This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957047.

H2020-LC-SC3-EE-2020-1/LC-SC3-B4E-6-2020

Big data for buildings

Building Information aGGregation, harmonization and analytics platform

Project Nº 957047

D2.3 - Final technical specifications and description of the integrated BIGG solution

Responsible:	CSTB
Document Reference:	D2.3
Dissemination Level:	Public
Version:	1.0
Date:	30/11/2023

Contributors Table

DOCUMENT SECTION	AUTHOR(S)	CONTRIBUTOR(S) to results	REVIEWER(S)
Section I, V	Nicolas Pastorelly, Alan Redmond, Bruno Fies (CSTB)	N.A.	Laura El Jaouhari Balán, María Pérez Ortega (Inetum), Chris Develder (IMEC)
Section II	Nicolas Pastorelly, Alan Redmond, Bruno Fies (CSTB) Mario De Marco, Adelfio D’Angiò, Pasquale La Pietra (Intuicy)	Nico Vermeir, María Pérez, Théa Gutmacher, Jarne Kerkaert (Inetum BE)	Laura El Jaouhari Balán, María Pérez Ortega (Inetum), Chris Develder (IMEC)
Section III	Nicolas Pastorelly, Alan Redmond, Nicolas Bus, Guillaume Picinbono, Audrey Bouet, Gerson Jean-Baptiste (CSTB) Mario De Marco, Adelfio D’Angiò, Pasquale La Pietra (Intuicy) Stoyan Danov, Eloi Gabaldon (CIMNE)	Nico Vermeir, María Pérez, Théa Gutmacher, Jarne Kerkaert, Sampath Mukherjee (Inetum BE) Pierre Lehanneur, Riccardo De Vivo, Frederic Wauters (Helexia/Energis)	Laura El Jaouhari Balán, María Pérez Ortega (Inetum), Chris Develder (IMEC)
Section IV	Pierre Lehanneur, Riccardo De Vivo, Frederic Wauters (Helexia/Energis) Mario De Marco, Adelfio D’Angiò, Pasquale La Pietra (Intuicy) Stoyan Danov , Eloi Gabaldon, Jordi Carbonell (CIMNE) Stratos Keranidis, Polychronis Symeonidis (domX)	Nicolas Pastorelly, Alan Redmond, Bruno Fies (CSTB)	Laura El Jaouhari Balán, María Pérez Ortega (Inetum), Chris Develder (IMEC)

Table of Contents

Table of Figures

List of Tables

Table of Acronyms and Definitions

Acronym	Definition
AHU	Air Handling Unit
AI	Artificial Intelligence
AITB	BIGG AI Toolbox (see § III.3.3.)
API	Application Programming Interface
BDHF	BIGG Harmonized Format (BHF)
BMS	buildings management systems (BMS)
BPMN	BPMN is the chosen notation to represent the UCs. Business Process Model and Notation (BPMN) is the standard for business process modelling. It is provided by the Object Management Group (OMG).
CLI	Command Line Interface (aka “terminal”)
CMMS	Computerized maintenance management systems
DEEP	De-Risking Energy Efficiency Platform
DHW	Domestic Hot Water
DR	Demand Response (DR)
DSF	Demand Side Flexibility
ECM	Energy Conservation Measure
EEM	Energy efficiency measures (EEM)
EFFIG	Energy Efficiency Financial Institution Group
EPC	Energy Performance Certificate
EPCo	Energy Performance Contract
ES ECMS	Energy Conservation Measures (ECMs)
ESCO	Energy Service Company
ETSI	European Telecommunications Standards Institute
EUBSO	Eeuropean Union Building Stock Observatory
GDPR	General Data Protection Regulation[1]
HVAC	Heating Ventilation and Air Conditioning

[1] https://gdpr-info.eu/

INSPIRE	The INSPIRE Directive, establishing an infrastructure for spatial information in Europe to support Community environmental policies, and policies or activities which may have an impact on the environment entered into force in May 2007. INSPIRE is based on the infrastructures for spatial information established and operated by the Member States of the European Union. The Directive addresses 34 spatial data themes needed for environmental applications. See https://inspire.ec.europa.eu/
JSON	JavaScript Object Notation
LD	Linked-Data[2]
OBDA	Ontology-Based Data Access
OSS	Open-source software
OWL	Web Ontology Language
R2RML	RDB to RDF Mapping Language
RAF	Reference Architecture Framework
RDF	Resource Description Framework
RES	Renewable Energy Source
RML	RDF Mapping Language
SAREF	Smart Applications REFerence ontology
Service	Services are processes (whatever is their level) that are mainly intended to be available outside BIGG and are used by external applications consuming them to implement their own actions. Note that nothing forbids some of the BIGG own processes to consume such services.
UC	Use Case. In this document, the various use cases mentioned are taken from the D6.1 and detailed according to a chosen formalism.
W3C	World Wide Web Consortium
XML	Extensible Markup Language

[2] https://www.w3.org/wiki/LinkedData

I. INTRODUCTION

The BIGG project aims at demonstrating the application of big data technologies and data analytic techniques for the complete building life cycle of more than 4000 buildings in 6 large-scale pilot test beds. The proposed solutions will be deployed and tested cross pilot and country validation of at least two business scenarios in Spain and Greece.

The BIGG project achieves its targets by: (1) The Open Source BIGG Data Reference Architecture 4 Buildings for collection/funnelling, processing and exchanging data from different sources (smart meters, sensors, BMS, existing datasets); (2) An interoperable buildings data specification, BIGG Standard Data Model 4 Buildings, based on the combination of elements from existing frameworks and EC directives, such as SAREF, INSPIRE, BIM, EPCHub that will be enhanced to reach full interoperability of building dates; (3) An extensible, open, cloud-compatible BIGG Data Analytics Toolbox of service modules for batch and real-time analytics that supports a wide range of services, new business models and support reliable and effective policy-making.

The goal of this document is to present the technical specifications and design of BIGG Architecture building blocks. As one of the main deliverables of the WP2, it presents the overall description of the elementary BIGG components and it describes the architectural possibilities to use and organize these components. These components are software units with well-defined purposes such as retrieving data, transforming data, analyzing data or coordinating data flows.

One of the key findings of BIGG is that, to fulfil all requirements from pilots, the BIGG architecture should not be exclusively a cloud-based system. The proposed solution must be modular and flexible in terms of BIGG components deployment choices. Actually, BIGG components must be deployable locally on partners infrastructures where BIGG components can be close to the place where the data that will be exploited resides. Therefore, the BIGG technical specifications specify a “pick and choose” system with components that end-users may take and deploy individually. Architectural guidelines describe state-of-the-art ways to organize these components’ interactions.

The document is organized in the following manner:

- The first section (§II) presents the technical approach to ensure modularity and versatility of BIGG software components.
- The second (§III) section presents the Reference Architecture Framework (RAF), describing state-of-the-art techniques to coordinate BIGG components, where the actual architecture deployment be either local (on client’s infrastructures) or in the cloud (on centralized shared infrastructures). In this chapter different categories of BIGG components are introduced to manage processes such as ingesting data, harmonizing data, analyzing and improving data, exposing data to external systems, or organizing BIGG pipelines.
- The final section (§IV) describes the actual instantiation of the BIGG reference architecture for the different BIGG business cases implemented during the project. This chapter explains which BIGG components were used by the solution-providers and how they were technically integrated to build the targeted solutions.

II. TECHNICAL APPROACH

This chapter defines the overall BIGG technical architecture choices made regardless of the BIGG components specific internal logics.

These technical choices are the result of a comparative analysis of practical options based on the former experience and the technical background of the involved partners. The analysis has been driven by several concerns:

- To be able to **deliver** BIGG framework composed of **reusable components** in a form that makes their **deployment as versatile as possible** on the various envisioned IT environments,
- To allow **diverse ways of integrating BIGG components**, based on what has been observed as current working methods and planned usages.

Three distinct ways of using software components to be delivered by BIGG project have been identified while analyzing the current practices of the partners with respect to data collection and analysis:

1. **BIGG components can be used from the CLI**. This exposition method is quite a common technique for chaining tools to assemble data processing pipelines where tools' inputs and outputs are inter-connected by the means of data files exchanges.
2. **BIGG components can be used as online services by providing a webservice API**. This is a widespread practice when the involved processes require heavy computational resources, not available on the end-users' information systems.
3. **BIGG components can be used via event stream messaging or message queuing systems**. This is the best suited solution to create a BIG DATA architecture able to process live events when required.

This means that components should be distributed in 3 flavors:

- A **CLI stand-alone tool consuming and/or producing data files** and configured by options;
- A **Web service exposing a REST** (Representational State Transfer) **API;**
- An **event stream messaging system compatible** node**:** In the context of big data management use cases, event stream messaging systems are more relevant to be used than message queuing (MQ) systems. In the BIGG project, the Kafka event stream messaging system will be used for the Reference Architecture Framework. Compatible components must thus be publishers (aka "producers") and/or subscribers (aka "consumers") that can connect to the Kafka message bus.

It must be noted that these 3 forms are agnostic to the programming language used to develop the end-user applications or services leveraging the BIGG framework components.

II.1. Vision

A problem encountered frequently when deploying software in an existing environment is its compatibility with the host OS (Operating System) as well as with the installed services and software libraries. For instance, an application can have a dependency on a piece of software that already has been installed on the target system, but with a version not compatible with what is expected. Installing it without updating the dependency will result in a non-working application. Updating the dependency incurs a high risk to break other parts of the system. This is known as the “dependency hell” in the software development community.

The easiest way to solve this problem is to **use the containerization technology** popularized by **Docker.**[3] Very briefly, this is a form or virtualization, but without the overhead of stacking a guest OS over the host OS. Containers provide an isolated environment in which applications are packaged with their own dependencies and executed without interacting with conflicting ones that might be present on the host system already.

The technical artefact at the heart of Docker based deployments is the **image**. It is a kind of disk snapshot containing the code to be executed in a container.

It must be noted that such images can package long running code such as a server as well as single shot code such as a tool processing a data file. **Container technology is thus equally suited for the 3 flavors of distribution presented above**.

One option for supporting the concept of "component flavors" introduced earlier is to package the business logic of the process as a shared library which entry-points are called from the code wrapping it either as a CLI tool, as a Web service or an event messaging compatible component.

NB: It is not mandatory to code all versions of the wrapped components: involved development teams are free to choose the technical implementations that best suit the pilots’ requirements and constraints.

[3] https://www.docker.com/

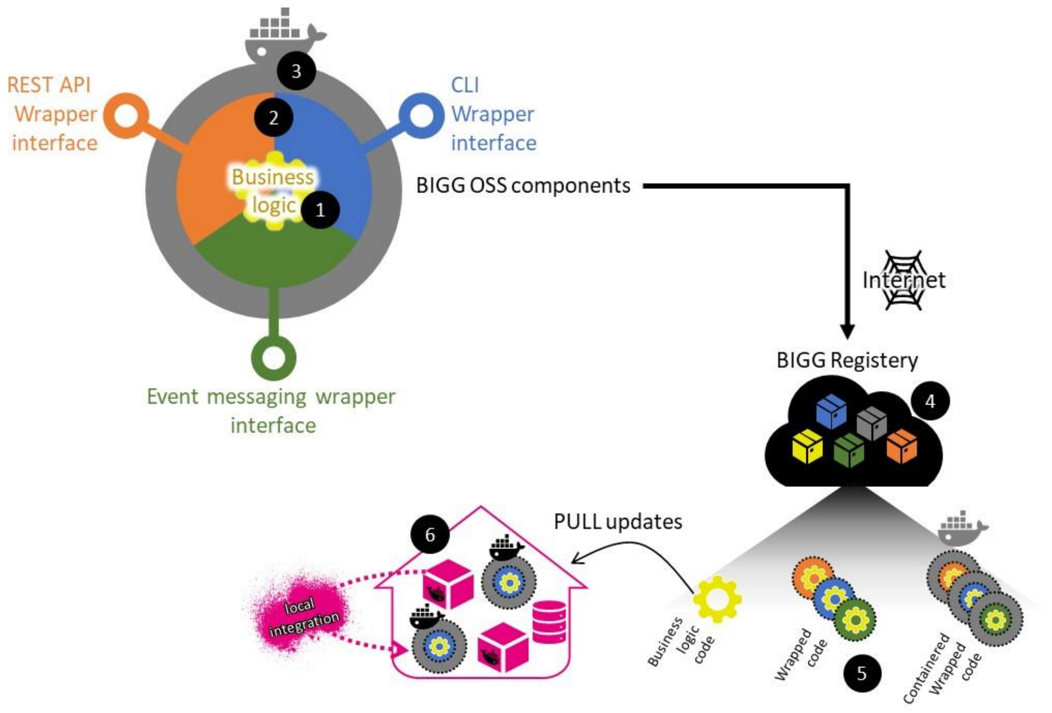


Figure 1) BIGG components flavors, distribution mechanisms and deployment

Figure 1) summarizes technical solutions to manage and share BIGG components among consortium partners:

1. For every business subject handled by BIGG (e.g., ingesting data, harmonizing data, analyzing and improving data, etc., see §III.3.), business logic software must be created. These business logic holds the added value mechanisms and features that have been created by BIGG developers specific to the business case at hand. These “low level” codes must be shared among the consortium's developers so that the code can be commonly improved, reviewed and updated in the future. These business codes usually use dedicated technologies and frameworks or libraries (e.g., Python + Pandas, R + ggplot2) to offer a software solution to a business case.
2. Business logic code can be wrapped in different interface-implementations, where options depend on the possible targeted deployments. The three options are to get business code accessible through CLI APIs, a web service REST API, and/or an event message compatible interface. For the latter point, the Kafka even streaming message system has been chosen.
3. For easiest deployments, may it be in the cloud or on local systems, the created wrapped components must be provided as Docker images. This point is very important to ensure that the components are packaged in such a way that their deployment will be “frictionless” on any targeted system.
4. The different “flavors” of the components must be published to a BIGG images registry where every partner will be able to retrieve the shared assets depending on their requirements, see §II.2.
5. Depending on each business case, components in different required flavors will be stored in the repository. The business logic code holds the added value processes that have been designed in the BIGG project. This code needs to be shared in the repository too.

Depending on what is required by each partner, versions of specific wrapped components can be created and shared, same thing for the dockerized versions of a specific service.

6. Finally, every partner implementing real-world business cases will be able to retrieve the required versions of specific BIGG components to deploy and integrate them in specific infrastructures using the BIGG RAF (see III.2.) or implementing a custom way to integrate them (see II.4.).

II.2. Distribution mechanism

A straightforward way to make images produced by contributors available to potential users is to host them on the **container registry** available on GitLab[4] for instance. This way, the repository used for the source control management of the component provides its deployment source at the same time. Running an image is then achieved with the `docker run` command once the local configuration has been set to include the URL to the image registry.

