing and data quality checks when needed to support services deployed at the edge. These may include unit conversion, handling missing values, outlier detection, duplicate removal, and adjusting transmission frequency.
4. **DC short-term storage:** DC includes subcomponents for short-term data storage, which is crucial for managing periods of faulty connections when buildings have services deployed in the cloud. This allows reliable operation when services are disconnected until connectivity is restored.
5. **DACM data processing:** upon receiving data from DC and CC, the DACM conducts further data processing and quality checks complementary to those performed at the DC and CC levels.
6. **DACM data log and alarms:** DACM also records data logging events and sends proactive notifications, known as alarms, triggered by predefined thresholds or conditions. These help troubleshoot, identify trends, and capture potential issues or deviations from expected behaviour.
7. **Data storage:** the dynamic data is then sent to and stored within purpose-built databases in the Data Repository component.

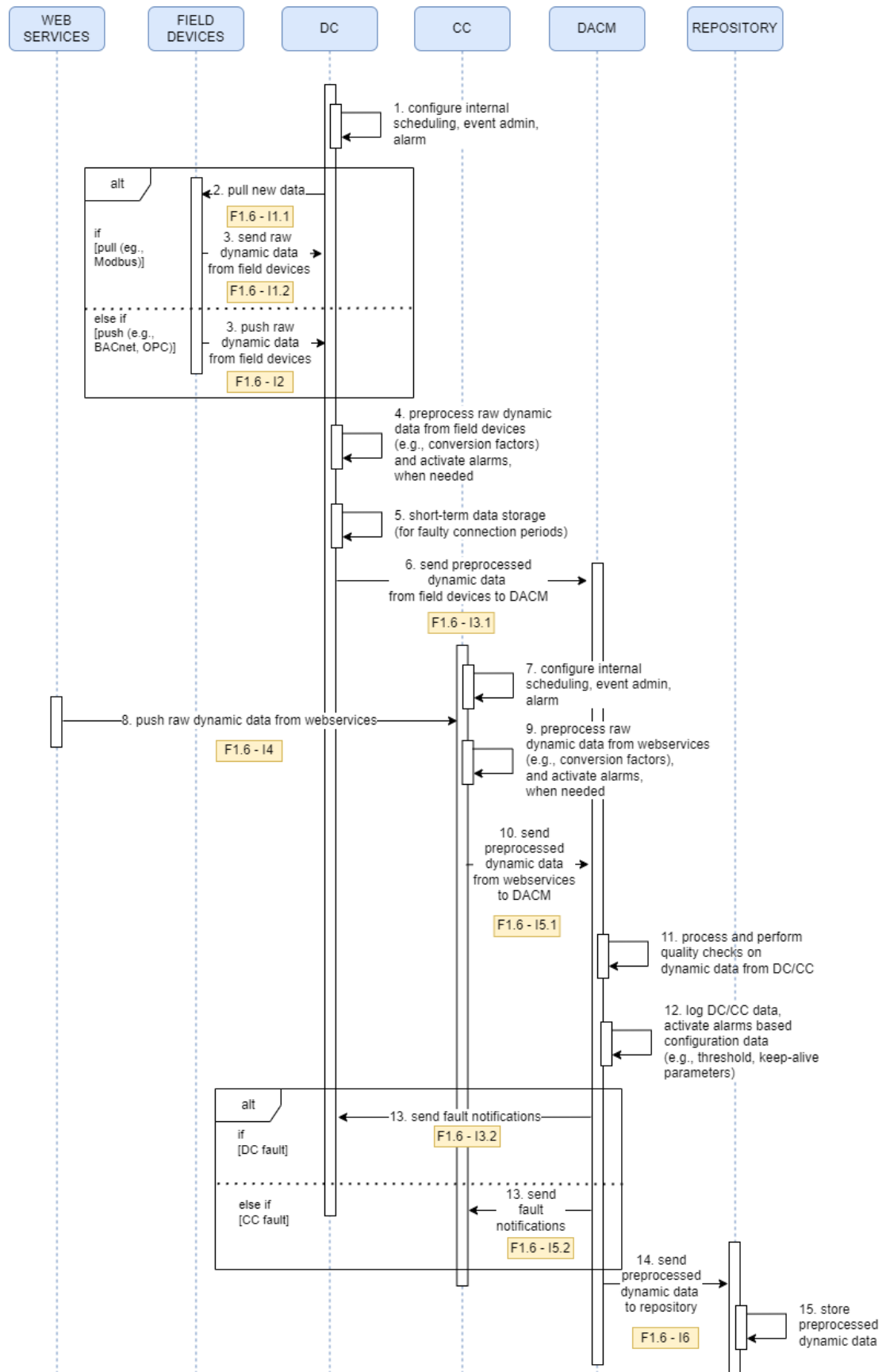


Figure 10 – Sequence diagram of F-1.6 (read, preprocess and store raw data)

6.2 F-2 – External functionalities

- **F-2.1 – Read dynamic data**

Higher-layer services use API requests to access dynamic data in this external functionality. This data may either be historical, requiring interaction with the Data Repository, or real-time, necessitating interaction with the DACM. The endpoint configuration and messaging subscription per service is facilitated by Internal Functionality F1.5, which allows the services to be instantiated and configured based on abstract and uniform access of the buildings' data points based on their knowledge graphs.

The execution of this functionality follows a sequence of steps outlined in Figure 11:

1. **Authentication:** the services initiate this functionality with an authentication process to verify their identity and permissions. Upon confirmation, if the requested dynamic data aligns with the service's authorised roles, the service will proceed to request the data via the API.
2. **Data retrieval request:** the data retrieval process occurs through two methods:
 - a. **Direct API requests:** the service makes direct API requests per endpoint. The request is sent to the DACM for real-time data, which then returns the data to the API. Alternatively, the request is directed to the data repository for historical data, which then returns the data to the API.
 - b. **Message broker subscription:** services may also subscribe to specific topics within the DACM's message broker via the API. Upon new data availability, the DACM pushes the data to API.
3. **Data distribution:** the API receives the requested data and sends it to the respective higher-layer services.

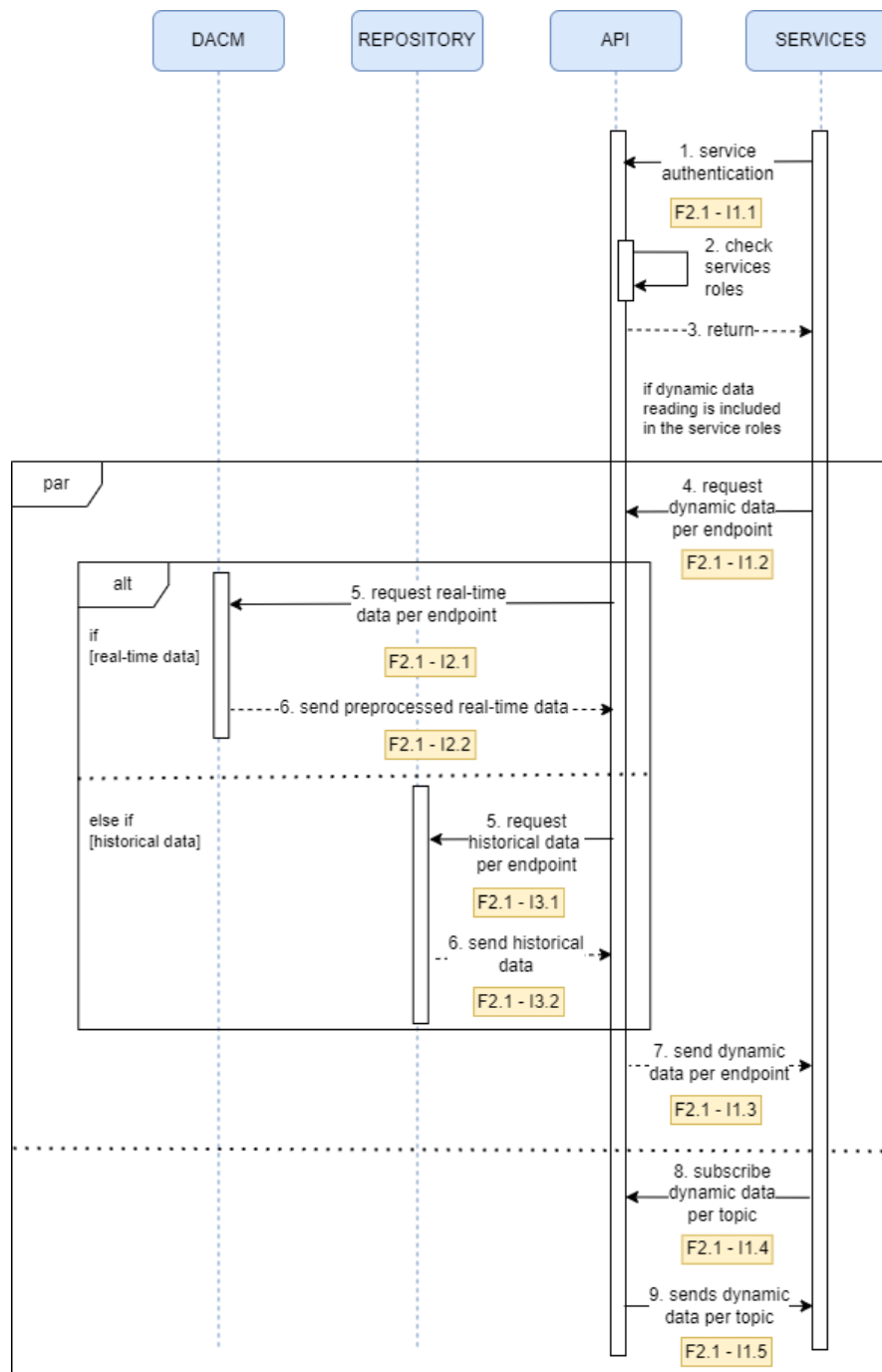


Figure 11 – Sequence diagram of F-2.1 (read dynamic data)

• F-2.2 – Write commands

Higher-layer services use API requests to generate new control commands in this external functionality. These commands undergo evaluation at two levels. First, an orchestrating service manages command prioritisation, especially in scenarios where multiple services coexist within the same building and need to control similar parameters for different purposes. Second, each command is scrutinised to ensure validity, preventing issues such as values falling outside acceptable ranges.

The execution of this functionality follows a sequence of steps outlined in Figure 12:

1. **Authentication:** higher-layer services initiate this functionality with an authentication process to verify their identity and permissions. Upon confirmation, the service will request data

writing via the API if the requested writing commands align with the service's authorised roles.

2. **Command writing request and priority logic:** the services send requests to the API to write commands for each endpoint. Prior to sending the writing request to the DACM, the API checks the priority logic to confirm if a given service has the priority to issue such a command when orchestrating several services simultaneously.
3. **Command writing request and evaluation:** if a given service's command has priority, the DACM receives and evaluates the writing command. This evaluation is necessary to prevent potential issues such as commands with values out of range or sending conflicting commands in rapid succession, which may not be compatible with a device's hardware (e.g., sending both on and off commands in a short period).
4. **Command writing:** Upon approval, the DACM will send the commands to the DC and CC components. This can be achieved through publishing new values per topic via a broker or sending API requests.
5. **Writing notifications:** after successful writing, the low-level components notify the upper layers to confirm the control command has been successfully sent.

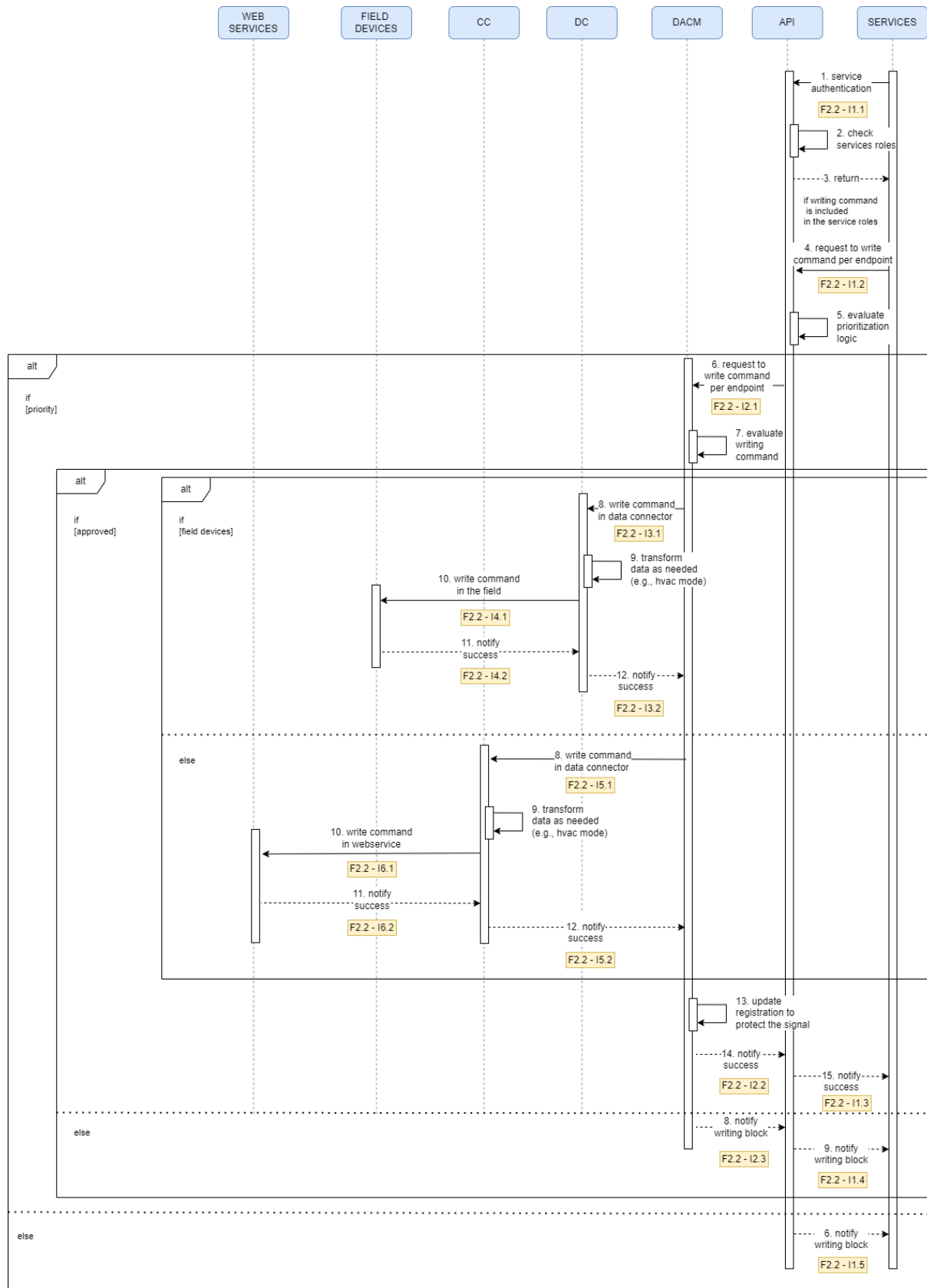


Figure 12 – Sequence diagram of F-2.2 (write commands)

7 Functional & Non-Functional Requirements

This section delineates functional and non-functional requirements for BuildON software architecture.

The common non-functional requirements for BuildON software architecture have been derived and extended as needed from the proposed Non-Functional Requirements (NFRs) outlined in WP2 D2.1. Meanwhile, the functional requirements for each core component have been derived from the sequence diagrams presented in the previous section, which support BuildON's envisaged functionalities.

7.1 BuildON non-functional requirements

Based on WP2 D2.1, the non-functional requirements pertinent to the software architecture encompass the following:

7.1.1 Modularity

The BuildON system must ensure that core components are designed to be modular and extensible, facilitating their reuse across various contexts. It must also guarantee the smooth and seamless ability to accommodate new devices, gateways, web services, and services without compromising performance; such services can be added and removed during operation without jeopardising the service provision (other than a short downtime of components). This is supported by the following NFRs from D2.1:

- NFR04: BuildON services should be able to operate in different systems.
- NFR13: BuildON should allow easy replication of its services.
- NFR15: BuildON should support extensibility of new functionalities in an easy to implement way.
- NFR25: BuildON should allow scalability of its data infrastructure.

7.1.2 Efficiency

The BuildON system must effectively handle, process, and store large volumes of data. This entails implementing optimal strategies for data storage allocation and minimising inefficient resource usage, including storage space, processing power, and memory. This is supported by the following NFRs from D2.1:

- NFR03: BuildON should be able to process, store and exploit a high amount of data.
- NFR17: BuildON should implement optimal data storage allocation.

7.1.3 Interoperability

The BuildON system needs to guarantee seamless communication among its core components and higher-layer services, aligning with the three interoperability layers outlined by the GridWise Architecture Council: technical (syntax), informational (semantics), and organisational (pragmatics). This entails ensuring compatibility in data formats and protocols, promoting a shared understanding of exchanged data via knowledge graphs, and facilitating effective coordination through well-defined governance frameworks and standards. This is supported by the following NFRs from D2.1:

- NFR01: BuildON should process data of high frequency in near real-time conditions.
- NFR04: BuildON should allow the interoperable use of data for [all services].

7.1.4 Security

The BuildON system needs to implement robust measures to uphold the confidentiality and integrity of data, safeguarding against unauthorised access or privacy breaches. This is supported by the following NFRs from D2.1:

- NFR02: BuildON should secure safe data transmission.
- NFR10: BuildON should be able to secure stored data safely.
- NFR11: BuildON should implement usage control policies for its services.

7.1.5 Reliability and availability

The BuildON system must prioritise the reliability of the systems and availability of data to minimise downtime and errors, ensuring consistent and uninterrupted service delivery. This is supported by the following NFRs from D2.1:

- NFR06: BuildON should provide commands from and to gateway devices.
- NFR09: BuildON should be able to provide remote control capabilities on demand.
- NFR16: BuildON should identify potential failures in its architecture.
- NFR19: BuildON tools and services should reduce downtime as much as possible.
- NFR20: BuildON should perform mitigation techniques for data retrieval.
- NFR22: BuildON should be able to isolate errors to maintain its functionalities.
- NFR23: BuildON should be able to handle multiple requests in bilateral signal communication.
- NFR24: BuildON should prevent its users from harmful equipment control actions.

7.2 Core components functional requirements

Based on the sequence diagrams (F1.1 to F2.2), the core components' functional requirements include the following:

7.2.1 Edge Domain Controller (DC)

Table 2 outlines the functional requirements pertinent to the DC. Most of these requirements are considered essential (must-haves), except R1.9 and R1.14, which depend upon hardware capabilities. Additionally, requirements R1.3, R1.8, and R1.12 are conditional on the presence of lightweight services at the edge to support applications, thereby necessitating more comprehensive data processing capabilities.

Table 2. Functional Requirements of the Edge Domain Controller

Identification	Name	Description	Related to	Derived from	Priority
<requirement ID>	<functional requirement name>	< description of functional requirement>	<component or actor name>	<core functionality>	<MoSCoW prioritisation>
Req-1.1	DC configuration	Configure field devices, including the population of internal data models and services	-	F1.2	must-have
Req-1.2	DC configuration	Configure internal scheduling and event admin	-	F1.6	must-have
Req-1.3	DC configuration	Configure internal alarming	-	F1.6	could-have
Req-1.4	DC configuration	Notify newly configured DC	DACM	F1.2	must-have
Req-1.5	DC configuration	Transform internal data model into configuration files (annotated point list) following a schema (e.g., based on Haystack or Web Of Things)	-	F1.2	must-have
Req-1.6	DC configuration	Send configuration files (annotated point list)	DACM	F1.2	must-have
Req-1.7	Data exchange	Pull/receive raw data	Field devices	F1.6	must-have
Req-1.8	Data process	Preprocess raw dynamic data from field device (e.g., unit conversion, conversion factors) when needed	-	F1.6	should-have
Req-1.9	Data storage	Short-term data storage	-	F1.6	could-have
Req-1.10	Data exchange	Send real-time data (it could be pre-processed or not, depending on Req-1.5)	DACM	F1.6	must-have
Req-1.11	Data exchange	Receive command data to write	DACM	F2.2	must-have
Req-1.12	Data process	Transform command data to be written as needed (e.g., HVAC mode)	-	F2.2	should-have
Req-1.13	Data exchange	Write command data	Field devices	F2.2	must-have
Req-1.14	Data exchange	Notify success command data writing	DACM	F2.2	could-have

7.2.2 Cloud Controller (CC)

Table 3 outlines the functional requirements pertinent to the CC. Most of these requirements are considered essential (must-haves), except for requirement Req-1.13, which is constrained to hardware capabilities and availability of API requests. In addition, requirements Req-1.3, Req-1.8, and Req-1.11 are conditional on lightweight services at the edge to support applications, necessitating more comprehensive data processing capabilities at the edge.

Table 3. Functional Requirements of the Cloud Controller

Identification	Name	Description	Related to	Derived from	Priority
<functional requirement ID>	<requirement name>	<description of functional requirement>	<component or actor name>	<core functionality>	<MoSCoW prioritisation>
Req-1.1	CC configuration	Configure web services, including population of internal data models and services	-	F1.3	must-have
Req-1.2	DC configuration	Configure internal scheduling and event admin	-	F1.6	must-have
Req-1.3	DC configuration	Configure internal alarming	-	F1.6	could-have
Req-1.4	CC configuration	Notify newly configured CC	DACM	F1.3	must-have
Req-1.5	CC configuration	Transform internal data model into configuration files (annotated point list) following a schema (e.g., based on Haystack or Web Of Things)	-	F1.3	must-have
Req-1.6	CC configuration	Send configuration files (annotated point list)	DACM	F1.3	must-have
Req-1.7	Data exchange	Receive raw data	Web services	F1.6	must-have
Req-1.8	Data process	Preprocess raw dynamic data from web services	-	F1.6	should-have
Req-1.9	Data exchange	Send real-time data (it could be pre-processed or not. Depending on Req-1.5)	DACM	F1.6	must-have
Req-1.10	Data exchange	Receive command data to write	DACM	F2.2	must-have
Req-1.11	Data process	Transform command data to be written as needed (e.g., HVAC mode)	-	F2.2	should-have
Req-1.12	Data exchange	Write command data	Web services	F2.2	must-have
Req-1.13	Data exchange	Notify success command data writing	DACM	F2.2	could-have

7.2.3 Data Access and Control Manager (DACM)

Table 4 outlines the functional requirements pertinent to the DACM. Most of these requirements are considered essential (must-haves), except for Req-1.3, Req-1.10, Req-1.15 and Req-1.16, which are non-essential functionalities such as notifications, alarms, and command writing assessment.