II.3. General guidelines for components

As introduced above, the components are expected to be available as CLI tools and/or as Web services exposing a REST API and/or Kafka compatible components.

Common practices for passing data and returning results are expected to be followed so that their consumers do not need to adapt to every individual choice made by the component providers. Respecting this guideline also brings the benefit of exposing a coherent and homogenous interface looking more industrial.

II.3.1. CLI integration guidelines

Command line interface[5] tools are expected to use:

- positional arguments by default for identifying datasets (both input and output).
- options for the process configuration parameters. Sensitive defaults should be supported for the options to simplify the command for commonly encountered use cases.

Example:

```
$ bigg-tool /path/to/inputfile1 /path/to/inputfile2 /path/to/ouputfile \
      --option1 foo –option2 bar
```

If the tool uses a single input dataset, it can be convenient to support piping from the `stdin` stream, allowing the end-user to build processing chains. In this case, producing the output to the `stdout` stream is expected too. Discriminating between the "file paths" and "standard streams" use cases can be achieved by reserving the positional arguments for datasets and use options for the rest. The decision is then based on the presence of the positional arguments or not. Mixed cases can be supported by using the "-" (dash) in place of a file path, but this is not mandatory.

An additional motivation for supporting standard streams is that it also brings the benefit of parallelizing processes invoked in the chain and not using the disk storage for the intermediate data if they are not to be kept. Both side-effects provide a noticeable performance boost for the execution of the whole chain, especially when high volumes of data are involved.

[4] At the time this document is written it is not decided yet if the entire BIGG repository will be publicly available or not. If the repository is restricted it will be at least accessible upon request.

[5] https://en.wikipedia.org/wiki/Command-line_interface

If standard streams are supported, care must be taken that progression messages printed by the process are written to `stderr` (and not to `stdout`) so that they don't end up in the captured data.

Exit codes must conform to the common practices of *nix systems, 0 meaning "successful command" and anything else meaning that either an error occurred during the process, or the command was invalid (missing required parameter, not existing file…)

CLI tools must provide the standard `-h/--help` option to display a usage notice the same way *nix commands do. External links to detailed documentation can be included in the returned text if relevant. They `--version` option should also be implemented to make consuming processes able to check it if needed.

II.3.2. Web service integration guidelines

Since the process will use most of the time complex inputs and datasets, its execution must be handled by a HTTP POST request what body contains the parameters encoded as `multipart/form-data.`[6] If no dataset is to be passed, the `application/x-www-form-urlencoded`[7] format can be used since it is more compact. In this case, it must be clearly stated in the specification of the tool, but this is not encouraged since it adds a burden on the end-user's side and does not bring a noticeable benefit to performance, considering the generally small size of the involved inputs.

When the process is expected to return a dataset, it must be the content of the response body and the `Content-Type` header[8] of the response must be set to reflect its type. Response custom headers can be used if additional information is to be passed. They MUST be clearly documented in this case.

The status (success or failure) of the request must use the standard HTTP status codes, namely 200 for a successful execution, 400 for an invalid request, 422 for a processing error caused by the input data and parameters provided by the request. The 500-status code must be reserved for unexpected server errors.

To make tools as self-documenting as possible, the Web service version is expected to implement the request `GET /help` returning the usage information. External links to detailed documentation can be included in the returned text if relevant. The version should also be returned in plain text as the response to the `GET /version` request.

II.3.3. Event stream messaging system (Kafka) integration guidelines

II.3.3.a. Key concepts

Kafka[9] project was started at LinkedIn and become open source later on in 2011. Since then, it has evolved and established itself as a standard tool for building real-time data pipelines. The official documentation defines it as a “distributed streaming platform” and says it is similar to enterprise messaging system. Kafka has three main components:

1. Producer

[6] https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST

[7] https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST

[8] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Type

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

2. Broker
3. Consumer

The “producers” are client application sending some messages. The “brokers” receive these messages from publishers and store them in the message log. The “consumers” read the message records from the brokers and persist them, as an example, in some repository like Cassandra, HBase, MongoDB, etc.

Over the years, a set of applications was built around Kafka to compose a whole ecosystem:

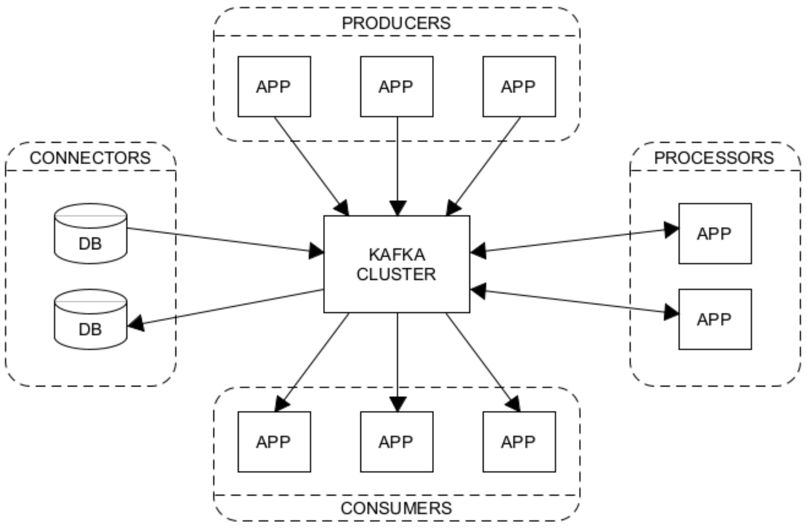


Figure 2) Kafka ecosystem

As presented in Figure 2) a “cluster” represents a set of brokers running in a group of computers. Kafka exposes stream processing API used by the “processors” and, moreover, it is possible to integrate other stream processing frameworks on top of it like Spark or Storm. The “connectors” are tools used to import data from databases into Kafka or export data from Kafka to databases. These are not just out of the box connectors but also a framework to build specialized connectors for any other application. Before implementing the integration with Kafka, it is necessary to clarify some key concepts of this message broker:

- **Producer**
  The “producer” is an application sending messages to Kafka. Each message is defined in Kafka as a “record”. A record could be a simple string or a complex object. Kafka is completely agnostic from the record format: for it is a simple array of bytes.
- **Consumer**
  The “consumer” application receives messages from the broker in a poll loop waiting for them. In Kafka there is the concept of “commit” to notify the broker the messages are received and processed correctly.
- **Broker**
  Kafka is defined as a message broker because it acts as an intermediary between the producers sending messages and consumers receiving them. Notably, producers and consumers are not directly connected: Kafka offers a loosely coupled integration architecture.
- **Cluster**
  Generally, a cluster is a group of computers acting together for a common purpose.

This is the same for Kafka: deployed in a distributed system, each computer is executing one coordinated instance of the broker.

- **Topic**
  It is an arbitrary unique name given to a data stream that comprises all messages its producers send and its consumers subscribe to.

- **Partition**
  A topic can contain a huge amount of data. Storing and processing such quantity of data can be scaled by dividing all the messages sent to a topic over multiple “partitions”. As an example, a Kafka cluster can organize and distribute each topic partition on a different computer. The number of topic partitions is determined by the user, Kafka is not involved in this decision. Different configurable criteria can be used to divide messages among the partitions: range, hashing, round-robin, etc.

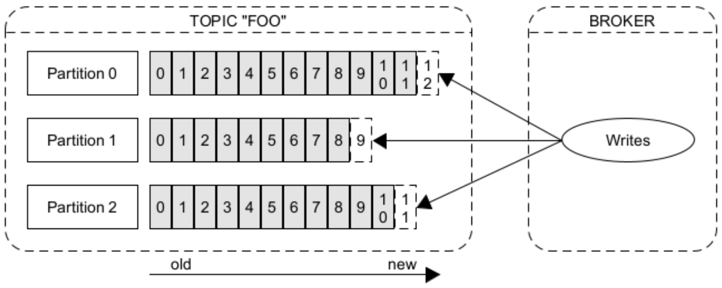


Figure 3) Kafka partitioning

- **Offset**
  The “offset” is simply the sequence number Kafka attributes to each message in a partition. This number is immutable and for the first message in a topic is zero and then is incremented by 1 for the next message and so on. The offset is local to a partition and is not globally indicating a message across the cluster. So, to directly identify a message it is necessary to know: the topic name, the partition number and the offset number. The offset allows the consumer to “commit” the exact message received, allows Kafka handle correctly the message sequence and allows operators to shift the current offset that Kafka is about to process to rewind the message sequence.

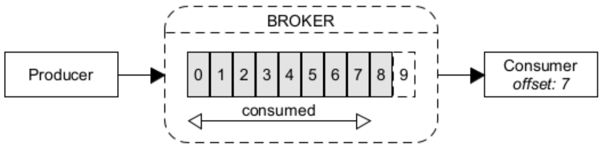


Figure 4) Kafka offsets management

- **Consumer Group**

  If the partitioning of a topic is a mechanism to assure scalability to the side of the producers, the possibility to organize groups gives scalability to the side of the consumers. In each group there could be instantiated more consumers to share the workload, each component of a group consuming messages from a different set of partitions. Usually, consumers doing the same work belong to the same consumer group and they must be fewer than the topic partitions: the consumers in excess will not receive any message and they will be eventually used as backup.

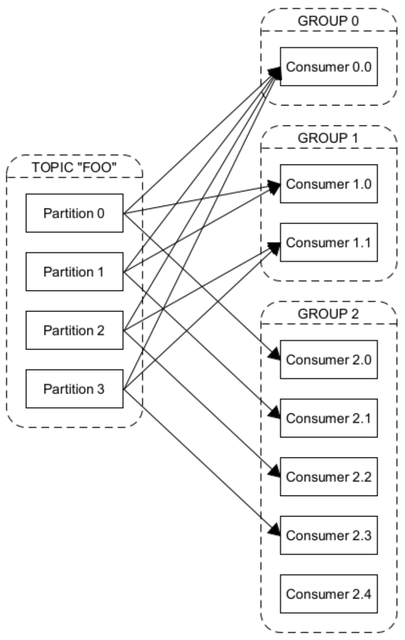


Figure 5) Kafka consumer groups

Moreover, Kafka adopts a rebalancing policy when the partitions are more than the consumers in a group as shown in Figure 6):

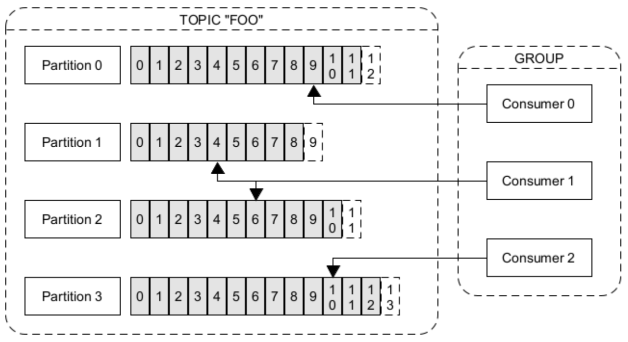


Figure 6) Kafka rebalancing policy

II.3.3.b. CLI

In Kafka installation home directory, the “bin” folder contains some useful commands:

```
# CREATE TOPIC

kafka-topics.sh --zookeeper localhost:2181 --create --topic <topic name> --
partitions 1 --replication-factor 1 --config <configuration>


# DESCRIBE TOPIC

kafka-topics.sh --zookeeper localhost:2181 --describe --topic <topic name>


# DELETE TOPIC

kafka-topics.sh --zookeeper localhost:2181 --delete --topic <topic name>


# LIST ALL TOPICS
```

```
kafka-topics.sh --zookeeper localhost:2181 --list


# FOLLOW TOPIC ON THE CONSOLE

kafka-console-consumer.sh  --bootstrap-server  localhost:9092  --topic  <topic
name> --from-beginning


# DESCRIBE CONSUMER GROUP

kafka-consumer-groups.sh  --bootstrap-server  localhost:9092  --describe  --
group <consumer group name>


# SET THE OFFSET TO A CONSUMER GROUP

kafka-consumer-groups.sh --bootstrap-server localhost:9092 --group <consumer
group name> --topic <topic name>:<partition> --execute --reset-offsets --to-
offset 1636

kafka-consumer-groups.sh --bootstrap-server localhost:9092 --group <consumer
group name> --topic <topic name>:<partition> --execute --reset-offsets --to-
latest
```

II.3.3.c. CODE

The examples included in this section use Java language. Kafka clients are available for a large panel of programming languages. For instance, Python users can head to `kafka-python`,[10] which is among the most popular ones.

II.3.3.c.1. Producer

A "producer" could be implemented with very few lines of code:

```
// istantiate a producer

KafkaProducer<String, Object> producer = new KafkaProducer<String,
Object>(<producerPropertiers>);


// instantiate a record

ProducerRecord<String, Object> record = new ProducerRecord<String, Object>(topic,
partition, timestamp, key, data, headers);


// send a record optionally with a callback

producer.send(record, <optional callback>);
```

II.3.3.c.2. Consumer

A "consumer" must implement a poll loop in a thread to keep waiting for batch of records coming from the broker, a possible pseudo-code implementation could be:

```
// define consumer

KafkaConsumer<String, byte[]> consumer = new KafkaConsumer<>(consumerProperties);
```

[10] documentation: https://kafka-python.readthedocs.io/en/master/

```
// map of offset to be committed to the broker (to empty on each poll)
Map<TopicPartition, OffsetAndMetadata> toCommitMap = Maps.newHashMap();

// subscribe to list of topics or a pattern
consumer.subscribe(topics);

[...]

<poll loop>
ConsumerRecords<String, byte[]> records = consumer.poll(pollTimeout);

// record loop
for (TopicPartition partition : records.partitions()) {
List<ConsumerRecord<String, byte[]>> partitionRecords = records.records(partition);
for (ConsumerRecord<String, byte[]> record : partitionRecords) {

// consume record
[...]

// prepare for commit up to this record
long lastOffset = partitionRecords.get(partitionRecords.size() - 1).offset();
toCommitMap.put(partition, new OffsetAndMetadata(record.offset() + 1));

// commit map
consumer.commitSync(toCommitMap);

[...]
```

II.4. BIGG components versatile integration options

The 3 BIGG components distribution mechanisms proposed in previous chapters imply different technical options to provide input and output data to these components. Table 1 presents the different input and output possibilities depending on the chosen distribution system as soon as the components are dockerized:

	INPUTS	OUTPUTS	#
CLI	(Via environment variables) Shared external storing systems reference as input	Shared external storing systems reference as output	1
	command line arguments		2
	STDIN (File stream)	STDOUT (File stream)	3
	Mounted folder containing input files	Mounted folder containing output files	4
Webservice	Webservice API call	Webservice API response	5
Event stream messaging system	Input messages	Output messages	6

Table 1: BIGG sample integration options

1. A dockerized component exposing a CLI interface can use **environment variables to share connection to external persistent storing systems** like databases. Reference to external storage system can then be away to share input and output data for BIGG components.
2. A dockerized component exposing a CLI interface can **use command line arguments to provide input data do BIGG components**. Command line inputs contain variables providing data to be processed by the component.
3. In computer programming, standard streams are interconnected input and output communication channels between a computer program and its environment when it begins execution. The three input/output (I/O) connections are called standard input (**STDIN**), standard output (**STDOUT**) and standard error (stderr).[11] A dockerized component exposing a CLI interface can **use STDIN to get input data as a stream and provide outputs to STDOUT as a stream**.
4. A dockerized component exposing a CLI can **use the shared folder mounting feature of docker** to share a directory that could be used to get input files and provide output storage for result files created by the process.
5. If a dockerized component is using a **web service front end** to expose services proposed by the component, then a **REST API request** (via HTTP POST/GET/PUT…) can be used as an input while the standard **HTTP response** can be used for the output of the service.
6. In the case where a dockerized component is using an event stream messaging system, then the **messaging protocol** proposed by the event stream messaging system **can be used to provide input data** to the component, the component can **use the messaging protocol to provide output information** to the system (e.g see Kafka concepts in section II.3.3.).