Table 4. Functional Requirements of the Data Access and Control Manager

Identification	Name	Description	Related to	Derived from	Priority
<functional requirement ID>	<requirement name>	< description of functional requirement>	<component or actor name>	<core functionality>	<MoSCoW prioritisation>
Req-1.1	DC/CC configuration	Request and receive configuration files (annotated point list based on Haystack or Web Of Things)	DC/CC	F1.2 -1.3	must-have
Req-1.2	DC/CC configuration	Validate configuration files (annotated point list based on Haystack or Web Of Things)	-	F1.2 -1.3	could-have
Req-1.3	DC/CC configuration	Send validation results and allow DC/CC configuration files (annotated point list based on Haystack or Web Of Things) to be inspected, corrected and approved	BuildON System integrators	F1.2 -1.3	should-have
Req-1.4	DC/CC configuration	Notify accepted configuration	DC/CC	F1.2 -1.3	must-have
Req-1.5	DC/CC configuration	Configure DC field devices and CC web services on DACM	-	F1.2 -1.3	must-have
Req-1.6	DC/CC configuration	Send DC/CC configuration files (e.g., based on Haystack or Web Of Things)	Repository	F1.4	must-have
Req-1.7	Data exchange	Receive real-time data (it could be pre-processed or not. Depending on Req-1.5 from DC/CC)	DC/CC	F1.6	must-have
Req-1.8	Data process	Process and perform quality checks on raw/pre-processed dynamic data from DC/CC	-	F1.6	must-have
Req-1.9	Data persistence and logging	Log data	-	F1.6	must-have
Req-1.10	Data persistence and logging	Activate alarms based on configuration data (e.g., thresholds, keep-alive parameters)	-	F1.6	could-have
Req-1.11	Data persistence and logging	Send fault notification	DC/CC	F1.6	should-have

Req-1.12	Data exchange	Send pre-processed real-time data	Repository	F1.6	must-have
Req-1.13	Data exchange	Send pre-processed real-time data	API	F2.1	must-have
Req-1.14	Data exchange	Receive requests to write command data	API	F2.2	must-have
Req-1.15	Write protection	Evaluate writing command	-	F2.2	should-have
Req-1.16	Write protection	Notify writing block	API	F2.2	should-have
Req-1.17	Data exchange	Write command	DC/CC	F2.2	must-have
Req-1.18	Data exchange	Receive notification on successful command writing	DC/CC	F2.2	must-have
Req-1.19	Write protection	Update registration to protect the signal	-	F2.2	must-have
Req-1.20	Data exchange	Notify success command writing	API	F2.2	must-have

7.2.4 BuildON Data Repository

Table 5 outlines the functional requirements pertinent to the Repository. Most of these requirements are considered essential (must-haves), except for requirement Req-1.4, which depends on the format and condition of the static data input.

Table 5. Functional Requirements of the BuildON Data Repository

Identification	Name	Description	Related to	Derived from	Priority
<functional requirement ID>	<requirement name>	<description of functional requirement>	<component or actor name>	<core functionality>	<MoSCoW prioritisation>
Req-1.1	Data ingestion	Receive static files	Data integrators	F1.4	must-have
Req-1.2	Data ingestion	Receive configuration files (annotated point list)	DACM	F1.4	must-have
Req-1.3	Data storage	Store static data files and configuration files (annotated point list)	-	F1.4	must-have
Req-1.4	Data process	Check static data	-	F1.4	should-have
Req-1.5	Data process	Annotate static data	-	F1.4	must-have
Req-1.6	Data integration	Generate partial knowledge graphs	-	F1.4	must-have
Req-1.7	Data validation	Validate partial knowledge graphs	-	F1.4	must-have
Req-1.8	Data integration	Create merger knowledge graphs	-	F1.4	must-have
Req-1.9	Data management	Receive services' metadata requirements	API	F1.5	must-have
Req-1.10	Data management	Translate metadata requirements into SHACL shapes	-	F1.5	must-have

Req-1.11	Data validation	Validate knowledge graphs against service requirements	-	F1.5	must-have
Req-1.12	Data management	Send SHACL validation results	API	F1.5	must-have
Req-1.13	Data management	Receive requests for metadata per endpoint	API	F1.5	must-have
Req-1.14	Data management	Instantiate ETL tools	-	F1.5	must-have
Req-1.15	Data management	Send metadata per endpoint	API	F1.5	must-have
Req-1.16	Data management	Receive preprocessed dynamic data	DACM	F1.6	must-have
Req-1.17	Data storage	Store preprocessed dynamic data	-	F1.6	must-have
Req-1.18	Data exchange	Send preprocessed historical data	API	F2.1	must-have

7.2.5 Universal API

Table 6 outlines the functional requirements pertinent to the Universal API. Most of these requirements are considered essential (must-haves), except for requirement Req-1.9, Req-1.10, Req-1.15 and Req-1.26, representing non-essential functions such as notifications, service selection, service list creation, and service prioritisation.

Table 6. Functional requirements of the Universal API

Identification	Name	Description	Related to	Derived from	Priority
<functional requirement ID>	<requirement name>	<description of functional requirement>	<component or actor name>	<core functionality>	<MoSCoW prioritisation>
Req-1.1	Actors/users registration	Allow actors and users to be registered and assign roles to actors	BuildON admin	F1.1	must-have
Req-1.2	Roles assignment	Allow to assign user roles	BuildON admin, BuildON system integrators, DC providers, CC providers, Service providers	F1.1	must-have
Req-1.3	Roles assignment	Apply for user roles and permissions	-	F1.1	must-have
Req-1.4	Roles assignment	Create components, roles and credentials	-	F1.1	must-have
Req-1.5	Services configuration	Allow to register service requirements	Service providers	F1.5	must-have
Req-1.6	Services configuration	Compile metadata requirements per services	-	F1.5	must-have

Req-1.7	Services configuration	Send services metadata requirements	Repository	F1.5	must-have
Req-1.8	Services configuration	Receive service validation results	Repository	F1.5	must-have
Req-1.9	Services configuration	Notify services validation results	BuildON System integrators	F1.5	should-have
Req-1.10	Services configuration	Allow to select services	BuildON System integrators	F1.5	should-have
Req-1.11	Services configuration	Configure API (e.g., create endpoints per service)	-	F1.5	must-have
Req-1.12	Services configuration	Request metadata per endpoint	Repository	F1.5	must-have
Req-1.13	Services configuration	Activate endpoints per service	-	F1.5	must-have
Req-1.14	Services configuration	Notify about service activation, including endpoint info	Service providers	F1.5	must-have
Req-1.15	Services configuration	Create a list of services available and prioritisation logic to orchestrate them.	-	F1.5	should-have
Req-1.16	Services authentication	Receive service authentication request	Services	F2.1-2.2	must-have
Req-1.17	Services authentication	Check services roles	-	F2.1-2.2	must-have
Req-1.18	Data exchange	Receive data read/write requests from services	Services	F2.1-2.2	must-have
Req-1.19	Data exchange	Request pre-processed historical data	Repository	F2.1	must-have
Req-1.20	Data exchange	Receive pre-processed historical data	Repository	F2.1	must-have
Req-1.21	Data exchange	Request pre-processed real-time data	DACM	F2.1	must-have
Req-1.22	Data exchange	Receive pre-processed real-time data	DACM	F2.1	must-have
Req-1.23	Data exchange	Send pre-processed real-time and historical data	Services	F2.1	must-have
Req-1.24	Data exchange	Receive requests to write command data	Services	F2.2	must-have
Req-1.25	Data exchange	Request to write command data	DACM	F2.2	must-have
Req-1.26	Data exchange	Evaluate prioritisation logic	-	F2.2	should-have
Req-1.27	Data exchange	Receive notification on the writing command block	DACM	F2.2	must-have
Req-1.28	Data exchange	Notify writing command block	Services	F2.2	must-have
Req-1.29	Data exchange	Receive a notification on a successful write command	DACM	F2.2	must-have
Req-1.30	Data exchange	Notify success command writing	Services	F2.2	must-have

8 Software Architecture

Once the core functionalities and fundamental interactions have been identified and established, in this section, we proceed to outline the BuildON detailed software architecture and components specifications. This involves aligning the described conceptual architecture with the required functionalities while addressing functional and non-functional requirements for each component, as identified in the previous sections.

Figure 13 provides an intricate view of the proposed software architecture, outlining its core components and their corresponding sub-components, which will be elaborated on in subsequent sections. Further explanation of the development plan for these components, clarifying their proposed architectural patterns, is presented in Section 9.1.

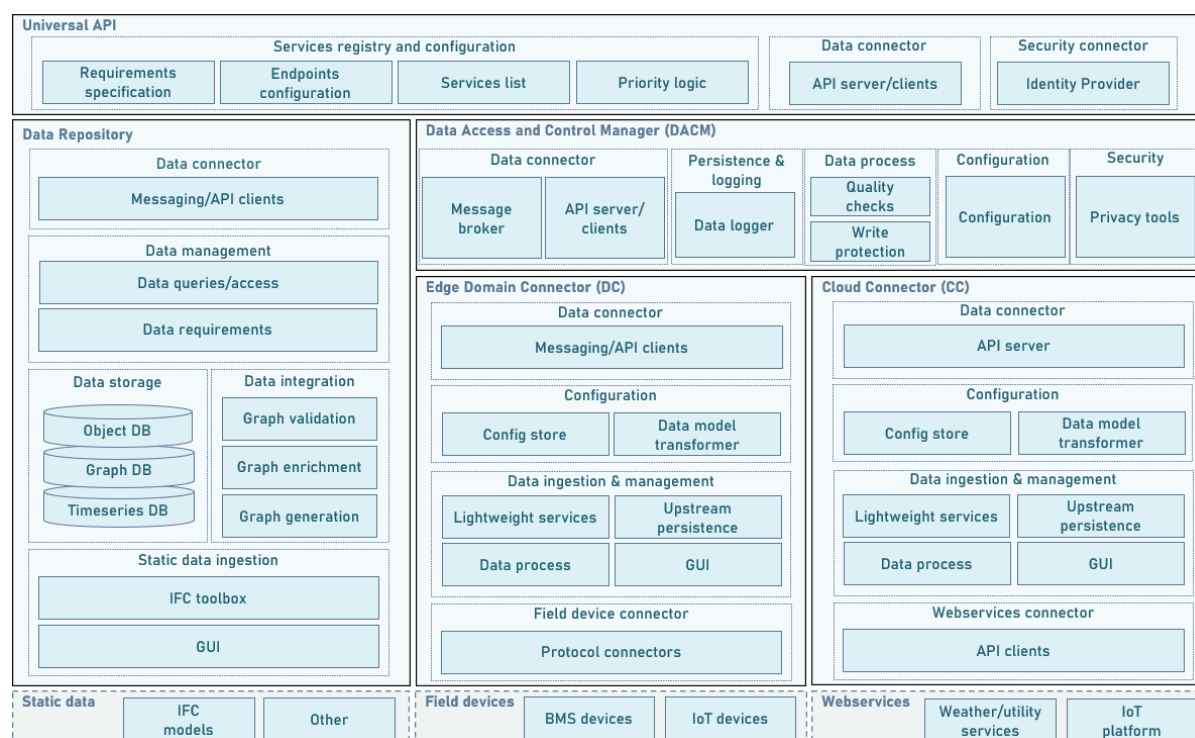


Figure 13 – BuildON software architecture

8.1 Components

8.1.1 Edge Domain Controller (DC)

Following the specification of the DC functionalities and requirements, two DC implementations are provided using existing gateways:

- The Holistic gateway is mostly used for the BuildON residential pilots to monitor and control different types of smart home systems. This gateway consists of a platform to connect various controller and sensor types on a closed local wireless network and serves as a data acquisition and ingestion element to offer data to the DACM and Smart Home solutions owned by Holistic. The gateway also provides advanced features, such as energy management and AI-based IoT control, that supports other requirements, including lightweight services on the edge.

- The Yodiwo gateway is used for the BuildON tertiary pilots to access different energy assets through a set of available building automation drivers (BACnet, Modbus, KNX or LoRA¹⁶). The gateway uses these drivers to capture and process, in real or post-time, any signal coming from sensors and metering devices installed on the building premises. This gateway also offers advanced features to support other requirements, including implementing lightweight services at the edge.

While these are specific gateways, both need to be updated to support BuildON's common architecture and interfaces. This exercise is meant to showcase that various gateways can be easily integrated within the BuildON framework (for both domestic and non-domestic use cases), assuming alignment with the reference architecture. This should highlight the potential for a highly extensible and replicable solution that includes third-party providers that can implement the specific interfaces and benefit from the higher-layer components.

8.1.1.1 Description and Component Diagram of Holistic DC

The Holistic gateway operates on a Raspberry Pi and will act as a BuildON DC consisting of several sub-components, which include:

- The **Protocol Connectors** subcomponent facilitates communication with IoT devices based on Message Queue Telemetry Transport (MQTT)¹⁷ and HTTPS protocols. MQTT will be primarily used for real-time communication due to its lightweight and efficient publish-subscribe model, suitable for resource-constrained devices. Meanwhile, HTTPS will be used when MQTT is unavailable or to transfer historical data or logs.
- The **Data Process** subcomponent preprocesses incoming data from IoT devices before storing it in short-term storage and committing it to the DACM. This subcomponent also handles command processing before executing actions in DC.
- The **Upstream Persistence** subcomponent supports short-term storage, specifically with SQLite, to store data in case of internet failures or when real-time updates are not necessary. This ensures data resilience and availability even in less-than-optimal network conditions.
- The **Lightweight Services** subcomponent incorporates a subset of control logic essential for executing processes triggered by the internal scheduler. This design choice ensures efficiency and simplicity in managing control logic. The internal scheduler will be a built-in Linux scheduler (Cronjob) to trigger processes based on preset schedules and intervals.
- The **GUI** subcomponent, accessible via Wi-Fi, allows users to easily set up the IoT devices, enhancing overall configurability and user experience.
- The **Configuration** subcomponent helps to compose and store configuration files (as annotated point lists) with SQLite for each IoT device. These configuration files will be shared with the DACM for registry and validation and later with the Data Repository for semantically generating knowledge graphs.
- The **Data Connectors** subcomponent uses the MQTT protocol for real-time data transmission to the DACM, ensuring timely updates and responsiveness. Also, it employs HTTPS to send data, providing a secure and reliable communication channel.

¹⁶ <https://lora-alliance.org/>

¹⁷ <https://mqtt.org/>

Figure 14 illustrates the diagram of the subcomponents of the Holistic DC.

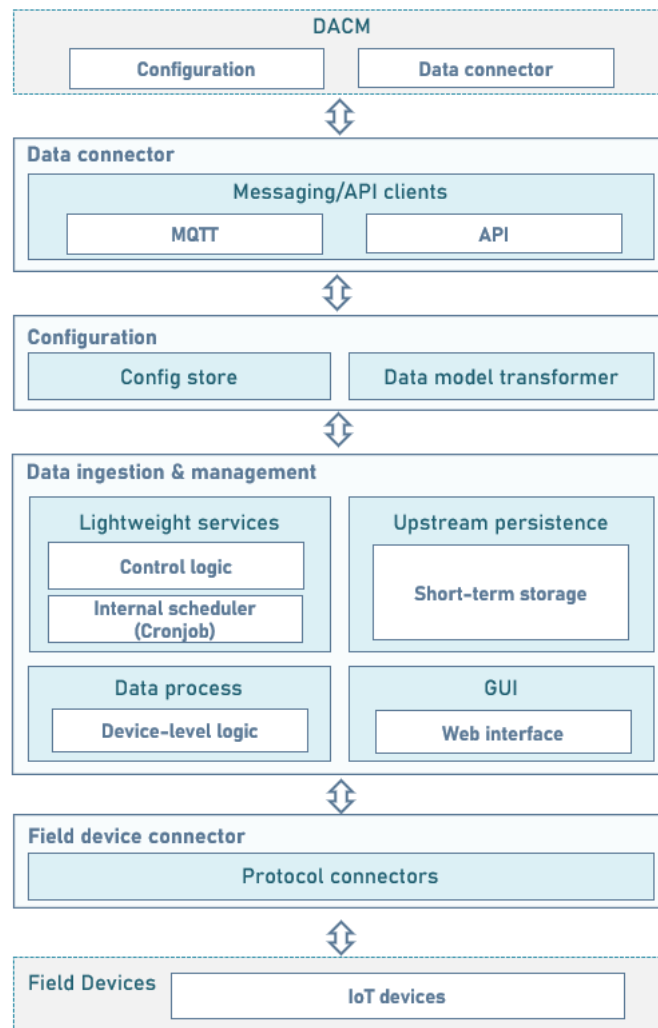


Figure 14 – Edge Domain Controller (Holistic DC) subcomponents diagram

8.1.1.2 Description and Component Diagram of Yodiwo DC

The Yodiwo gateway integrates software agents and scripts installed on Industrial Linux mini-PCs or off-the-shelf appliances based on OpenWRT¹⁸, sourced from Yodiwo's hardware partners. As such, it will act as a BuildON DC consisting of several key subcomponents, which include:

- The **Protocol Connectors** subcomponent facilitates device connectivity via various industrial protocols such as BACnet, Modbus, KNX, and more. It harnesses hardware with at least two Ethernet ports, custom General Purpose Input/Output (GPIOs, and one or more industrial serial ports (RS232/RS422/RS485). Additionally, it provides alternative methods for device connectivity by enabling communication over Wi-Fi and mobile networks, supporting wireless networking (Wi-Fi) and mobile networks (4G/5G), respectively.
- The **Data Process** subcomponent is responsible for preprocessing incoming data from field devices before using it in (optional) lightweight services applied at the edge, storing it in short-term storage and transmitting it to the DACM. In addition, it manages command processing to ensure proper

¹⁸ <https://openwrt.org/>

execution of actions in the DC.

- The **Upstream Persistence** subcomponent is responsible for preventing data loss in cases where DC devices lose internet connectivity and access to the DACM services for some time. The agent within the gateway is able to store data locally if it fails to send and attempt to resend it when the connection to the node service is restored. For this purpose, we use LiteDBv5 or simple text files: when data fails to be forwarded to the DACM, the agent writes these messages to a separate storage partition and periodically uses the cron Linux subsystem to schedule retransmission tries. This follows the Unix/Linux approach of compartmentalisation and simpler systems, each designed to carry out specific tasks as reliably and efficiently as possible.
- The **Lightweight Services** subcomponent operates via the core agent, functioning as a system service that automatically start upon device power-up. Its primary tasks include collecting and transmitting data to the cloud and executing commands received from the cloud to modify settings. Supporting services and scripts for the node agent will be developed in Python 3 and Bash, ensuring compatibility with most Linux distributions. Moreover, this subcomponent will be equipped with specialised modules such as Real-time Clocks, enhancing data logging capabilities to withstand power cycles.
- The **Web Interface** subcomponent supports user interactions for the initial configuration of the devices, leveraging services such as sshd, rsyslog, logrotate, and additional software (firewall, OpenVPN). Such services are configured to onboard the device and help achieve the desired level of protection and reliability for the device. The Yodiwo gateway uses bash scripts, Ansible¹⁹ roles and playbooks to assist with specific actions and processes, such as adding the gateway to the FEM platform (provisioning) or the DACM, adding or customising gateway security settings (security hardening) and changes to system settings (e.g., network configuration).
- The **Configuration** subcomponent transforms and stores configuration files, represented as annotated point lists, for each field device. These configurations will be shared with the DACM for registry and validation and later with the Data Repository for generating knowledge graphs.

The Data Connectors subcomponent provides WAN upstream connections with the DACM, which will be established using MQTT or the SignalR protocol. The communication subcomponent will use Yodiwo's gateway's secure broker for the MQTT to ensure data integrity and privacy. For SignalR, an evolution of WebSockets, the communication will operate over the HTTPS 443 port. Both connection types will be secured using TLS v1.2 and VPN. The VPN will be employed using OpenVPN, which will be automatically set up with per-DC-device certificates upon provisioning.

¹⁹ Ansible is an open-source automation tool, or platform, used for IT tasks such as configuration management, application deployment, intra-service orchestration, and provisioning. It aims to provide a simple, yet powerful, automation engine that can automate complex multi-tier IT application environments.