   It must be said that components exposing web services or using event stream messaging systems can be triggered through an HTTP request or a message event with parameters but can still use a shared resource like a shared database or shared folders to deal with input and output data when a huge amount of data is involved in the process. For instance, passing via a HTTP request a big amount of data to be processed by a micro-serviced component may not be a robust technical strategy.

[11] https://en.wikipedia.org/wiki/Standard_streams

Here there is a docker exec command describing how a BIGG component could be triggered via a command line with options to provide input parameters to the service:

```
EXEC docker run -it --rm <image> <env Variables> <command_and_options> (intputs) | STDIN
```

With this vision of BIGG components exposing various possible technical interfaces implementations, different integration options can be envisioned to build processing pipelines.

Figure 7) is an example of different BIGG components integration via a custom integration mechanism.

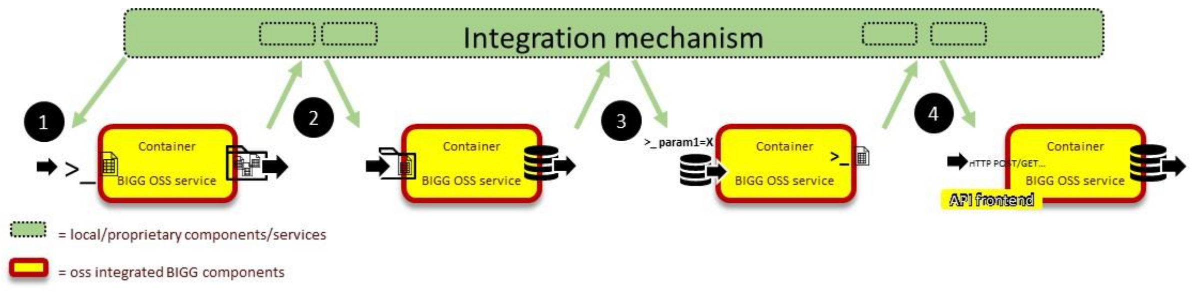


Figure 7) Example of a custom BIGG components integration solution

1. A custom integration mechanism calls the first component of a pipeline using STDIN to provide input information and variables to mount a shared folder using Docker features. The component outputs are stored in the shared folder as files.
2. The custom integration mechanism gets result files from the shared folder and passes the shared folder reference (as input for the next component in the pipeline) along with connection parameters to a shared database. The second component then uses the shared database to store its results after processing files provided by the first component in the shared folder.
3. The integration mechanism is triggered by the third component and provides it a reference to the shared database and extra variables in the CLI input. This component is generating the result on STDOUT processing data provided as parameter and data stored in the shared database.
4. The integration mechanism then uses the content provided by the previous component on STDOUT to push it as a parameter of a REST API call to trigger the next component which will store his processing results in a shared database.

This fictious sequence is proposed to display how BIGG components expose versatile options for adaptable pipelines integration. It demonstrates that various methods to handle input data and output data can be implemented by diverse systems from the simplest one (e.g., a simple script to chain several components) to the more complex one using an event stream messaging system and the RAF described in the next Section III.

For instance, some data scientists could use some of the BIGG dockerized components in direct local integration with tools like Jupyter Notebook[12] or Matlab[13] via CLI calls.

[12] https://jupyter.org/

[13] https://www.mathworks.com

III. RAF – Reference Architecture Framework

III.1. Big data key concepts

III.1.1. Definition

Big Data is a complex set of concepts, technologies, frameworks, architectures having a single simple objective: to manage a big amount of data. Its definition is often based on words starting with "V"[14] and, despite more and more V's have been added over time (up to ten), the standard definition counts 5 Vs, as sketched in Figure 8)

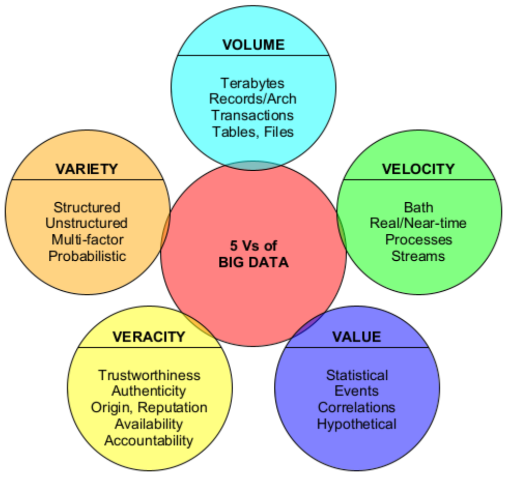


Figure 8) Big Data 5 Vs

<u>Volume</u>

Big data volume defines the ‘amount’ of data that is produced. The value of data is also dependent on the size of the data. Today data is generated from various sources in different formats – structured and unstructured. Some of these data formats include Word and Excel documents, PDFs and reports along with media content such as images and videos. Due to the data explosion caused to digital and social media, data is rapidly being produced in such large chunks, it has become challenging for enterprises to store and process it using conventional methods of business intelligence and analytics. Enterprises must implement modern business intelligence tools to effectively capture, store and process such unprecedented amount of data in real-time.

<u>Velocity</u>

Velocity refers to the speed at which the data is generated, collected and analyzed. Data continuously flows through multiple channels such as computer systems, networks, social media, mobile phones etc. In today’s data-driven business environment, the pace at which data grows can be described as ‘torrential’ or ‘unprecedented’. Now, this data should also be captured as close to real-time as possible, making the right data available at the right time. The speed at which data can be accessed has a direct impact on making timely and accurate business decisions. Even a limited amount of data that is available in real-time yields better business results than a large volume of data that needs a long time to capture and analyze.

[14] Laney, D. (2001), “3-D Data Management: Controlling Data Volume, Velocity and Variety”

Several Big data technologies today allow us to capture and analyze data as it is being generated in real-time.

Value

Although data is being produced in large volumes today, just collecting it is of no use. Instead, data from which business insights are garnered add ‘value’ to the company. In the context of big data, value amounts to how worthy the data is of positively impacting a company’s business. This is where big data analytics come into the picture. While many companies have invested in establishing data aggregation and storage infrastructure in their organizations, they fail to understand that the aggregation of data doesn’t equal value addition. What you do with the collected data is what matters. With the help of advanced data analytics, useful insights can be derived from the collected data. These insights, in turn, are what add value to the decision-making process. One way to ensure that the value of big data is considerable and worth investing time and effort into, is by conducting a cost vs. benefit analysis. By calculating the total cost of processing big data and comparing it with the ROI that the business insights are expected to generate, companies can effectively decide whether or not big data analytics will actually add any value to their business.

Veracity

The Veracity of big data or Validity, as it is more commonly known, is the assurance of quality or credibility of the collected data. Can you trust the data that you have collected? Is this data credible enough to glean insights from? Should we be basing our business decisions on the insights garnered from this data? All these questions and more, are answered when the veracity of the data is known. Since big data is vast and involves so many data sources, there is the possibility that not all collected data will be of good quality or accurate in nature. Hence, when processing big datasets, it is important that the validity of the data is checked before proceeding with processing it.

Variety

While the volume and velocity of data are important factors that add value to a business, big data also entails processing diverse data types collected from varied data sources. Data sources may involve external sources as well as internal business units. Generally, big data is classified as structured, semi-structured and unstructured data. While structured data is one whose format, length and volume are clearly defined, semi-structured data is one that may partially conform to a specific data format. On the other hand, unstructured data is unorganized data and does not conform with traditional data formats. Data generated via digital and social media (images, videos, tweets, etc.) can be classified as unstructured data. The sheer volume of data that organizations usually collect and generate may look chaotic and unstructured. In fact, almost 80% of data produced globally including photos, videos, mobile data, and social media content, is unstructured in nature.

III.1.2. Layers

In order to offer all the features promised by its 5 V's definition, a Big Data architecture is typically organized in layers, as sketched in Figure 9)

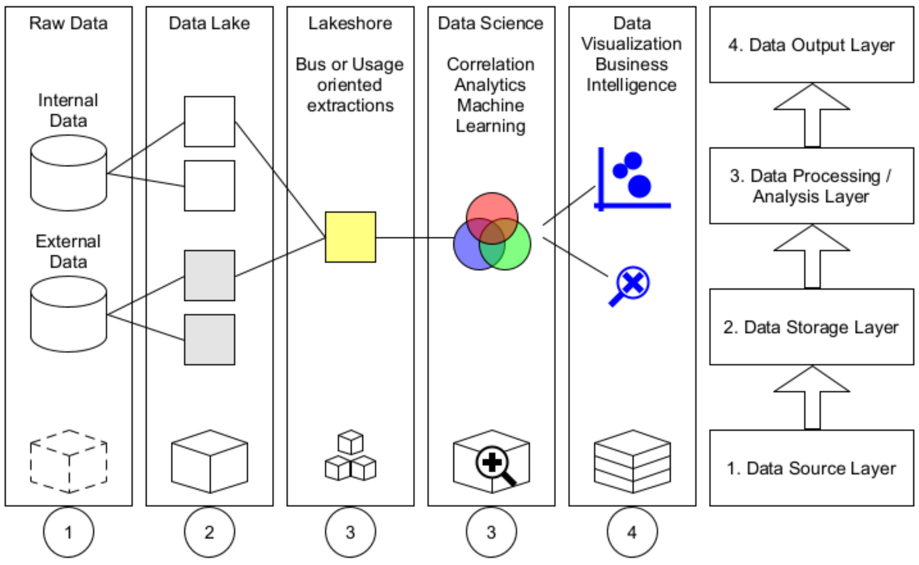

Raw Data
Internal Data
External Data
Data Lake
Lakeshore
Bus or Usage oriented extractions
Data Science
Correlation Analytics Machine Learning
Data Visualization Business Intelligence
4. Data Output Layer
3. Data Processing / Analysis Layer
2. Data Storage Layer
1. Data Source Layer
1
2
3
3
4

Figure 9) Big Data architecture layers

1) Data Source Layer

The very first step is to collect data both internal and external, in both modalities push and pull, from different sources (smartphone, networking, sensor, social media, health, etc.) in different formats (structured or unstructured). The streaming data will be fed into the processing layer, and the accumulated historical data will be stored in the storage layer, to be further analyzed with specific analytical tools in the analytical layer, based on the demands from the application layer.

2) Data Storage Layer

In this layer all the incoming raw data will be stored persistently. Usually, a generic purpose NoSQL repository is used to agnostically store the messages (Hadoop, MongoDB, etc.) following a "schema on read" approach, in other words applying a structure to the data only in the extraction phase. This type of storage is typically called a "data lake" because all the data sources are pouring content into it like a river in a lake. To retrieve the data stored in the data lake, its content needs to be enriched with labels and metadata.

3) Data Processing / Analysis Layer

When you want to use the stored data and find out something useful, you will need to process and analyze it. This layer is responsible for acquiring data from the data lake and, if necessary, converting it to a format that suits how the data is to be analyzed. The concept of "lakeshore" is introduced to define a repository containing structured, mapped and organized data, derived from the raw data contained in the data lake. A lakeshore fits the data model used by the analysis layer to feed ML modules and to extract statistics, business intelligence, AI models, etc.

4) Data Output Layer

This is how the insights gleaned through the analysis is passed on to the people who can take action to benefit from them. This output can take the form of reports, charts, figures and key recommendations. Ultimately, your Big Data system's main task is to show, at this stage of the process, how measurable improvement in at least one KPI that can be achieved by acting based on the analysis you have carried out. The consumers can be visualization applications, human beings, business processes, or services. Real-time analysis can leverage NoSQL stores (for example, Cassandra, MongoDB, and others) to analyze data produced by web-facing apps.

III.1.3. Lambda and Kappa Architecture

One of the first generic Big Data reference architecture was designed by Nathan Marz. The Lambda Architecture[15] conceives the "data source layer" as a continuous stream of data that splits into two separate flows: the "batch layer", that stores all the historical data and pre-computes the views to be offered to the "serving layer", and the "speed layer" that stream processes the incoming data and offers a real time view to the "data access layer". This last has a "strabic" view on the last layers: it accesses both "serving layer" and "speed layer" merging and collecting the best quality of data according to the different use cases, as sketched in Figure 10):

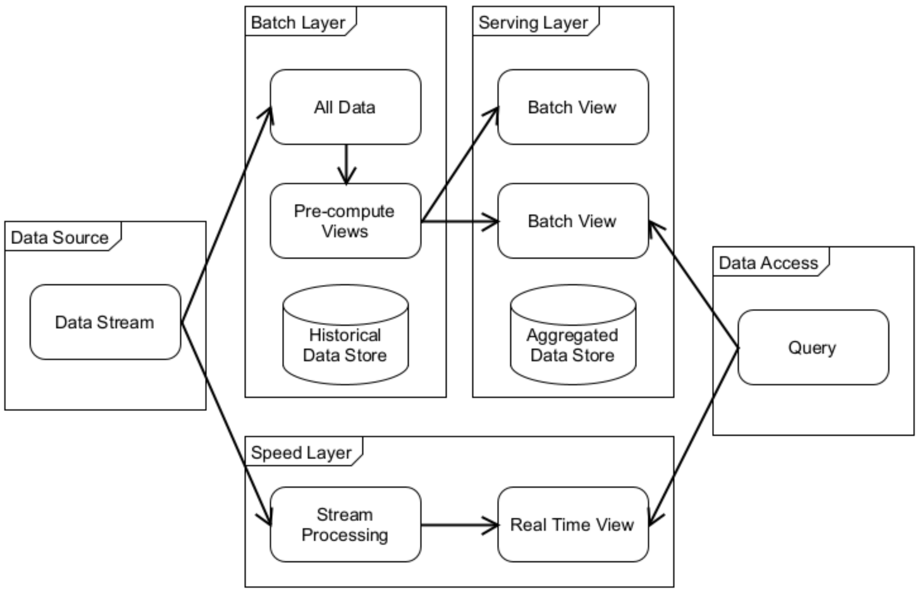


Figure 10) Big Data Lambda architecture

An alternative generic reference architecture was successively designed by Jay Kreps. The Kappa Architecture[16] focuses only on data processing as a stream. It is not intended to replace the Lambda architecture but rather to simplify it. The idea is to manage data processing in real time and continuous reprocessing in a single flow processing engine. All reprocessing is done starting from the data stream. This requires that the incoming data stream can be played again (very quickly), either in its entirety or from a specific point. If there are any changes to the code, a second processing of the flow proceeds to a new reproduction of all the previous data through

[15] https://en.wikipedia.org/wiki/Lambda_architecture and https://www.oreilly.com/radar/questioning-the-lambda-architecture/

[16] http://milinda.pathirage.org/kappa-architecture.com/

the last engine in real time and to the replacement of the data stored in the service level. This architecture aims at simplification by maintaining a single code base rather than managing one for each level, batch and speed, as in the Lambda architecture. Also, queries need to search in one service location only, rather than accessing batch and speed views, as illustrated in Figure 11):

For many real-time scenarios, the Lambda architecture is perfect. The same cannot be said for Kappa architecture. If batch and streaming analytics are at the same level, the Kappa architecture is probably the best solution. In some cases, however, accessing a complete set of data in a batch view can lead to a level of optimization that makes the Lambda option more performing and perhaps even easier to implement.

There are also some extremely complex situations in which batch and streaming algorithms produce very different results (using machine learning models, advanced systems or naturally very expensive operations that must be performed differently in real time) that require the use of Lambda option.

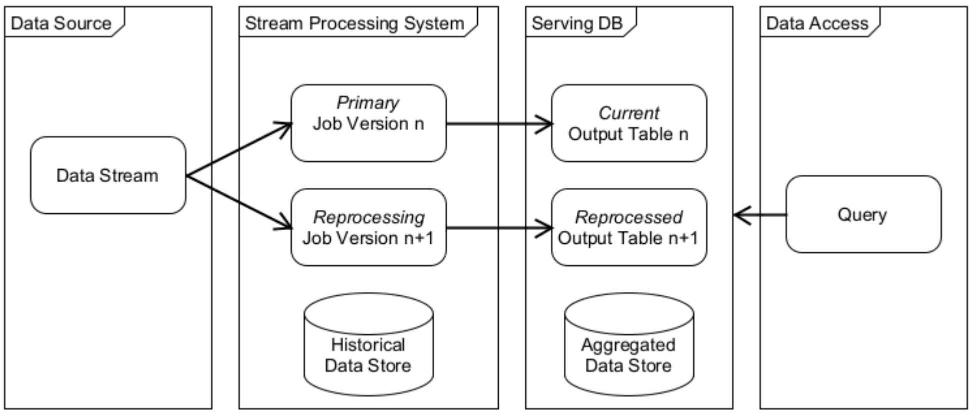


Figure 11) Big Data Kappa architecture

III.2. BIGG RAF description

III.2.1. The concept of harmonized data

Building energy information gathering techniques have evolved significantly, driven by technological advancements and concerns about energy efficiency and sustainability. These technological advancements have enhanced data accuracy and availability, opening new possibilities for optimizing energy performance and driving sustainable practices in the built environment. However, the lack of standardization and harmonization of data definitions across diverse applications and databases poses a critical challenge to realizing these benefits. The BIGG project recognizes this challenge and endeavors to overcome it by developing an open-source Big Data Reference Architecture and a BIGG AI Toolbox (AITB) capable of supporting diverse use cases and applications throughout the building life cycle. **The BIGG project's essence revolves around harmonizing data across various systems and platforms by harnessing the power of semantic technologies, particularly ontologies.** Ontologies enable a structured and standardized representation of data meaning and interrelationships in machine-processable formats, fostering common vocabularies and shared understandings, thereby facilitating seamless data exchange and integration across heterogeneous systems. An essential principle in BIGG Standard Data Model 4 Building Ontology[17] development is the reuse and alignment of existing ontologies. By building upon and extending established ontologies in relevant domains, the BIGG project avoids redundant efforts, while ensuring compatibility and consistency between diverse data sources. Reusing and aligning existing ontologies create a robust foundation for connecting and relating heterogeneous datasets, preserving the original context of information.

III.2.2. RAF overview

Using the data harmonization concept, the detailed RAF is presented in the Figure 12)

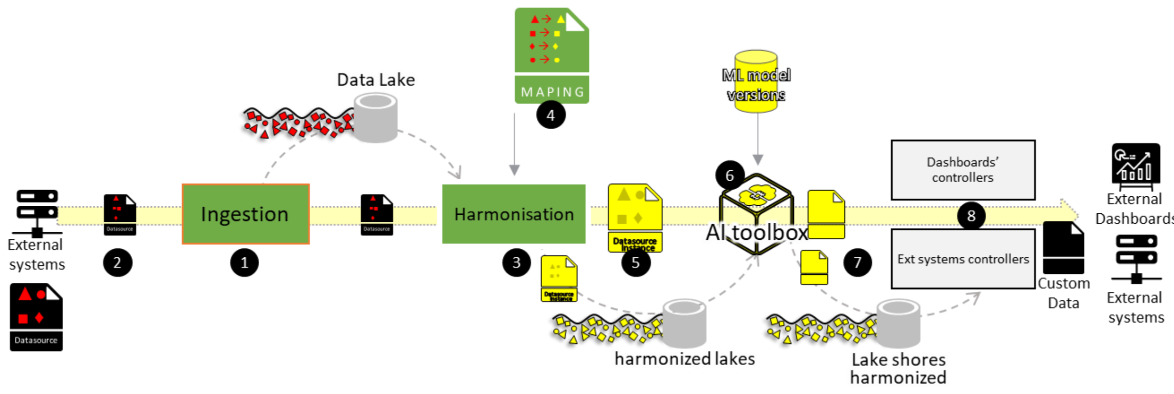


Figure 12) BIGG RAF overview

In the following description, the numbers will make reference to black numbers in Figure 12).

The RAF comprises a pipeline of several components that can be orchestrated in different ways, using a message broker is the most effective (see next section). The first step uses BIGG ingestion modules (1) that are software components to acquire specific data (2) from different custom sources using standard protocols like HTTP or MQTT for instance (see §III.3.1.). These ingested data are stored in data lakes for later processing and/or directly

[17] https://github.com/biggproject/Ontology

pushed through an event message bus like Apache Kafka[18] to the harmonization module (3). The harmonization module (see §III.3.2.) is flexible enough to only require a standardized mapping file (4) to enable conversion of the initial specific data to the BIGG Standard Data Model 4 Building (5) required as input by the BIGG AI Toolbox (AITB, cf.6).

The data harmonization is the process of bringing data from different sources into a consistent format, the BIGG Standard Data Model 4 Building Ontology,[19] making it easier to analyze and use. Harmonization is important because it improves data quality, consistency, and integration, enabling organizations to make better decisions and improve their operations. In the context of the AITB, data harmonization is essential for the success of data pipelines, which extract valuable insights and predictions related to building energy consumption and performance. By ensuring that data inputs into the pipelines adhere to standardized formats and terminologies, data harmonization enables the pipelines to seamlessly use this harmonized data across various applications and analysis tasks. Overall, data harmonization is a critical step towards realizing data-driven excellence and transforming data into an asset that propels an organization's operations and growth endeavors.

The harmonized data can be stored in "harmonized data lakes" (e.g., triple stores) and/or directly pushed through an even bus to the toolbox. The BIGG toolbox leverages several data analytics and AI/ML curated technologies to process the building/energy related data and produce insights output in the BIGG Standard Data Model 4 Building (7). The Toolbox uses machine learning techniques to produce valuable predictions (see § III.3.3.). The knowledge created by the toolbox can be stored in "harmonized lakes shores" and is used by specific dashboard and external system software components (see 8, § III.3.4.) that will transform back the harmonized data into the data formats required by external dashboards or systems (like for instance any building management systems autonomously controlling buildings' appliances based on BIGG system predictions).

The following sections will describe the RAF elements in more details.

III.2.3. RAF connectivity: Message Broker

Whatever the generic reference architecture to be used, either Lambda or Kappa, it is certainly not possible to ignore a fundamental element present in both: the message broker. This software module can be considered as a message-oriented middleware, an intermediary agent that connects all the applications with each other. In a microservices-based architecture, the message broker can offer a lot of features:

- topic-based message routing with publisher-subscriber pattern
- message routing to one or more microservices
- message queuing for batch workloads
- enhanced and loosely coupled services interoperability
- message storage and/or buffering to guarantee delivery
- partitioning and load balancing
- event-driven choreographed architecture

To use a Message Broker tool, 3 fundamental service archetypes can be identified, illustrated in Figure 13):

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

[19] See D4.2 Description of the final harmonization layer

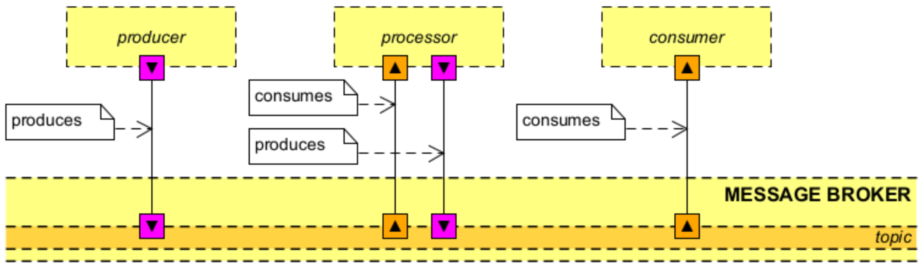


Figure 13) 3 fundamental service archetypes for a Big Data architecture

1. Producer

It is a service that publishes a message on a specific topic. As an example, a "producer" could be a service collecting IoT messages and publishing them on a specific message broker topic. Such a service is commonly indicated as "ingestor".

2. Processor

A “processor” both consumes a message from a topic and, after some computations or transformations, produces a message on another topic. A service of this archetype could be used to harmonize the incoming messages translating them from their original formats in a “common language” defined by the standard BIGG models.

3. Consumer

This archetype of services consumes message from a topic (or more topics) for different purposes: to send the message to another service, to persist the message into a repository, to produce logging or tracing, etc. This kind of services are usually identified as “adapter”. Consumers subscribe to messages for listening to them and processing the conveyed or related data on the fly.

III.2.4. RAF description

III.2.4.a. Requirements

In the BIGG project, a participant could have his own IT infrastructure: edge computing components based typically on IoT hardware like sensors, devices and meters, and a cloud computing platform probably containing a message broker, a data lake, and one or more lakeshores so that he could want only to use some of the analytics services offered by BIGG. Alternatively, a participant could have none or a subset of these elements, needing a platform that could offer all the BIGG services. Moreover, a participant (or a simple user) may need to deploy the BIGG solution locally on a development PC for testing purposes, on premise in his company, or on a shared cloud platform. In some scenarios, a participant could want to use BIGG as it is, out of the box, in another scenario, instead, he could want to modify and customize some of its components or to completely replace them.

To meet all these scenarios and requirements, the BIGG RAF must be:

1. **Big Data enabled**: BIGG RAF must be a Big Data architecture offering its minimal set of backbone services like a message broker, a data lake and at least one lakeshore.

2. **Virtual / Containerized:** Virtualization is the best solution to install the BIGG platform in whatever environment the participants want to. This can be achieved with OS level virtualization preparing a virtual machine image to run on the proper container (VMware,

VirtualBox, etc.) or virtualizing every platform component starting from the backbone (message broker, datalake repository, etc.) up to the single microservice (ingestor, Harmonizer, etc.) into some single process containers using, as an example, Docker.

3. **Pluggable:** If a participant wants to use the BIGG platform with a "black box" approach, then this platform has to expose clearly defined and well documented input / output services.

4. **Modular / Composable:** A microservices oriented architecture is, by definition, modular and composable and could offer to the participants the possibility to modify or replace a single component or to even reorganize the data flow and the business processes by simply reconfiguring the sequence or the pipeline (i.e. changing input and output topics on the message broker).

III.2.4.b. Design

The diagram in Figure 14) shows a detailed version of the BIGG reference architecture.

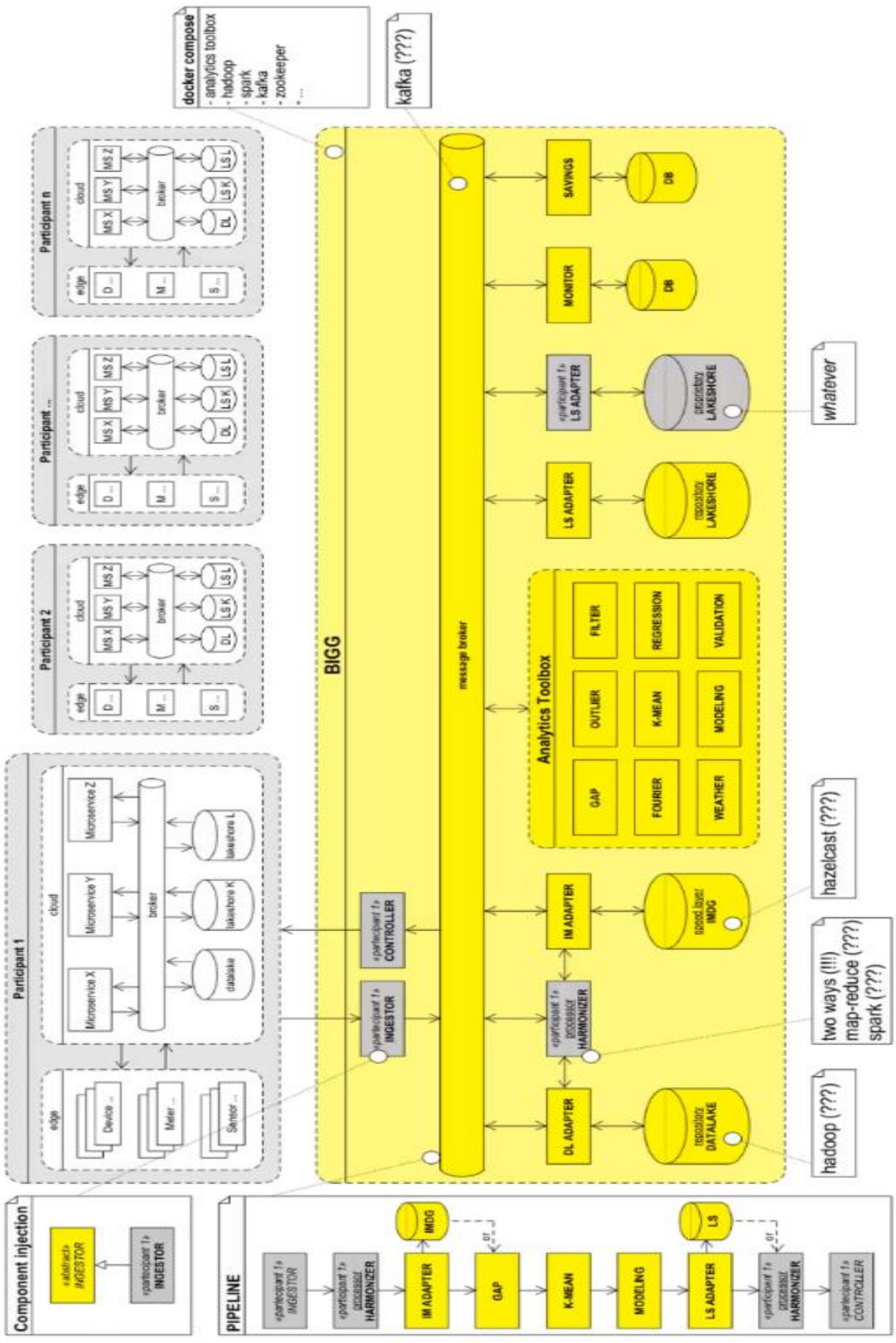


Figure 14) Overview of BIGG reference architecture

In the top there are grey boxes representing the participants IT infrastructure and in the yellow box in the bottom is the BIGG RAF. These are its main generic components:

Ingestor

This microservice is the BIGG platform input. It is a “producer” exposing a REST API to receive all incoming data and to put them on the message broker proper topic

Controller

The “Controller” is an “adapter/consumer” retrieving all the response data from the dedicated message broker topic and sending them to the participant IT infrastructure. It is the output of the BIGG platform.