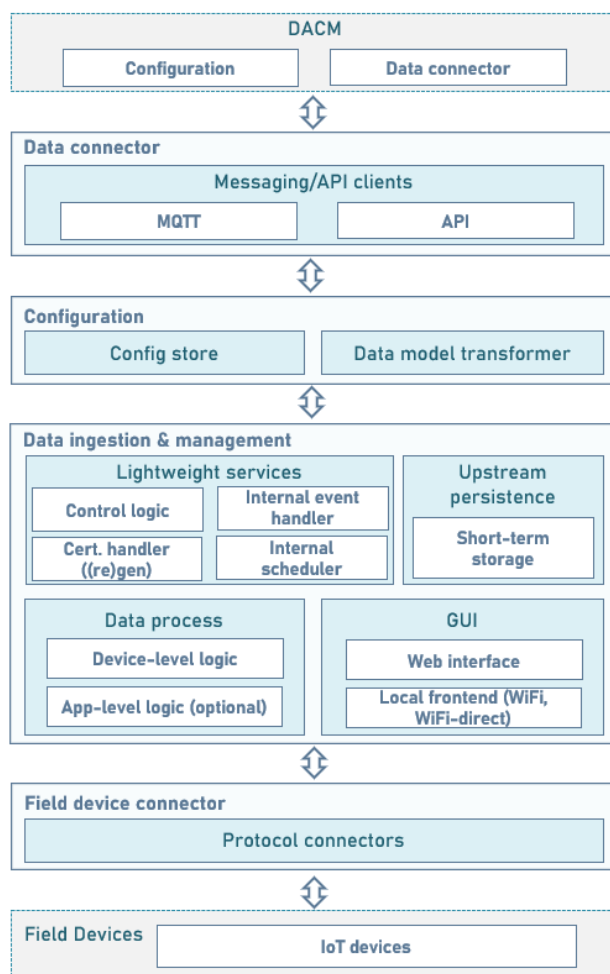


Figure 15 – Edge Domain Controller (Yodiwo DC) subcomponents diagram

To further understand the Yodiwo gateway, Figure 16 illustrates an internal procedure, demonstrating an interaction between an agent, a Modbus-based device, and an MQTT broker facilitated through an internal broker.

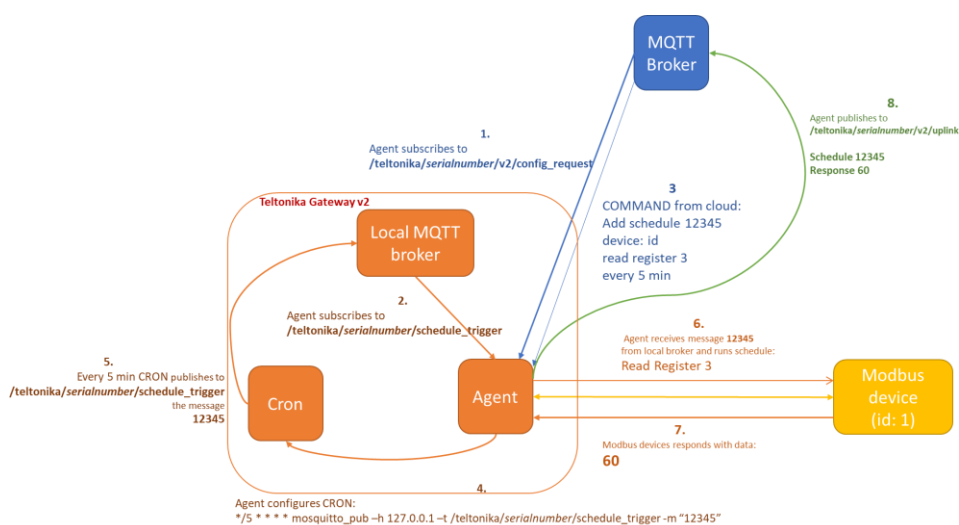


Figure 16 – Edge Domain Controller (Yodiwo DC) internal procedures

8.1.1.3 Implementation libraries, frameworks and tools

Table 7 and Table 8 list the libraries, frameworks and tools to support the development of Holistic and Yodiwo's DCs, respectively.

Table 7. Libraries, frameworks and tools to support the Edge Domain Controller for Holistic DC

Subcomponent	Libraries, frameworks and tools
Protocol Connectors	Python, Paho MQTT Library, MQTT Mosquitto Broker
Command Data Process	Python
Upstream persistence	SQLite
Lightweight control logic interface	Python
Internal Scheduler	Built-in Linux internal scheduler, Cronjob
Config Store	SQLite
Web Interface	Flask
Data connectors	Python, Paho MQTT Library

Table 8. Libraries, frameworks and tools to support the Edge Domain Controller for Yodiwo DC

Subcomponent	Libraries, frameworks and tools
Protocol Connectors	Modbus, BACnet, KNX, LoRa
Command Data Process	.Net
Upstream persistence	text files, LiteDB
Lightweight control logic interface	Control over Data channel, onboarding/provisioning over HTTPS
Internal Scheduler	Cron
Config Store	text files in /etc
Data connectors	MQTT, SignalR, HTTPS

8.1.2 Cloud Controller (CC)

Following the specification of the CC functionalities and requirements, the proposed BuildON CC will consist of several key subcomponents, including:

- The **Web services Connector** relies on REST APIs to connect with diverse web services (e.g., IoT platforms, utility and weather providers).
- The **Data Process** subcomponent preprocesses incoming data before forwarding it to DACM and handles the incoming commands from the DACM before initiating the appropriate actions. This ensures a streamlined flow of information.
- The **Lightweight Services** subcomponent relies on an internal scheduler to schedule specific actions (control logic) optimally. This ensures seamless coordination of tasks and actions within the cloud infrastructure.
- The **GUI** component, accessible via the internet, allows users to quickly set up the devices via IoT platforms and other external web services, enhancing overall reconfigurability and user experience.
- The **Configuration** subcomponent supports creating configuration files (as annotated point lists) and uses PostgreSQL to store them securely. These configuration files are shared with the DACM for registry and validation and later with the Data Repository for semantically generating knowledge graphs.
- The **Data Connectors** subcomponent uses HTTPS to share data with the DACM and REST API for incoming commands and data from DACM. This will ensure smooth data exchange and integration with the DACM within the cloud environment.

8.1.2.1 Component diagram

Figure 17 illustrates the CC's subcomponent diagram.

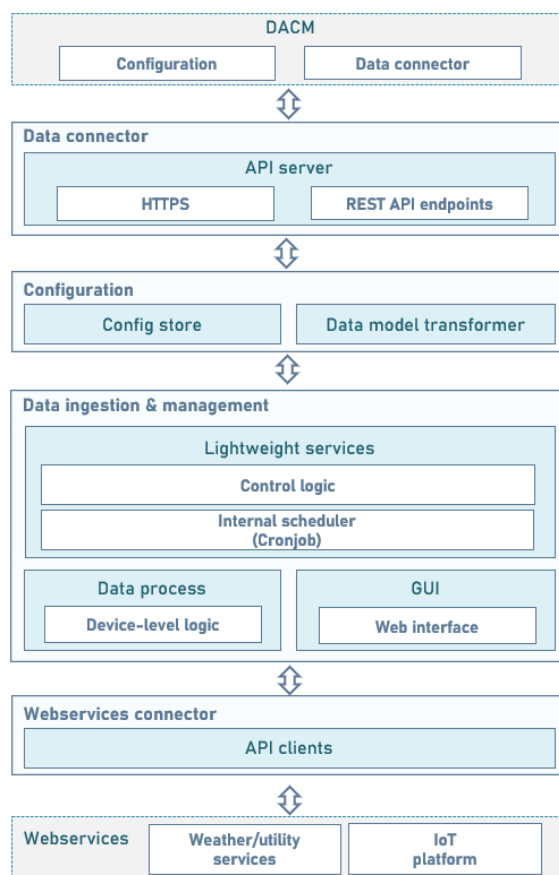


Figure 17 – Cloud Controller (Holistic DC) subcomponents diagram

8.1.2.2 Implementation libraries, frameworks and tools

Table 9 lists the libraries, frameworks and tools to support the development of the CC.

Table 9. Libraries, frameworks and tools to support the Cloud Controller

Subcomponent	Libraries, frameworks and tools
Web services connector	Python, Library Requests
Command data process	Python
Lightweight control logic interface	Python
Internal scheduler	Python, Library schedule
Config Store	PostgreSQL
Web Interface	Flask
Data Connectors	Python, Fast API Library Requests

8.1.3 Data Access and Control Manager (DACM)

Following the specification of the DACM functionalities and requirements, the proposed BuildON DACM will consist of several key subcomponents, including:

- The **Data Connector** subcomponent handles the connection between the DC, CC, Data Repository, and API. To connect to the DC/CC, the DACM uses a Message Broker for real-time data, command, and signal exchange. This will involve implementing protocols such as MQTT & CoAP tailored to the project's requirements. Additionally, an event streaming platform like Kafka²⁰ facilitates seamless communication between these layers. For the DACM, data such as device status updates and smart meter measurements are uploaded to the repository through REST API calls. These APIs will ensure data integrity and synchronisation across the architecture.
- The **Configuration** subcomponent allows setting up field devices and web services based on the configuration files received from the DC and CC, which support proper metadata annotations per data point.
- The **Data Processing** sub-component performs data preprocessing and quality checks before storing and sharing data with upper-layer components and services. It also enforces write protection to prevent issues, such as values exceeding acceptable ranges.
- The **Security Connector** subcomponent implements security policies based on service specifications and on-site DC deployments. It enforces necessary security restrictions by communicating with a Keycloak-based security framework via REST API. It is a back-end application within the DACM, which facilitates secure interconnection and interaction.
- The **Persistence & Logging** subcomponent implements the task of logging data related to the mechanisms requiring continuous monitoring. This will be achieved by integrating a new platform with the relevant sub-components or adapting existing software such as Netdata²¹ or Grafana²² to meet the project's specific requirements.

8.1.3.1 Component diagram

Figure 18 illustrates the diagram of the DACM's subcomponents.

²⁰ <https://kafka.apache.org/>

²¹ <https://www.netdata.cloud/>

²² <https://grafana.com/products/cloud/logs/>

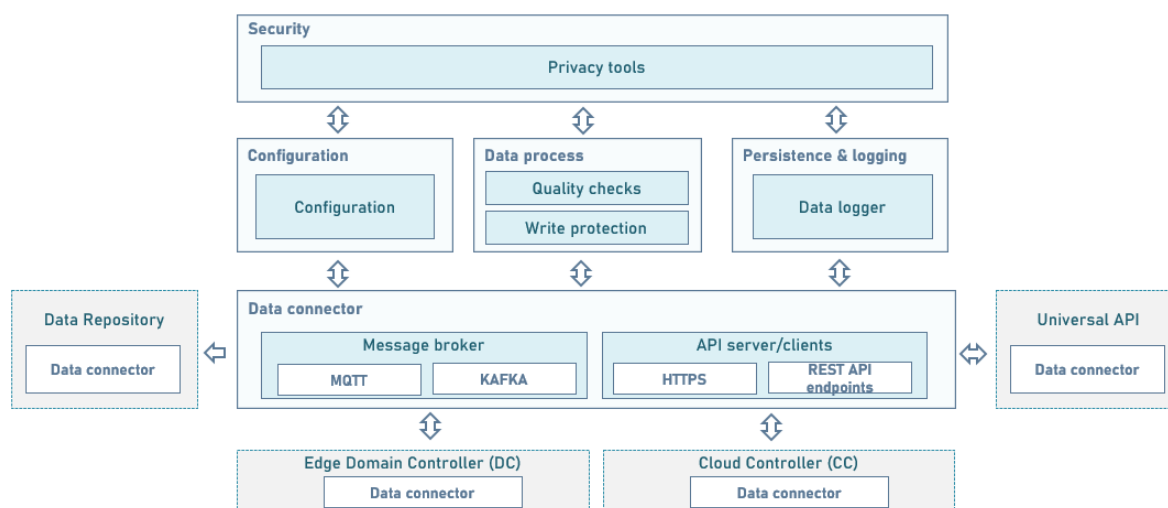


Figure 18 – Data Access and Control Manager (DACM) subcomponents diagram

8.1.3.2 Implementation libraries, frameworks and tools

Table 10 lists the libraries, frameworks and tools to support the development of the DACM.