Message Broker

This component will be the backbone of the whole solution interconnecting all the microservices to enact a choreographed data pipeline. Apache Kafka could be the proper tool to use as the message broker.

Harmonizer (see §III.3.2.b.)

Following the “producer” pattern, this microservice will read messages from the message broker, will translate them in the standard BIGG models and will write the results on the message broker. It will expose this translation functionality in both directions: from participant proprietary language to common language and from common language to participant proprietary language to make the result readable by the third-party external participant system. As an alternative, this service could be implemented as a Hadoop map-reduce task or as a Spark streaming job.

DL Adapter and Datalake

An “adapter/consumer” is a service consuming messages from the broker and sending them to a third-party system, to a repository or to a different outbound. In this case the DL Adapter is used to persistently store all the messages it retrieves from the broker into the data lake. It is also used to retrieve the same messages from the data lake. It is, in other word, the interface used by the BIGG platform to its central repository. For this kind of usage, a generic purpose repository should fit the needs: HBase on Hadoop or MongoDB.

IM Adapter and IMDG

The speed layer is a critical component. Low latency in memory computations could be useful for timeseries real time analysis and monitoring. The IM Adapter component will offer an interface to an in-memory repository capable of processing up to 200,000 transactions per second. Redis and Hazelcast are two tools that can address this need.

Analytics Toolbox (see §III.3.3.)

This is the very BIGG platform core: a set of services that will add value, quality and insight to the incoming data:

- Gap detection – to rise alerts
- Outlier detection – to fine-tune the timeseries
- Timeseries filtering – to isolate interesting time or value intervals
- Fourier transform – to obtain a frequency-based domain model
- K-Mean – to group datasets with a fast-clustering algorithm
- Regression – to find correlation between variables
- Weather – to analyze weather data
- Modeling – to compute predictive models and timeseries digital twins
- Validation – to validate datasets and timeseries

LS Adapter and Lakeshore

As an opposite to the data lake approach, a lakeshore contains an already harmonized, transformed, eventually aggregated dataset that could be served at once to the analytic services layer. This adapter will store and retrieve computed data to and from a time-series specialized repository that could be Apache Cassandra or InfluxDB.

Services as Monitor, Savings, Tracing, etc

Due to the flexibility offered by the microservices architecture and the message broker, it will be relatively simple to add new services like for monitoring, for tracing, for savings estimation, etc.

III.2.4.c. Distribution

This architecture will be distributed as a "docker compose" text file listing all the single component docker image so that at once will be put up and running (as an example):

- Kafka
- Zookeeper (required by Kafka)
- Hadoop
- HBase
- Spark
- The Analytics Toolbox (defined as a "docker compose" in turn)
- All the microservices one by one

In this way it will be quite simple to download from the shared repository and deploy the BIGG infrastructure on whatever environment the single participant wants to. Moreover, some relevant components (Ingestor, Controller and Harmonizer) will be distributed in a base implementation that could be "overridden" (or completely replaced) by a customized implementation done by the different participants. This sort of "component injection" can be easily achieved by simply editing the "docker compose" text file.

As described in D2.1 deliverable, there are two families of components mainly:

- **Data harmonization components**, responsible for converting external data to and from to BIGG harmonized format aimed at easing the use of tools provided by the platform
- **Data analysis components,**[20] responsible for the heavy lifting job of processing the incoming data related to energy consumption to produce the KPIs (Key Performance Indicators) relevant for decision making.

This classification is kept hereafter to help identify commonalities that can emerge.

III.3. RAF components explained

III.3.1. Ingestor Components

III.3.1.a. Overview

The Ingestor is a micro-service responsible to receive inbound transmissions about generic data, anagraphical data, measurements, events and statuses from field devices, field data-logger or external services, in order to route this data internally in the system.

[20] aka "AI Toolbox"

The Ingestor must be:

- **Unsolicited:** listening to incoming transmissions and activating itself as reaction to external activity.
- **Available:** keeping operative as much as possible, limiting at all possible the downtime of the service
- **Responsive:** responding in a timely manner if at all possible.
- **Generic and durable:** managing incoming transmission in generic way, without claiming to understand what's the nature or the content of the transmission; requiring no implementation and no configuration to receive new types of transmissions, if at all possible.
- **Efficient:** engaging a minimal amount of computational resources.
- **Scalable and elastic:** allowing more instances of the micro-service to run in the system on different nodes to (possibly linearly) increase the throughput; making it easy to add, remove or move instances across the system.
- **Robust:** coping with errors and with erroneous inputs without compromising the operation of the system.

III.3.1.b. Architecture

The Ingestor is the entrance point for data transmissions coming from the field devices and external systems. Data are pushed to the "input" topic as soon as possible. From there, messages are ready to be consumed by the Harmonizer microservice. It is an HTTP based ingestor, it uses Spring Boot REST. The ingestor also uses Spring Kafka, all properties are configurable using the present Spring Kafka properties (see Figure 15).

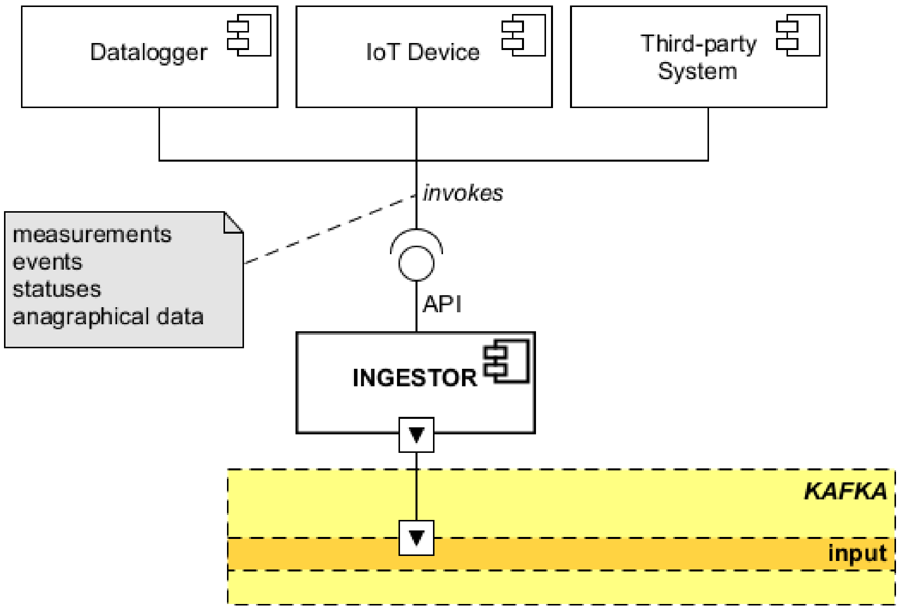


Figure 15) Ingestor artefact

III.3.1.c. Message Format

The Message Format is the format of all incoming data transmissions. This format is derived from the Kafka record format and therefore it is compound of:

- **Value**
  a sequence of bytes like those sent on a serial port, contained in a binary file or transmitted via network interaction. For example, the body of the request received by the Ingestor
- **Type**
  the type or format of the value field. It is a text string that should unambiguously indicate how to read the value.
- **Key**
  for each fixed type, the key is a text string identifier of the source of the data message. The couple (type, key) must be a universal identifier of the source of any data transmission.
- **Metadata**
  arbitrary textual information about a data message or about its source. Metadata are structured as a collection of name-value pairs with textual names and textual values. Names are repeatable. Examples of metadata are:
  - **encoding=UTF-8**: indicating the encoding of the value bytes
  - **requestId=6651a560-da17-4807-ab4b-ebc01895f1fd**: a unique id for the data message initialized by the ingestor.
  - **requestTs=2019-07-08T20:01:10.804+02:00**: timestamp at which the data entered the system, initialized by the ingestor.

III.3.1.d. Database

Currently, the Ingestor has no need of a database, thus it has no database.

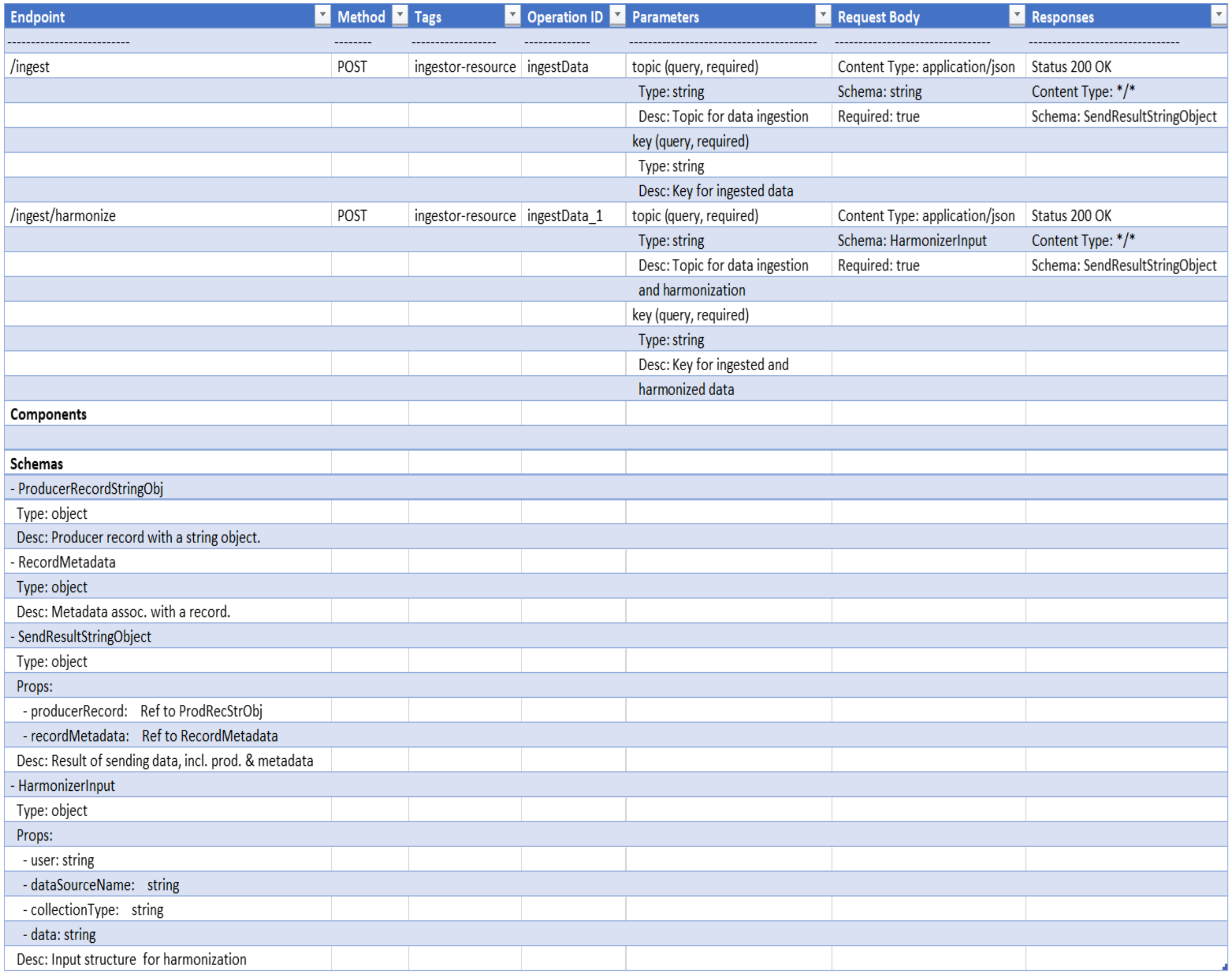
III.3.1.e. Configuration

The microservice configuration will be written in a YAML format file containing the main information as:

- Kafka URL as hostname and port (e.g., “127.0.0.1:9092”)
- The path and the port the API will be published on (e.g., “localhost:8888/api/ingestor”)
- Logging parameters as level and loggers (e.g., “level: INFO”)

However, for the ingestor a json file is mostly used.

For more details, please refer to the GitHub: https://github.com/biggproject/WP3-HttpIngestor



Endpoint	Method	Tags	Operation ID	Parameters	Request Body	Responses
-------------------------	--------	-----------------	-------------	-------------------------------------	------------------------------	------------------------------
/ingest	POST	ingestor-resource	ingestData	topic (query, required)	Content Type: application/json	Status 200 OK
				Type: string	Schema: string	Content Type: */*
				Desc: Topic for data ingestion	Required: true	Schema: SendResultStringObject
				key (query, required)		
				Type: string		
				Desc: Key for ingested data		
/ingest/harmonize	POST	ingestor-resource	ingestData_1	topic (query, required)	Content Type: application/json	Status 200 OK
				Type: string	Schema: HarmonizerInput	Content Type: */*
				Desc: Topic for data ingestion	Required: true	Schema: SendResultStringObject
				and harmonization		
				key (query, required)		
				Type: string		
				Desc: Key for ingested and		
				harmonized data		
Components						
Schemas						
- ProducerRecordStringObj						
Type: object						
Desc: Producer record with a string object.						
- RecordMetadata						
Type: object						
Desc: Metadata assoc. with a record.						
- SendResultStringObject						
Type: object						
Props:						
- producerRecord: Ref to ProdRecStrObj						
- recordMetadata: Ref to RecordMetadata						
Desc: Result of sending data, incl. prod. & metadata						
- HarmonizerInput						
Type: object						
Props:						
- user: string						
- dataSourceName: string						
- collectionType: string						
- data: string						
Desc: Input structure for harmonization						

Figure 16) BIGG ingestor open-api parameter details

III.3.1.f. Implementation

Two main classes will be implemented for this component:

DatalakeResource: The http end-point receiving http requests from the external world.

KafkaPusher: service able to dispatch messages via Kafka. The name of the topic is dynamic.

III.3.1.g. API

The Ingestor will offer an http API named “api/datalake”. Receiving external transmissions via http is the objective of this API. Three ways of http post calls are supported:

1. Form posts,
2. Form file posts,
3. Generic posts.

In the next three paragraphs we are going to see these three equivalent ways more in details. We assume that the bytes to transmit into the system are the following 54 ASCII characters:

```
{"ts":1553071755,"ms":"temperature","v":23.6,"u":"°C"}
```

We also assume that the message needs some metadata to be tracked together with the message itself and they are the following:

- IEEE EUI64 identifier: 70B3D54750100052
- location: 1st floor
- division: automotive

We will use the IEEE EUI64 identifier as message key, while location and division as metadata.

III.3.1.g.1. pi/datalake: Form posts

The Ingestor can accept incoming transmissions as form posts. This is the standard way html forms are submitted, with content-type: application/x-www-form-urlencoded. Here is an example of how to send data using the curl program:

```
curl --verbose \
     --request POST \
     --data location=1st-floor \
     --data division=automotive \
     --data-urlencode
payload='{"ts":1553071755,"ms":"temperature","v":23.6,"u":"°C"}' \
https://host.name/ingestor/api/datalake/test-type/70B3D54750100052
```

III.3.1.g.2. api/datalake: Form file posts

The Ingestor can accept incoming transmissions as multipart form post. This is the standard way used by html forms with files, with `content-type: multipart/form-data`. Supposing that the file /tmp/payload.json contains the JSON document to send, here is an example of how to send data using the curl program:

```
curl --verbose \
  --request POST \
  --form location=1st-floor \
  --form division=automotive \
  --form sn=70B3D54750100052 \
  --form dea=faab234875d86b99a1dd8f9afeffa4e7 \
  --form payload=@/tmp/payload.json \
  https://host.name/ingestor/api/datalake/test-type/70B3D54750100052
```

III.3.1.g.3. api/datalake: Generic posts

The Ingestor can accept incoming transmissions as an arbitrary http post. This is the fallback method when the content-type is not `application/x-www-form-urlencoded` neither multipart/form-data. In this case metadata must be encoded as http query string parameters in the URL. Here is an example of how to send data using the curl program:

```
curl --verbose \
  --request POST \
  --header 'Content-type: application/json' \
  --data '{"ts":1553071755,"ms":"temperature","v":23.6,"u":"°C"}' \
  'https://host.name /ingestor/api/datalake/test-
type/70B3D54750100052?location=1st-
floor&division=automotive&dea=faab234875d86b99a1dd8f9afeffa4e7'
```

III.3.1.g.4. *Inferred message key*

In general, the URL to post data to the Ingestor is

```
https://host.name/ingestor/api/datalake/type/key
```

Where host.name, type and key are variables. The key in the url is used as key of the message. The key in the url is optional for the Ingestor. On the other hand, it is not optional for a message. If the key is not in the URL, in other words, if data are posted to an URL like this:

```
https://host.name/ingestor/api/datalake/type
```

then the Ingestor tries to infer the key from the metadata. In fact, the micro-service looks for metadata with one of these names:

- sequence,
- key,
- sn,
- id,
- code,
- serial.

If such element metadata exists and it is single-valued, then the value is used as key for the message. Otherwise, a random key is generated.

III.3.1.h. GUI

The Ingestor currently does not need a GUI, thus it has none.

III.3.2. Harmonization components

III.3.2.a. Overview

The Digital Building Twin involves a multitude of data models and formats coming from different sources such as open data providers, assets management software and sensors. Regarding data formats, relational data bases and XML are still present, Open Data portals heavily rely on CSV, and web APIs on JSON. The RDF data model is used as a federated model to reach semantic interoperability and querying of data having heterogeneous formats.

The BIGG harmonizer aims at converting any data in scope of energy performance of building that fit with the BIGG ontology into RDF. In our approach we focus on JSON for data coming from sensors, CSV for data coming from Open Data and Asset Management Software and RDF for alignment with existing ontologies.

The harmonizer must generate data respectful of W3C recommendations and BIGG ontology by providing the workflow of functionalities as illustrated in Figure 17):

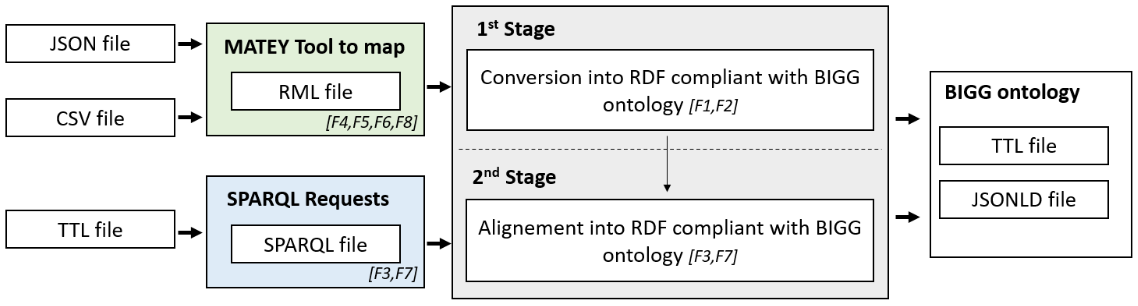


Figure 17) General workflow of the Harmonizer module

- F1: Converts JSON files into RDF compliant with BIGG ontology.
- F2: Converts CSV files into RDF compliant with BIGG ontology.
- F3: Aligns Data described using standard ontologies covering the same scope (IFCOWL, SAREF, SSN/SOSA, geoNAMES, QUDT, WGS84, FOAF)
- F4: Allows to map input objects with BIGG classes.
- F5: Allows to map input attributes with BIGG data properties.
- F6: Interprets implicit links between objects through object properties.
- F7: Reconciles input values with open registers, BIGG taxonomies and enumerations.
- F8: Materializes data context.

The following sections provide details about the required functionalities:

Converting input formats to RDF (F1, F2)

JSON (JavaScript Object Notation) is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute-value pairs and arrays. It is a common data format with diverse uses in electronic data interchange.

RDF (Resource Description Framework) is a World Wide Web Consortium (W3C) standard originally designed as a data model for metadata. It has come to be used as a general method for description and exchange of graph data. RDF provides a variety of syntax notations and data serialization formats, with Turtle (Terse RDF Triple Language) currently being the most widely used notation.

This function outputs data compliant to W3C recommendations and without loss of information. The compliance of harmonized data is ensured by functionalities F4 and F5. Interoperability of harmonized data is ensured by functionalities F3, F6 and F7. The quality of output data is ensured by functionalities F8 and F9.

Converting relational data to RDF on the fly

Ontology-Based Data Access (OBDA) is a popular approach for tackling the challenge of data integration and query answering over multiple data sources. OBDA provides a conceptual layer in the form of an ontology that defines a shared vocabulary, models the domain, hides the structure of the data sources, and enriches incomplete data with background knowledge. **In the BIGG project, the ontology for modeling the buildings/energy domains is the BIGG Standard Data Model 4 Building Ontology**. Users can then pose queries over this high-level conceptual view, and the OBDA system will translate them into queries over the underlying data sources.

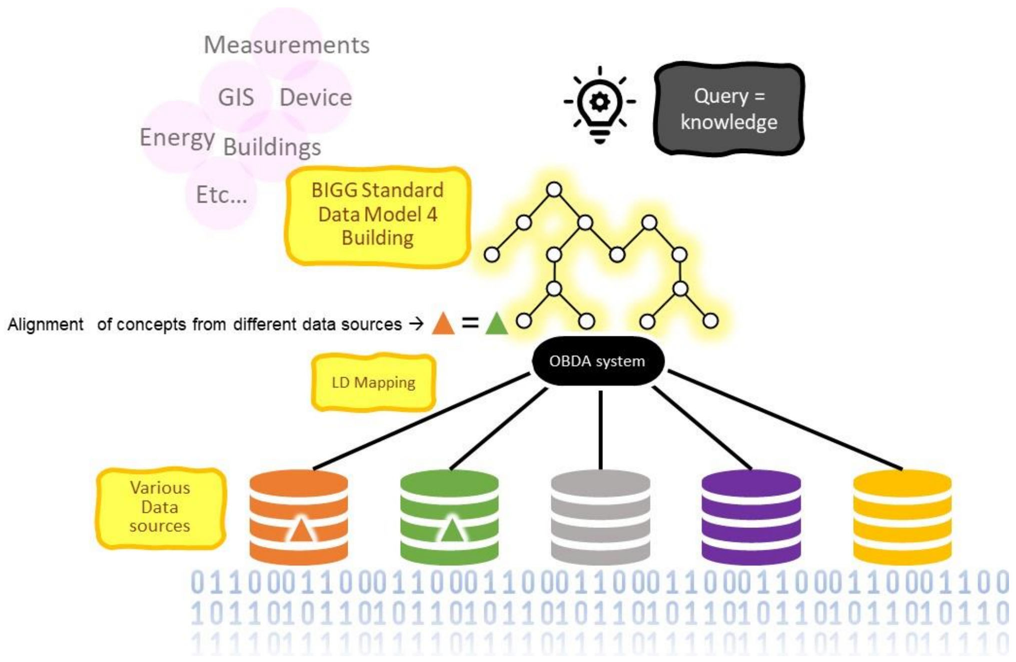


Figure 18) OBDA concept in the context of BIGG

One of the key benefits of OBDA is that it allows users to query data sources without needing to understand the structure of the data sources, the relation between them, or the encoding of the data. If an OBDA system uses the BIGG ontology to map the BIGG high-level concepts in the user query to the corresponding data in the underlying data sources, it can be used to integrate data from a variety of different sources, including relational databases, NoSQL databases, and XML documents. Thus, aligned with the BIGG project objectives, OBDA systems can use the BIGG ontology to provide a unified view of the data, regardless of the underlying data sources.

R2RML, which stands for "RDB to RDF Mapping Language," is a specification and language used for mapping relational data (typically stored in a relational database) to RDF (Resource Description Framework) data, which is commonly used in the context of semantic web and linked data applications. The principle of R2RML is to define a standardized way to create mappings between relational database schemas and RDF data, making it easier to integrate. R2RML provides a standardized and flexible way to map relational data to RDF, making it easier to integrate structured data from relational databases into the semantic web and linked data environments. These mappings facilitate interoperability and data integration across different data sources and applications.

In the context of the BIGG project where many relational databases can be used as data sources, it is interesting to demonstrate an OBDA as a powerful tool for integrating and querying data from multiple sources because it is particularly well-suited for applications that require complex queries over large and heterogeneous datasets. For that purpose, a demonstration of the Ontop[21] OBDA system has been added in the BIGG repository to be shared with the BIGG developers' community. This is a powerful tool to dynamically query various databases and generate the BIGG data model directly at the harmonizer level. Ontop is one of the exiting R2RML implementations, it is an open-source ontology-based mature

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

framework[22] designed for querying relational databases using ontologies. It offers several advantages, particularly for those working with semantic web technologies and linked data. Here are some of the advantages of the Ontop framework:

- Integration of Relational Databases with Semantic Web: Ontop allows you to bridge the gap between relational databases and semantic web technologies. You can use it to create a virtual RDF graph representation of your relational database, making it easier to work with structured data in the context of the semantic web.
- Ontology-Based Querying: With Ontop, you can use ontologies to define the meaning of your data. This makes it possible to perform ontology-based querying over your relational data. Queries can be expressed in a query language designed for querying RDF data.
- SPARQL Compatibility: Ontop is compatible with SPARQL, which is a widely used query language for querying RDF data. This means that users familiar with SPARQL can use Ontop to query relational databases without having to learn a new query language.
- Virtual RDF Graph: Ontop creates a virtual RDF graph that reflects the structure and semantics of your relational data. This makes it easy to navigate and query your data as if it were stored in RDF format.
- Reasoning and Inference: Ontop supports ontology reasoning and inference, which means that it can infer additional facts and relationships based on the ontology. This can help uncover implicit knowledge in your data.
- Query Optimization: Ontop includes query optimization techniques to improve the efficiency of query processing, which is crucial for querying large and complex databases.
- Mapping Configuration: Ontop uses mapping configurations to define how data from the relational database should be mapped to the RDF model. This configuration allows for flexibility and customization in how data is represented in the virtual RDF graph.
- Industry Adoption: Ontop has gained recognition and adoption in the semantic web and linked data communities, and it has been used in various research projects and applications.
- Open Source: Ontop is open source, which means that it is freely available for use and can be customized or extended to meet specific project requirements.

Overall, Ontop provides a powerful tool for integrating relational databases into the semantic web and for leveraging the benefits of ontologies in data querying and reasoning. It is particularly useful when you want to combine structured data from databases with the flexibility and expressiveness of semantic web technologies.

Ontop has been installed and configured to ingest a relational database containing descriptions of buildings, sensors and measurements. The results show that Ontop allow to directly query the relational database through a SPARQL query. On one way the SPARQL query is translated on-the-fly into a SQL query and on the way back, the SQL results is converted into RDF results. The main requirement to settle this process is to build a mapping file -based on R2RML standard- that describe correspondences between relational entities and graph entities. This mapping file is coherent with RML mapping files used to convert JSON to RDF as part of RML overall standard.

[22] http://www.semantic-web-journal.net/content/ontop-answering-sparql-queries-over-relational-databases-1 / https://research.bcgl.fr/pdfs/ontop-iswc20.pdf

Aligning data (F3)

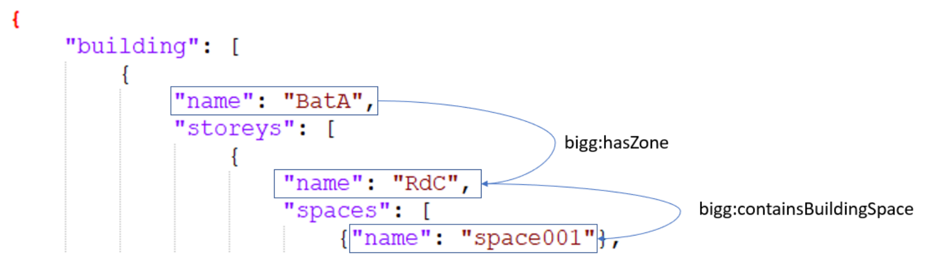
The BIGG ontology has been aligned with few existing ontologies related to Digital Building Twin.