Table 10. Libraries, frameworks and tools to support the Data Access and Control Manager

Subcomponent	Libraries, frameworks and tools
DC/CC Connector	Python, FastAPI, Git, Swagger, MQTT
Data Repository connector	Python, FastAPI, Git, Swagger, MQTT
Security connector	Python, Git
Persistence and Logging	Python, JavaScript, Git, open-source tools (e.g., Grafana, Netdata)

8.1.4 BuildON Data Repository

Following the specification of the Data Repository functionalities and requirements, the proposed BuildON Data Repository comprises several key subcomponents, including:

- The **Static Data Ingestion** subcomponent allows users with appropriate permissions to upload data files through a user-friendly web-based Graphical User Interface (GUI). Supported data formats include CSV, JSON, XML data files, and BIM data models adhering to open standards like the Industry Foundation Classes (IFC). This subcomponent also hosts an IFC toolbox that checks the data completeness of the IFC using the Model View Definition (MVD) standard and enriches the IFC using geometric enrichment algorithms. These functionalities are required to extract topological information (used in the knowledge graphs) from Building Information Models or other files that support geometric primitives.
- The **Data Storage** subcomponent is a shared storage repository supporting the knowledge graph, time series, and object databases deployed on a cloud infrastructure. The time series database will capture dynamic data from field devices and web services loaded through the

Data Management subcomponent, while the object database accommodates various data types, both structured and unstructured, uploaded by the Data Ingestion subcomponent. The graph database will also store structured RDF data generated by the knowledge graph generation tool in the Data Integration subcomponent. This RDF data, stored in TTL format, will support read and write operations through a SPARQL endpoint used by the Data Management subcomponent.

- The **Data Integration** subcomponent is responsible for generating, enriching, and validating knowledge graphs for each pilot, which are crucial in enabling the integration among the pilots' heterogeneous data sources. The generation process involves a series of Extract, Transform, and Load (ETL) tools designed to transform diverse static and configuration files into RDF data following selected ontologies. Each ETL process will be dedicated to transforming specific input data types into RDF and creating partial graphs that are stored in the graph database. These partial graphs will then be merged into a unified graph for each pilot, which will be subsequently enriched with additional RDF instances based on predefined inference rules from the chosen ontologies. A validation tool employing SHACL shapes will also be hosted in this subcomponent to ensure that the resulting enriched knowledge graphs support predefined data requirements. The outcomes of this validation process will be reported, providing users with actionable insights to address any necessary corrective actions.
- The **Data Connector and Management** subcomponent loads the dynamic data from the DACM to the Data Storage subcomponent and exposes the stored time series data and static information (metadata) to BuildON Services through the API. Based on the BuildON Services data and metadata requirements, this subcomponent coordinates data transformation operations and responses to data and metadata queries, using messaging protocols such as MQTT and leveraging the Kafka distributed streaming platform. Additionally, this component handles the loading of configuration files from the DACM, which is essential for populating knowledge graphs with descriptions of field devices (e.g., their data points, topology, and relationships). It provides a comprehensive REST API to interact with the Data Storage subcomponent, where all files are stored.

8.1.4.1 Component diagram

Figure 19 illustrates the diagram of the subcomponents of the Data Repository.

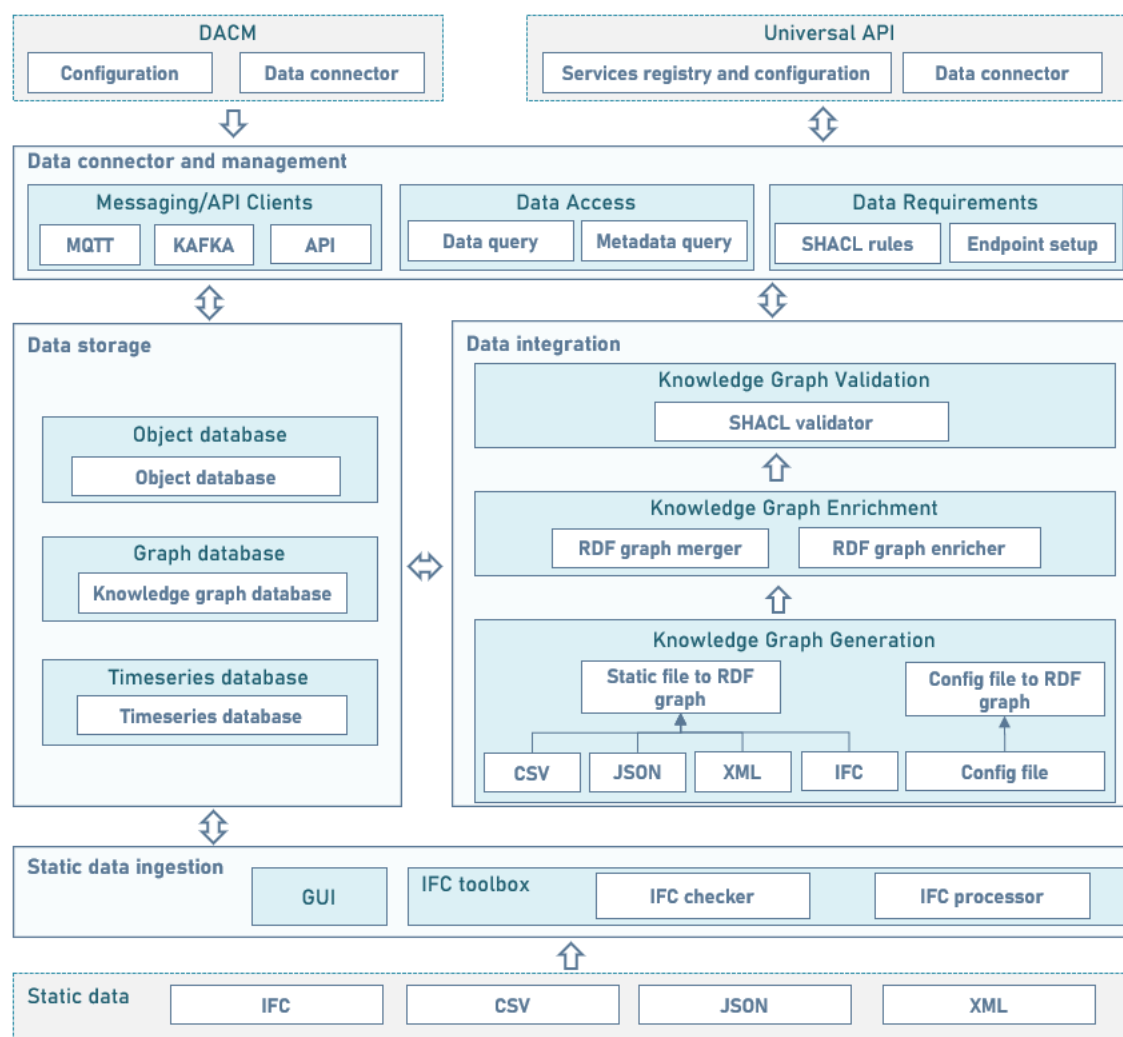


Figure 19 – Data Repository subcomponents diagram

8.1.4.2 Implementation libraries, frameworks and tools

Table 11 lists the libraries, frameworks and tools to support the development of the Data Repository.

Table 11. Libraries, frameworks and tools to support the BuildON Data Repository

Subcomponent	Libraries, frameworks and tools
Static data ingestion	Spring Framework Ecosystem (web-based applications)
Data storage	InfluxDB, PostgreSQL (time series database); GraphDB, Apache Jena and RDFLib (graph database; MongoDB (object database)
Data Integration	Spring Framework Ecosystem (web-based applications); Apache Jena or RDFLib/pySHACL (RDF generation, enrichment and validation);
Data connection and management	Kafka streaming platform, FastAPI (REST API)

8.1.5 Universal Building API

Following the specification of the Universal API functionalities and requirements, the proposed BuildON Universal API will consist of several subcomponents, including:

- The **Data Connector** subcomponent is designed based on API endpoints, serving two purposes: 1) to exchange data between the services and the DACM and 2) to facilitate data exchange between the services and the Data Repository. The endpoints for services and DACM will support requests to read real-time data from the DACM and to write outputs (commands, recommendations, etc.) to the DACM. The services and Data Repository endpoint will support requests to retrieve historical data (for analysis, model training, etc.) from the Repository and store data produced by the services to the Repository. Separate endpoints will be generated for each of the services developed within the framework of WP4 & WP5 in strict accordance with the specifications provided by the developers of each service.
- The **Context Discovery** sub-component performs two essential tasks: 1) the services' requirements specification and 2) the endpoint configuration. The output of these sub-tasks will be a detailed mapping of the specifications in terms of data, commands, signals and needed components per service, leading to detailed documentation about them to support their onboarding in different buildings.
- The **Services Registry** sub-component includes the overall services list and priority logic module. These will serve as an intermediate orchestrator between the various outcomes of separate services that can produce commands or suggestions for the same building on the field.
- Keycloak is part of the **Security Connector** subcomponent as an Identity Provider and ensures a robust identity and access management solution using role-based access control.

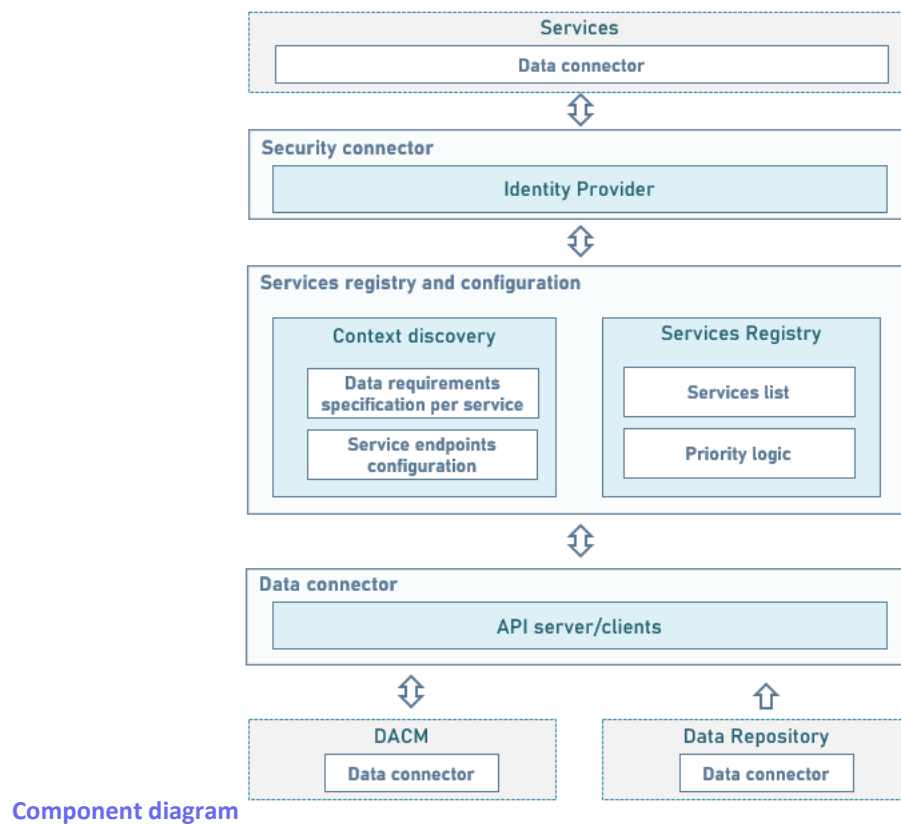


Figure 20 illustrates the diagram of the subcomponents of the Universal API.