BOT[23] ontology is a minimal ontology for defining relationships between the sub-components of a building. It was suggested as an extensible baseline for use along with more domain specific ontologies following general W3C principles of encouraging reuse and keeping the schema no more complex than necessary. BIGG defines some classes that are equivalent (according to OWL) to BOT classes.

SAREF[24] is an ontology supported by ETSI SmartM2M standard to achieve interoperability among IoT projects that can be extended to any IoT vertical domains such as smart buildings or energy. BIGG ontology reuse few SAREF concepts such as saref:Device, saref:Sensor and saref:UnitOfMeasure to describe sensors networks and time series.

Interpreting implicit link (F6)

Underlying structures of JSON are arrays and trees. Relations between objects are implicitly declared by using the native parent-child relation provided by the tree structure. As BIGG ontology defines several relations the harmonized need to match the parent-child relation between two JSON objects with one of the BIGG ontology relation. The next schema illustrates how two parent-child relation from the same JSON file can be interpreted as distinct relations in the BIGG ontology.



```
{
    "building": [
        {
            "name": "BatA",
            "storeys": [
                {
                    "name": "RdC",
                    "spaces": [
                        {"name": "space001"},
```

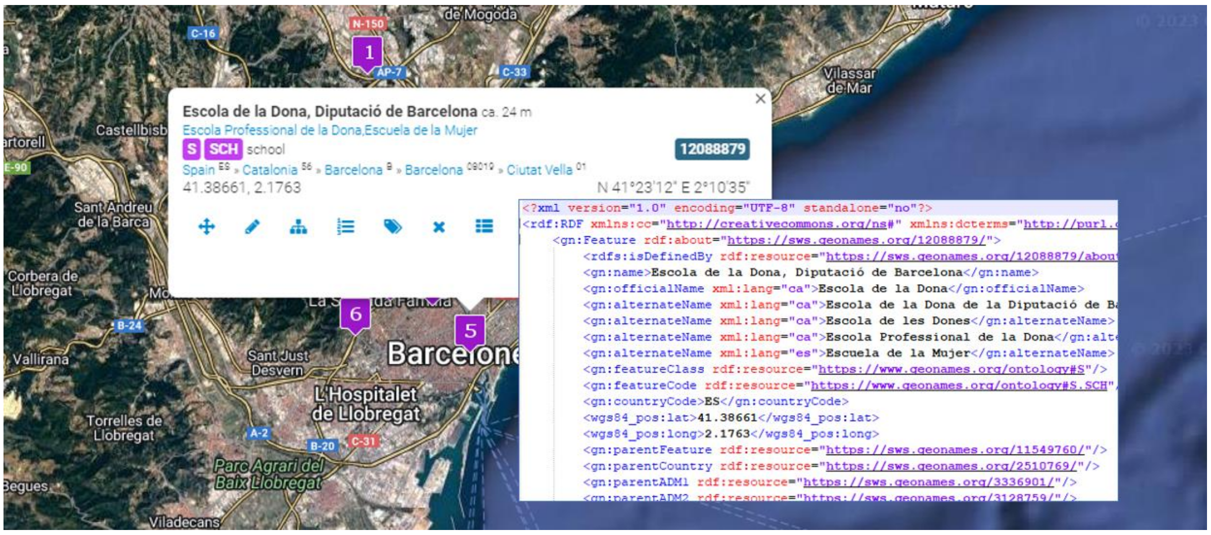
Figure 19) Sample of JSON file describing building structure.

Reconciliating data with enumerations (F7)

In this project geonames is used as a universal Building register. Geonames for building. Geonames building are reconciliated with BIGG building description by using the building location. This process allows to federate data provided by the building owner to data provided by the Open Data community. Figure 20) shows an example of data reconciliation.

[23] Mads Holten Rasmussen, Maxime Lefrancois , Georg Ferdinand Schneider , Pieter Pauwels. BOT: the Building Topology Ontology of the W3C Linked Building Data Group.. Semantic Web – Interoperability, Usability, Applicability, 2021, 12 (1), pp.143-161. ⟨10.3233/SW-200385⟩

[24] http://www.etsi.org/deliver/etsi_ts/103200_103299/103264/01.01.01_60/ts_103264v010101p.pdf

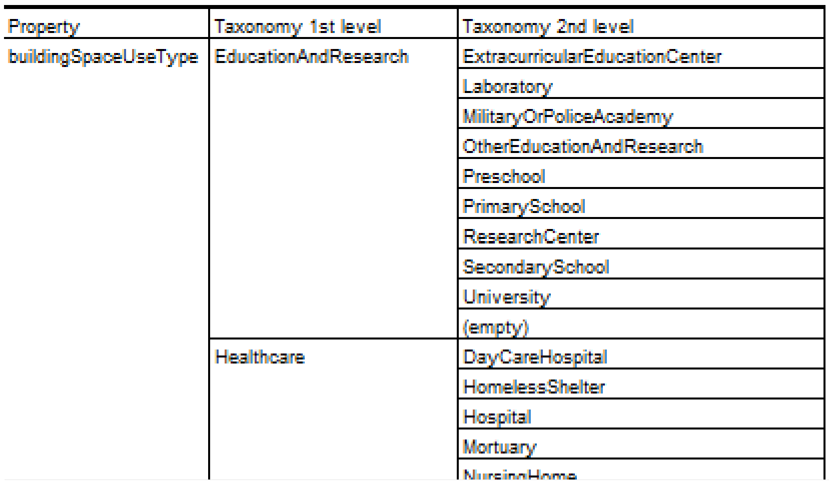


Escola de la Dona, Diputació de Barcelona ca. 24 m
Escola Professional de la Dona,Escuela de la Mujer
S SCH school 12088879
Spain ES » Catalonia 56 » Barcelona B » Barcelona 08019 » Ciutat Vella 01
41.38661, 2.1763 N 41°23'12" E 2°10'35"

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<rdf:RDF xmlns:cc="http://creativecommons.org/ns#" xmlns:dcterms="http://purl.c
    <gn:Feature rdf:about="https://sws.geonames.org/12088879/">
        <rdfs:isDefinedBy rdf:resource="https://sws.geonames.org/12088879/about
        <gn:name>Escola de la Dona, Diputació de Barcelona</gn:name>
        <gn:officialName xml:lang="ca">Escola de la Dona</gn:officialName>
        <gn:alternateName xml:lang="ca">Escola de la Dona de la Diputació de Ba
        <gn:alternateName xml:lang="ca">Escola de les Dones</gn:alternateName>
        <gn:alternateName xml:lang="ca">Escola Professional de la Dona</gn:alte
        <gn:alternateName xml:lang="es">Escuela de la Mujer</gn:alternateName>
        <gn:featureClass rdf:resource="https://www.geonames.org/ontology#S"/>
        <gn:featureCode rdf:resource="https://www.geonames.org/ontology#S.SCH"/
        <gn:countryCode>ES</gn:countryCode>
        <wgs84_pos:lat>41.38661</wgs84_pos:lat>
        <wgs84_pos:long>2.1763</wgs84_pos:long>
        <gn:parentFeature rdf:resource="https://sws.geonames.org/11549760/"/>
        <gn:parentCountry rdf:resource="https://sws.geonames.org/2510769/"/>
        <gn:parentADM1 rdf:resource="https://sws.geonames.org/3336901/"/>
        <gn:parentADM2 rdf:resource="https://sws.geonames.org/3128759/"/>
```

Figure 20) Geonames provides universal IDs and descriptions for public buildings.

The BIGG ontology provides taxonomies to classify objects such as buildings, spaces or sensors. Figure 21) gives an excerpt of the Building usages taxonomy that description the main service provided by the building.



Property	Taxonomy 1st level	Taxonomy 2nd level
buildingSpaceUseType	EducationAndResearch	ExtracurricularEducationCenter
		Laboratory
		MilitaryOrPoliceAcademy
		OtherEducationAndResearch
		Preschool
		PrimarySchool
		ResearchCenter
		SecondarySchool
		University
		(empty)
	Healthcare	DayCareHospital
		HomelessShelter
		Hospital
		Mortuary
		NursingHome

Figure 21) Sample of one of the BIGG taxonomies classifying building space usages

When building usage classification used by raw data differs from the BIGG classification, the harmonizer needs to translate from one to the other. For instance, the French building classification entry “Université” should be converted into its corresponding entry in the BIGG taxonomy “University”.

Expliciting data context (F8)

When describing energy consumption as JSON time series, the context (Who, Where, How) is often skipped or implicitly indicated in the file name. The harmonized data need to retrieve the context (space, building, feature of interest) from the existing database or from other data sources.

```
{
    "_id" : "60f7fcd6e594e218826ca9f2",
    "datetime" : "2021-07-20T21:00:00.000Z",
    "cups" : "ES0031406110149001EJ0F",
    "consumptionKWh" : 12.0,
    "obtainMethod" : "Real"
}
```

Figure 22) Example of JSON time series with no context

Generating 5-star data

The Digital Building Twin requires 5-star data as introduced by Tim Berners-Lee[25]. Most of the time, raw data collected by the Digital Building Twin must be cleaned and enriched to by fully interoperable. The five levels defined by Tim Berners-Lee are: available (level 1), structured (level 2), open (level 3), universally identified (level 4) and linked to other data (level 5).

Level 1 is a requirement. Level 2 is ensured by RDF specification and functionality F1. The harmonizer ensures level 4 by generating universal by converting local identifier to universal identifier by using data provider identification and context. Finally, level 5 is ensured by functionality F6.

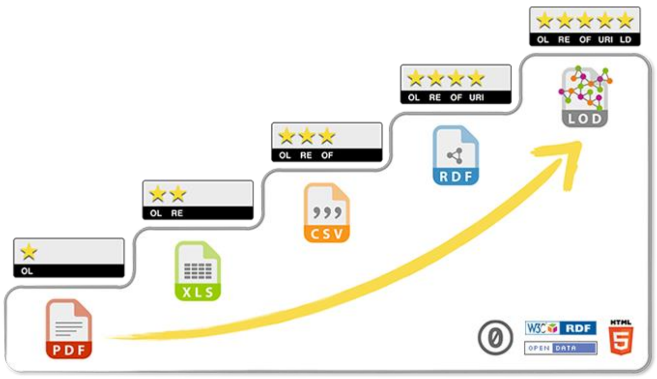


Figure 23) Five Star Scheme suggested by Tim Berners-Lee

The harmonizer mapping file

The harmonizer component requires a mapping file to convert custom data into the BIGG harmonised model. RML[26] has been chosen as the harmonizer's mapping language for two main reasons: (1) RML is based on RDF, which is consistent with the harmonized data, (2) RML is an extension of a W3C specification, R2RML[27].

The RML mapping file contains rules to be used to transform an input data into an RDF format.

RML is one of several solutions that have been proposed to map semi-structured data to RDF. It is an extensible language that can handle a variety of source formats, including JSON, CSV, and XML. It is also scalable and can be used to map large and complex datasets. Other mapping languages that were considered for the BIGG project include SPARQL-GENERATE

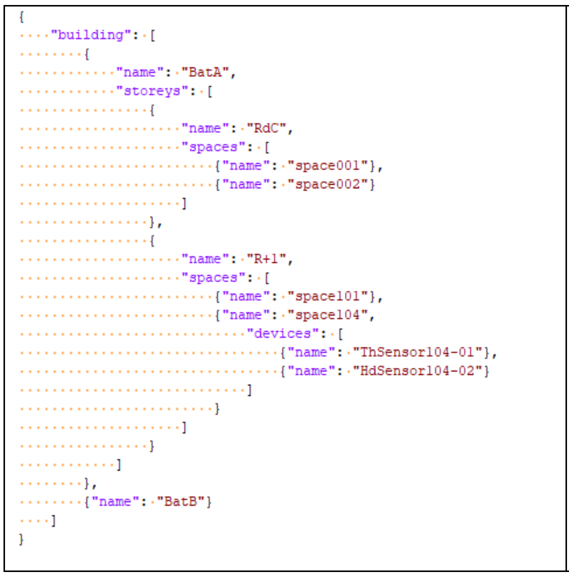
[25] https://5stardata.info/en/

[26] Anastasia Dimou, Miel Vander Sande, Pieter Colpaert, Ruben Verborgh, Erik Mannens, and Rik Van de Walle. RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In Proceedings of the Workshop on Linked Data on the Web, co-located with the 23rd International World Wide Web Conference (WWW 2014), Seoul, Korea, 2014.

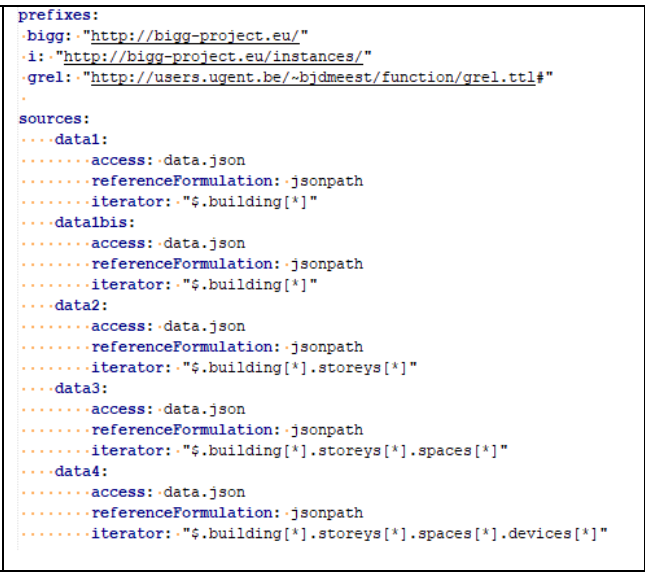
[27] https://www.w3.org/TR/r2rml/

and OpenRefine. However, SPARQL-GENERATE is not as extensible as RML, and OpenRefine does not provide access to the CSV-to-RDF mapping mechanism that is enacted internally.

YARRRML[28] is a human-readable mapping language developed by Ghent University, which allows to define rules and to convert it into RML or R2ML mapping language. In the context of BIGG, it has been used to ease the process of writing RML mapping files.There is a web-based version named MATEY,[29] it offers the possibility to write YARRML rules to transform a specified JSON file into an RDF document, and to generate the corresponding RML rules. In the case of the BIGG harmonizer, MATEY is used to generate RML rules, exported into a document, to map them with the dataset of each business case. Then, the RML mapping can be processed to generate the RDF conversion of the dataset according to the BIGG ontology. Figure 24) lists an example of a JSON input file with the corresponding RML mapping.