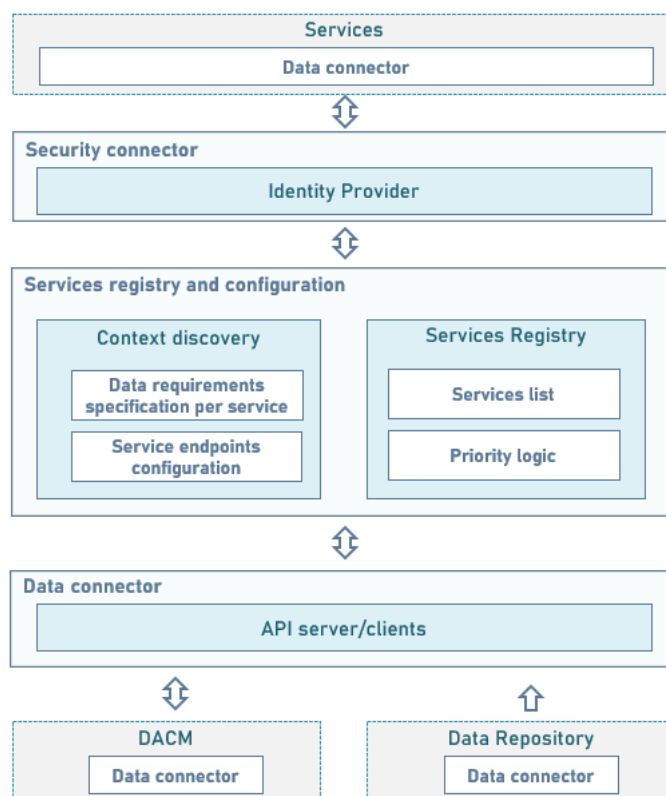


Figure 20 – Universal API subcomponents diagram

8.1.5.1 Implementation libraries, frameworks and tools

Table 12 lists the libraries, frameworks and tools to support the development of the Universal API.

Table 12. Libraries, frameworks and tools to support the Universal API

Subcomponent	Libraries, frameworks and tools
Context Discovery	Git, Swagger, Postman
Security Connector	Python, Git, FastAPI, Keycloak
Services Registry	Docker, Python, FastAPI
Data connector (REST API)	Python, Docker, FastAPI, Swagger

8.2 Interfaces specification

The upcoming subsections outline the interfaces connecting the core components, field devices, web services, higher-layer services, and users, as outlined in the sequence diagrams from Section 6. These subsections elaborate on their active and passive roles, the processes they're involved in, the types and formats of data exchanged, and the methods used (e.g., protocol, broker, GUI, API type).

8.2.1 DC and Field devices

Table 13. Interfaces between DC and Field devices

Interfaces	Active component/actor	Passive component/actor	Process	Data	Format	Method used
< interface's identifier>	<related active component name or actor>	<related passive component name or actor>	<process description>	< data exchanged>	<data format>	<protocol, broker, GUI, API type>
F1.6 - I1.1	DC	Field devices	Pull raw data	Signals, Measurements, data request	JSON, Binary	REST API, Modbus, BACnet, KNX, Dali
F1.6 - I1.2	Field devices	DC	Send raw data	Signals, Measurements, Real-time data	JSON, Binary	LoRa, MQTT, REST API
F1.6 - I2	Field devices	DC	Push raw data	Signals, Measurements, Real-time data	JSON, Binary	BACnet, OPC
F2.2 - I4.1	DC	Field devices	Write command	Commands, Signals	JSON, Binary	Modbus, BACnet, KNX, Dali, LoRa, OPC, REST API, MQTT
F2.2 - I4.2	Field devices	DC	Notify success command writing	Status updates	JSON, Binary	Modbus, BACnet, KNX, Dali, LoRa, OPC, REST API, MQTT

8.2.2 DC and DACM

Table 14. Interfaces between DC and DACM

Interfaces	Active component/actor	Passive component/actor	Process	Data	Format	Method used
< interface's identifier>	<related active component name or actor>	<related passive component name or actor>	<process description>	< data exchanged>	<data format>	<protocol, broker, GUI, API type>
F1.1 - I11	DC	DACM	DC/DACM connection	Authentication info (access token)	JSON	HTTPS
F1.2 - I3.1	DC	DACM	Notify new configuration	Status updates	JSON	HTTPS
F1.2 - I3.2	DACM	DC	Request configuration files (annotated point list)	Configuration data request	JSON	HTTPS

F1.2 - I3.3	DC	DACM	Send configuration files (annotated point list)	Configuration data	JSON	HTTPS
F1.2 - I3.4	DACM	DC	Notify accepted configuration	Status updates	JSON	HTTPS
F1.6 - I3.1	DC	DACM	Send pre-processed real-time data	Real-time data	JSON	HTTPS, MQTT, SignalR
F1.6 - I3.2	DACM	DACM	Send fault notification	Fault notification	JSON	HTTPS
F2.2 - I3.1	DACM	DACM	Write command	Commands, Signals	JSON	HTTP, MQTT
F2.2 - I3.2	DC	DACM	Notify success command writing	Status updates	JSON	HTTPS

8.2.3 CC and Web Services

Table 15. Interfaces between CC and Web Services

Interfaces	Active component /actor	Passive component /actor	Process	Data	Format	Method used
< interface's identifier>	<related active component name or actor>	<related passive component name or actor>	<process description>	< data exchanged>	<data format>	<protocol, broker, GUI, API type>
F1.6 - I4	Web Services	CC	Push raw data	Signals, Measurements, Real-time data	JSON	HTTPS
F2.2 - I6.1	CC	Web Services	Write command	Commands, Signals	JSON	HTTPS
F2.2 - I6.2	Web Services	CC	Notify success command writing	Status updates	JSON	HTTPS

8.2.4 CC and DACM

Table 16. Interfaces between CC and DACM

Interfaces	Active component /actor	Passive component /actor	Process	Data	Format	Method used
< interface's identifier>	<related active component name or actor>	<related passive component name or actor>	<process description>	< data exchanged>	<data format>	<protocol, broker, GUI, API type>

F1.1 - I15	CC	DACM	CC/DACM connection	Authentication info (access token)	JSON	HTTPS
F1.3 - I3.1	CC	DACM	Notify newly configured CC	Status updates	JSON	HTTPS
F1.3 - I3.2	DACM	DC	Request configuration files (annotated point list)	Configuration data	JSON	HTTPS
F1.3 - I3.3	CC	DACM	Send configuration files (annotated point list)	Configuration data	JSON	HTTPS
F1.3 - I3.4	DACM	DC	Notify accepted configuration	Status updates	JSON	HTTPS
F1.6 - I5.1	CC	DACM	Send preprocessed real-time data	Real-time data	JSON	HTTPS
F1.6 - I5.2	DACM	DACM	Send fault notification	Fault notification	JSON	HTTPS
F2.2 - I5.1	DACM	DACM	Write command	Signals and commands	JSON	HTTPS
F2.2 - I5.2	CC	DACM	Notify success command writing	Status updates	JSON	HTTPS

8.2.5 DACM and Repository

Table 17. Interfaces between DACM and Repository

Interfaces	Active component/actor	Passive component/actor	Process	Data	Format	Method used
< interface's identifier>	<related active component name or actor>	<related passive component name or actor>	<process description>	< data exchanged>	<data format>	<protocol, broker, GUI, API type>
F1.1 - I7	DACM	Repository	DACM/Repository connection	Authentication info (access token)	JSON	HTTPS
F1.4 - I2.1	DACM	Repository	Send configuration files (annotated point list)	Configuration data	JSON	HTTPS

F1.4 - I2.2	Repository	DACM	Send validation results	Validation results	JSON	HTTPS
F1.6 - I6	DACM	Repository	Send pre-processed real-time data	Signals, Measurements	JSON	HTTPS, MQTT

8.2.6 DACM and Universal API

Table 18. Interfaces between DACM and Universal API

Interfaces	Active component /actor	Passive component /actor	Process	Data	Format	Method used
< interface's identifier>	<related active component name or actor>	<related passive component name or actor>	<process description>	< data exchanged>	<data format>	<protocol, broker, GUI, API type>
F1.1 - I6	DACM	API	DACM/API connection	Authentication info (access token)	JSON	HTTPS
F2.1 - I2.1	API	DACM	Request pre-processed real-time data	Signals, measurements request	JSON	HTTPS
F2.1 - I2.2	DACM	API	Send pre-processed real-time data	Signals, measurements	JSON	HTTPS
F2.2 - I2.1	API	DACM	Request to write command	Signals and commands request	JSON	HTTPS
F2.2 - I2.2	DACM	API	Notify success command writing	Status updates	JSON	HTTPS
F2.2 - I2.3	DACM	API	Notify writing block	Error messages	JSON	HTTPS

8.2.7 Repository and Universal API

Table 19. Interfaces between Repository and Universal API

Interfaces	Active component /actor	Passive component /actor	Process	Data	Format	Method used
< interface's identifier>	<related active component name or actor>	<related passive component name or actor>	<process description>	< data exchanged>	<data format>	<protocol, broker, GUI, API type>

F1.1 - I3	Repository	API	Repository/ API connection	Authenticati on info (access token)	JSON	HTTPS
F1.5 - I2.1	API	Repository	Send Services metadata requirement s	Services metadata requirement s	JSON	HTTPS
F1.5 - I2.2	Repository	API	Send SHACL validation results	Validation results	JSON	HTTPS
F1.5 - I2.3	API	Repository	Request metadata per endpoint	Metadata request	JSON	HTTPS
F1.5 - I2.4	Repository	API	Send metadata per endpoint	Metadata per endpoint	JSON	HTTPS
F2.1 - I3.1	API	Repository	Request pre- processed historical data	Historical data request	JSON	HTTPS
F2.1 - I3.2	Repository	API	Send pre- processed historical data	Historical data	JSON	HTTPS

8.2.8 Universal API and Services

Table 20. Interfaces between Universal API and Services

Interfaces	Active component t/actor	Passive component /actor	Process	Data	Format	Method used
< interface's identifier>	<related active component name or actor>	<related passive component name or actor>	<process description>	< data exchanged>	<data format>	<protocol, broker, GUI, API type>
F1.1 - I19	Services	API	Services/API connection	Authenticati on info (access token)	JSON	HTTPS
F2.1 - I1.1	Services	API	Send services authenticati on to read	Authenticati on info (access token)	JSON	HTTPS
F2.1 - I1.2	Services	API	Request dynamic (real-time and historical) data	Dynamic data request	JSON	HTTPS

F2.1 - I1.3	API	Services	Send dynamic (real-time and historical) data	Dynamic data	JSON	HTTPS
F2.1 - I1.4	Services	API	Subscribe dynamic (real-time and historical) data per topic	Related data topics needed	JSON	HTTPS
F2.1 - I1.5	API	Services	Send dynamic (real-time and historical) data per topic	Dynamic data	JSON	HTTPS
F2.2 - I1.1	Services	API	Send services authentication to write	Authentication info (access token)	JSON	HTTPS
F2.2 - I1.2	Services	API	Request to write command	Signals and commands request	JSON	HTTPS
F2.2 - I1.3	API	Services	Notify success command writing	Status updates	JSON	HTTPS
F2.2 - I1.4	API	Services	Notify writing block (writing protection)	Error messages	JSON	HTTPS
F2.2 - I1.5	API	Services	Notify writing block (prioritisation logic)	Error messages	JSON	HTTPS

8.2.9 All core components and Users

Table 21. Interfaces between all core components and Users

Interfaces	Active component /actor	Passive component /actor	Process	Data	Format	Method used
<Interface identifier>	<related active component name or actor>	<related passive component name or actor>	<process description>	<data exchanged>	<data format>	<protocol, broker, GUI, API type>

F1.2 - I1	BEMS contractors	DC	Connect field devices			
F1.1 - I10	DC providers	DC	Configure DC credentials and roles to other components	DC credentials and roles	JSON if REST API	GUI, REST API
F1.2 - I2	DC providers	DC	Configure field devices on the DC	Filed devices information	-	GUI
F1.1 - I14	CC providers	CC	Configure CC credentials and roles to other components	CC credentials and roles	JSON if REST API	GUI, REST API
F1.3 - I1	CC providers	CC	Connect to and configure web services	Web services information	JSON if REST API	GUI, REST API
F1.3 - I2	CC providers	CC	Configure web services on CC	Web services information	JSON if REST API	GUI, REST API
F1.1 - I5	BuildON System integrators	DACM	Configure DACM credentials and roles to other components	DACM credentials and roles	JSON if REST API	GUI, REST API
F1.2 - I4.1	DACM	BuildON System integrators	Send validation results	Validation results	JSON	HTTPS
F1.2 - I4.2	BuildON System integrators	DACM	Approve DC/CC configuration files (annotated point list)	Action items	JSON	HTTPS
F1.1 - I1.3	BuildON Admin	Repository	Configure Repository credentials and roles to other components	Repository credentials and roles	JSON if REST API	GUI, REST API
F1.4 - I1	Data integrators	Repository	Upload static files	Static data and metadata	CSV, IFC, JSON, XML	GUI
F1.1 - I1.1	BuildON Admin	API	Register actors and users and assign roles to all actors	Actors' roles and users' information	JSON if REST API	Email, Keycloak GUI, Keycloak REST API
F1.1 - I1.2	BuildON Admin	API	Assign Repository user roles	Repository user roles	JSON if REST API	Email, Keycloak GUI, Keycloak REST API

F1.1 - I1.3	BuildON Admin	API	Request Repository credentials and roles	Credentials and roles request	JSON if REST API	Email, Keycloak GUI, Keycloak REST API
F1.1 - I1.4	API	BuildON Admin	Send Repository credentials and roles	Credentials and roles	JSON if REST API	Email, Keycloak GUI, Keycloak REST API
F1.1 - I4.1	BuildON System integrators	API	Assign DACM user roles	DACM user roles	JSON if REST API	Email, Keycloak GUI, Keycloak REST API
F1.1 - I4.2-4.9	BuildON System integrators	API	Request and send credentials/roles for all components	Credentials and roles	JSON if REST API	Email, Keycloak GUI, Keycloak REST API
F1.1 - I8	DC providers	API	Assign DC user roles	DC user roles	JSON if REST API	Email, Keycloak GUI, Keycloak REST API
F1.1 - I12	CC providers	API	Assign CC user roles	CC user roles	JSON if REST API	Email, Keycloak GUI, Keycloak REST API
F1.1 - I16	Service providers	API	Assign services to user roles	Services user roles	JSON if REST API	Email, Keycloak GUI, Keycloak REST API
F1.5 - I1	Service providers	API	Register service requirements	Metadata requirements, execution schedule/frequency	JSON	HTTPS
F1.5 - I3.1	API	BuildON System integrators	Notify validation results	Status updates	JSON	HTTPS
F1.5 - I3.2	BuildON System integrators	API	Select available/valid services for activation	Action items	JSON	HTTPS
F1.5 - I4	API	Service providers	Notify about service activation with endpoint info	Status updates and endpoint info	JSON	HTTPS
F1.1 - I18	Service providers	Services	Configure Service credentials and roles to other components	Service credentials and roles	-	GUI
F1.5 - I5	Service providers	Services	Configure service based on received endpoint info	Endpoints information	JSON if REST API	GUI, REST API

9 Deployment Plan Procedure

This section outlines the proposed technologies and security mechanisms to be used for the development of BuildON's core components of the architecture.

9.1 Technologies

The core components of the BuildON solution comprise applications which can be designed within specific platforms. In this document, the technical partners provided diagrams and descriptions of their components based on the internal structures and software subcomponents of their respective applications and platforms, as needed to satisfy both the functional and non-functional requirements.

Overall, the architecture patterns of the core components are classified into three main categories:

- **Monolithic:** This category contains applications in which the business logic components, the data access components (back-end) and the GUIs (front-end) are packaged into a single instance deployed on a specific target platform. These applications can exchange information synchronously or asynchronously with other applications through static or dynamic endpoints using various protocols and security standards.
- **Service-Oriented Architecture (SOA):** This category contains applications implementing the SOA design pattern. SOA enables modular components and services to be reused in different scenarios, reducing development time and increasing adaptability. These components are often packaged and deployed as containers in a cloud-computing infrastructure, and information is exchanged through a central messaging system, enabling asynchronous communication based on enterprise messaging standards such as JMS.
- **Microservices:** This category contains applications designed to run on multiple computational nodes. These applications are packaged as container images and deployed in a cloud computing infrastructure, providing flexibility, scalability, and high availability. In contrast with the SOA, microservices do not use a central messaging system for exchanging information. Each service instance is responsible for configuring, providing, and maintaining its endpoints, queues, and topics.

BuildON's core components (applications) follow the service-oriented and microservices architecture patterns. The proposed architecture requires service orchestration and an efficient data exchange system (message broker) that supports loose coupling between the core applications to achieve flexibility, scalability and reliability. The main objective of the service orchestration process is to effectively deploy BuildON's core applications to the available physical resources. It ensures that business logic operations run smoothly and that the available computing resources are allocated efficiently.

The deployment process will use Docker, a lightweight virtualisation system that runs on physical or virtual machines and cloud environments. 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. The distribution model of some cloud-based applications, such as CC, DACM, BuildON Data Repository, and Universal Building API, will be based on the PaaS approach, in which the technical partners are responsible for developing, running, and managing their applications without the complexity of building and maintaining the infrastructure. This

approach ensures that the allocation of hardware resources can be centrally managed using simple instructions. Meanwhile, other applications, such as the DCs, will be designed to be executed in embedded computers (gateways) with limited resources and specific hardware requirements. The traditional software installation procedure will be followed for these types of applications. Here, the software components will be packaged and distributed as installers so that BuildON users with proper permissions can download and install on the gateways.

The message broker provided by the DACM will offer a native integration environment for asynchronous communication between BuildON's core applications. It will also provide bi-directional communication between the core applications with fault-tolerance capabilities. For instance, supposing that any time consumers happen to be offline for a short period due to connectivity issues, the message broker will continue accepting messages by storing them in the configured queues. To achieve this, messaging protocols such as AMQP, KAFKA and MQTT will be evaluated based on their benefits and challenges, such as concurrency and service synchronisation issues.

9.2 Security mechanisms

BuildON's core components' design and data exchange should comply with the BuildON Data Security, Privacy and Protection (DSPP) frameworks. The specification of BuildON DSPP considers the following:

- **Default password authentication:** randomly generated strong passwords, remote provisioning, remote password (re)assignment (revocation and renegotiation)
- **Certificates:** revokable and updateable certificates (HTTPS with mTLS), expiration dates
- **Risk analysis, threat modelling and coordinated vulnerability disclosure:** STRIDE Threat modelling for software vulnerabilities (confidence limits), including a disclosure with safeguards.
- **Follow-up:** Continuous monitoring (watchdogs) and updates (e.g., libraries).
- **Encryption in transit with secure communications & authentication:** HTTPS, MQTT/TLS, SignalR / Websockets with TLS, VPNs, VLAN control and DMZs, tunnelling, channelling limitation (port forwarding), P2P Smart Grid elements/facilities and crypto agility.
- **Encryption at rest with sensitive parameters & personal data:** secure and encrypted storage (SHA-256 based) using TPM2.0.
- **Authentication and privileges:** users, hardware, and software use access control and authentication methods (role-based access control rights), such as Multi-Factor Authentication (MFA).
- **Data leakages and usage:** detection of unauthorised accesses and alerting based on telemetry.
- **Audits of events:** syslog, file accesses, database accesses, etc.
- **Resiliency:** edge computing. Local operation (isolated from communication network) when failure occurs.
- **Software stack:** secure OTA updates (to address vulnerabilities) – image signing (TPM2.0)
- **Sensitive data:** anonymisation, pseudonymisation
- **VPN channels:** at the very least, for management and control paths and data paths (depending on the use case).
- **Backups:** specify backup policies, strategies and targets.

- **Firewall:** default configurations for WAN, VLANs and DMZs, tunnelling, and port forwarding.

10 Conclusions

This document outlines the work in "Task 3.1 - Technical Requirements and Architecture Design" for the BuildON project. Its purpose is to establish a solid foundation for designing and developing technical advancements for the project. The software architecture specifications provided here aim to support the various usage scenarios (referred to as TUCs) identified during the analysis of user stories and requirements (D2.1).

The key points covered in this report are:

- Specifications for core functionalities: essential software features have been defined to support the identified TUCs.
- Conceptual architecture revision: adjustments have been made to the core components of the DoA conceptual architecture to accommodate these functionalities.
- Sequence diagrams: information flows supporting the identified functionalities using the core components have been illustrated.
- Definition of requirements: functional and non-functional requirements have been specified for each core component.
- Detailed design of core components: each core component has been thoroughly described, including its sub-components, dependencies (libraries, frameworks, tools), and interfaces.

While this work represents the initial version of the BuildON software architecture, it is important to note that assumptions have been made, and iterative updates are expected based on feedback from future developments. It should also be noted that implementing the components will help inform the various interfaces' shape, form, and messages in detail. This should help inform and refine some of the details discussed above – the reference architecture will then be more complete and will allow the specification at a good level of detail so that third parties can adapt/develop new gateways and tools to onboard the BuildON platform.