```
{
    "building": [
        {
            "name": "BatA",
            "storeys": [
                {
                    "name": "RdC",
                    "spaces": [
                        {"name": "space001"},
                        {"name": "space002"}
                    ]
                },
                {
                    "name": "R+1",
                    "spaces": [
                        {"name": "space101"},
                        {"name": "space104",
                            "devices": [
                                {"name": "ThSensor104-01"},
                                {"name": "HdSensor104-02"}
                            ]
                        }
                    ]
                }
            ]
        },
        {"name": "BatB"}
    ]
}
```



```
prefixes:
 bigg: "http://bigg-project.eu/"
 i: "http://bigg-project.eu/instances/"
 grel: "http://users.ugent.be/~bjdmeest/function/grel.ttl#"
 
sources:
    data1:
        access: data.json
        referenceFormulation: jsonpath
        iterator: "$.building[*]"
    data1bis:
        access: data.json
        referenceFormulation: jsonpath
        iterator: "$.building[*]"
    data2:
        access: data.json
        referenceFormulation: jsonpath
        iterator: "$.building[*].storeys[*]"
    data3:
        access: data.json
        referenceFormulation: jsonpath
        iterator: "$.building[*].storeys[*].spaces[*]"
    data4:
        access: data.json
        referenceFormulation: jsonpath
        iterator: "$.building[*].storeys[*].spaces[*].devices[*]"
```

Figure 24) Example of a JSON input file with the corresponding RML mapping

[28] https://rml.io/yarrrml/spec/https://rml.io/yarrrml/spec/

[29] https://rml.io/yarrrml/matey/

III.3.2.b. Core implementation

The BIGG Harmonizer components code and examples are available here: https://github.com/biggproject/Harmonizer. An Ontop OBDA system demonstration can be found here : https://github.com/biggproject/Harmonizer/tree/main/ontop.

The java library 'rml.jar' is used to process RML mapping rules on a selected dataset to generate an RDF document. The integration of this library into a Python module allows to experiment the harmonization of input data from multiple sources automatically.

The Python module created to harmonize input data with the BIGG ontology is composed of two stages, the first one is related to the RML mapping file, and the second one is related to the alignment of standards ontologies with the BIGG ontology.

The conversion step is the first stage of the Python module, it corresponds to the use of the java library to convert an input JSON file into an RDF file thanks to the mapping rules defined in the RML file. The module can output two serialization formats of RDF, the TTL (Turtle) or the JSON-LD[30] format.

The second stage of the Python module, corresponds to the use of SPARQL queries to add, translate or complete the RML stage. The second stage can also be used to align data based on standardized ontology into the BIGG compliant RDF. For instance, it allows to align data based on standards ontologies like IFCOWL, SOSA or SAREF, with the BIGG ontology.

The execution and test of the Python module can be done in a Jupiter Notebook, with the following command line:

For production integration, the Harmonizer can be operated as a microservice. As an input the Harmonizer receives as a configuration a Standard Mapping Template and produces a transformation of the original data model format into a RDF structure, which is then aligned to the BIGG Data Model. The Harmonizer will enable the translation of the incoming messages from the mapped data sources emitted from the Ingestors into the BIGG common message format.

The Harmonizer will operate in the system as a microservice based on the "Processor" pattern. It will be responsible for translating all incoming messages to a common message format so that any other component of the system can benefit of any data message in a fast and standardized way. The Harmonizer must be:

1. **Responsive**: minimizing the latency from the incoming message to the time the message has being stored and handled by the system.
2. **Maintainable and extensible**: allowing to respond quickly to the market that asks for new formats of data message to be supported by the system.
3. **Efficient**: engaging a minimal amount of computational resources.
4. **Elastic and scalable**: allowing more instances of Processor to run in the system on different nodes to increase (possibly linearly) the throughput; making it easy to add, remove or move instances in the system.
5. **Robust**: coping with errors and with erroneous inputs without compromise the operation of the system.

Figure 25) depicts the overview of a workflow involving the Harmonizer V2 component.

[30] https://json-ld.org/

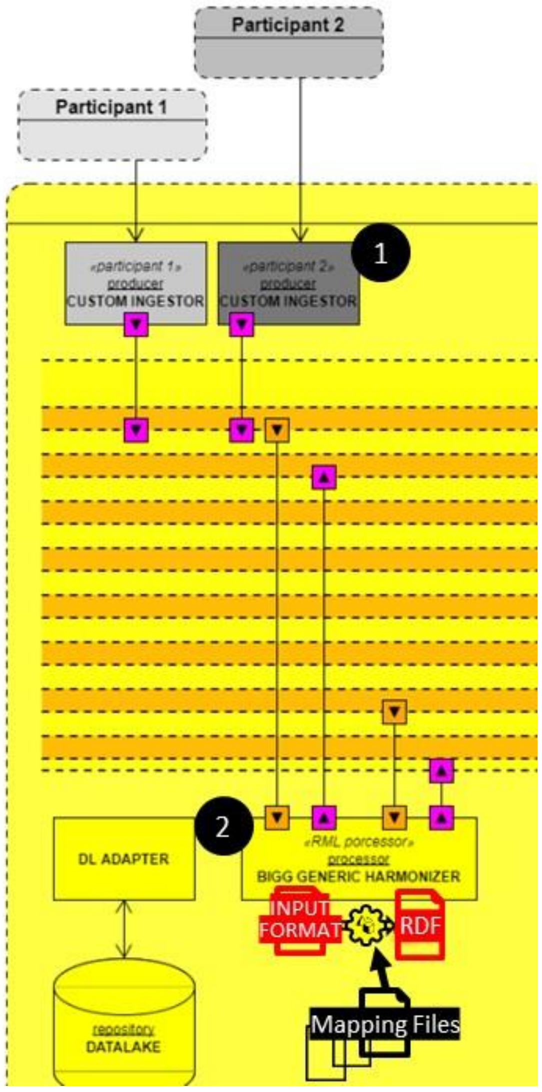

Participant 2
Participant 1
1
«participant 1» producer CUSTOM INGESTOR
«participant 2» producer CUSTOM INGESTOR
2
DL ADAPTER
«RML porcessor» processor BIGG GENERIC HARMONIZER
INPUT FORMAT
RDF
Mapping Files
repository DATALAKE

Figure 25) V2 version of the architecture of the ingestion/harmonization process

1. Custom ingestors will get connected to external participant's sources to feed participant's raw data into the BIGG system.
2. The Harmonizer is triggered on a raw data ingestion event. It uses the mapping files defined in WP4 in a generic process of converting input raw data structures into RDF harmonized information that can be used in later processes of an implemented BIGG components pipeline.

III.3.2.c. Architecture

The Harmonizer as sketched in Figure 26) is a micro-service cooperating with other components of the BIGG RAF:

- **Ingestor**: those components are responsible for introduce the data messages into the system, without taking care of the meaning and the format of the data. They, in a loosely-coupled way (based on Kafka topics), are responsible to feed the Harmonizer with data message of arbitrary format.
- **Adapter and other adapters**: other components that needs to receive data and they need the data in a common format. The Harmonizer delivers data to them, in a loosely-coupled way, based on Kafka topics.

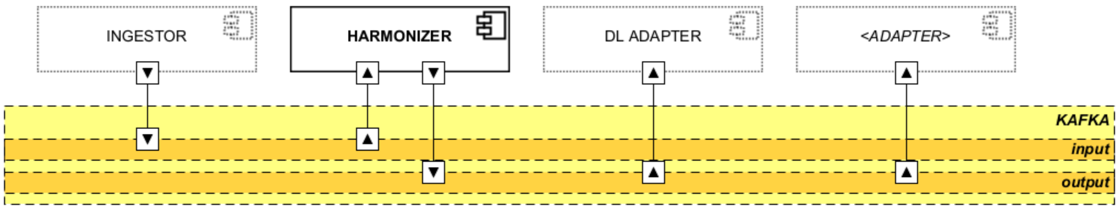


Figure 26) Harmonizer artefact

III.3.2.d. Harmonized Message Format

The Harmonized Message Format is the format of all incoming data message. It is derived from the Kafka record format and therefore it is compound of:

- **Value**
  The JSON serialization of the resulting harmonized message
- **Type**
  The type or format of the value field. It is a text string that must be enough to indicate how to read (de-serialize and de-harmonize) the value.
- **Key**
  For each fixed type, the key is a text string identifier of the source of the data message. The couple (type, key) must be a universal identifier of the source of the data message.
- **Metadata**
  Arbitrary textual information about a data message or about its source. Metadata is structured as a collection of name-value pairs with textual names and textual values. Names are repeatable. Examples of metadata are:
    - **encoding=UTF-8**: indicating the encoding of the value bytes
    - **requestId=6651a560-da17-4807-ab4b-ebc01895f1fd**: a unique id for the data message initialized by the ingestor.
    - **requestTs=2019-07-08T20:01:10.804+02:00**: timestamp at which the data entered the system, initialized by the ingestor.

III.3.2.e. Database

At present day, the Harmonizer has no need of a database, thus it has no database.

III.3.2.f. Configuration

The Harmonizer must both consume and produce Kafka messages, so the YAML configuration file must have consumer and producer configuration keys:

- consumer.bootstrap.servers (Kafka instance e.g., “localhost:9092”)
- consumer.topics (topics list or pattern e.g., “input-*”)
- producer.bootstrap.servers (Kafka instance e.g., “localhost:9092”)
- producer.topic (topic to produce to, e.g., “harmonized”)
- server path and port (e.s. “localhost:8093/api/processor”)
- logging (es. “Level: INFO”)

III.3.2.g. Implementation

The implementation of the Harmonizer is based on a simple interface named the “mapper”:

```
public interface Mapper<C,S> {
     public S mapFrom(C c);
     public C mapTo(S s);
}
```

All its implementations will be used to map from the original format to the harmonized format and vice versa. The following class diagram depicts the main components of the microservice:

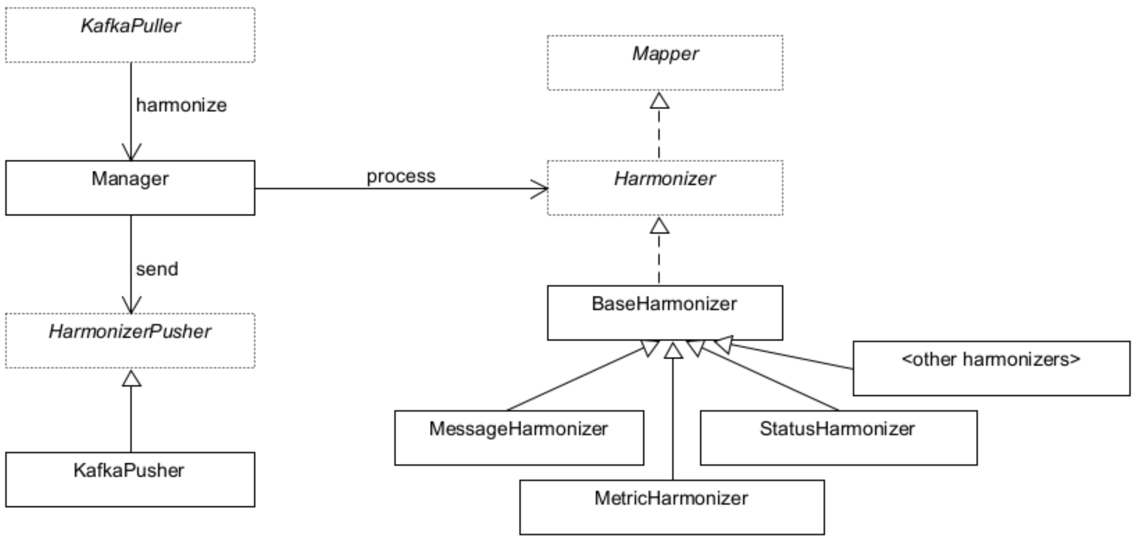


Figure 27) Main components of a BIGG Harmonizer

- **KafkaPuller**
  this class will consume input messages in original format and will deliver its content to the Manager
- **Manager**
  the manager will process the message sending its content to the right “Harmonizer” implementation (i.e., MessageHarmonizer, MetricHarmonizer, etc.) and, once received the corresponding harmonized message, propagating it to the “pusher”
- **Harmonizer**
  the “Harmonizer” interface will be an extension of the “mapper” interface described above. Every partner will develop the set of Harmonizers that will accomplish their respective requirements handling their proprietary formats
- **KafkaPusher**
  this class will send the harmonized messages to the proper Kafka topic to let the “adapters” to retrieve the content

III.3.2.h. GUI

The Harmonizer does not need a GUI, thus it has none.

III.3.3. AI toolbox : Data processing/analysis components

III.3.3.a. Overview

Forecasting and predictions form the bedrock of strategic planning and decision-making across a wide spectrum of disciplines. The emergence of Machine Learning (ML) and Artificial Intelligence (AI) has introduced a change in basic assumptions in how we approach forecasting, prompting a departure from conventional techniques. ML and AI tools offer several advantages over conventional forecasting methods, making them increasingly indispensable for solving complex forecasting problems in today's world.

ML and AI tools can:

- Uncover intricate patterns within vast and complex datasets, including non-linear relationships that might be overlooked by conventional techniques.
- Adapt to changing circumstances and learn from new data, enabling real-time decision-making based on up-to-date insights.
- Automate the forecasting process, reducing the need for manual intervention and enhancing efficiency.
- Conduct multivariate analysis, effectively considering a multitude of variables that might interact in complex ways.

The transition to ML and AI-driven forecasting is a necessity dictated by the demands of the modern world, characterized by unprecedented complexities, data volumes, and the rapid pace of change. By harnessing the power of ML and AI, we can unlock deeper insights, generate more accurate predictions, and adopt a proactive approach to decision-making.

ML and AI tools provided by BIGG AITB are essential for forecasting in today's world because the world is more complex and data-driven than ever before.

AITB is a Python and R library that can be used to analyze building data and construct AI models to optimize energy conservation. It is adaptable to different data formats and structures, and can handle a wide spectrum of data types, including real-time measurements, pulses, and index values. The AITB is designed to be used in conjunction with the BIGG data model,[31] but it can also be used independently.

The AITB is made up of function blocks, which are engineered to be versatile and applicable to a spectrum of building-related scenarios. These function blocks are categorized into four primary modules:

1. Data preparation modules: These modules synchronize with initial data management stages, such as quality assessments, outlier identification, and timestamp management.
2. Data transformation modules: These modules delve into data categorization and secondary dataset management, involving elements like calendar and weather data.
3. Modeling modules: These modules focus on constructing, evaluating, and testing data models.
4. Reinforcement learning modules: These modules pertain to the creation and training of reinforcement learning agents—a specialized facet of machine learning.

The AITB is a powerful and flexible tool that can be used to swiftly and accurately predict energy usage, pinpoint areas for energy savings, and optimize energy performance in buildings. It is a significant contribution towards advancing energy efficiency and sustainability in the building sector.

III.3.3.b. Core implementation

This section presents the data processing and analysis components created in the BIGG project. These components are presented here but are more deeply detailed and documented on biggproject/biggdocs GitHub.[32]

The BIGG GitHub has been separated in three separate sections:

1. BiggDocs: Where all the functions are defined and described in detail.
2. BiggPy:[33] Where the functions are implemented in Python language.
3. BiggR:[34] Where the functions are implemented in R.

The creation of these BIGG data processing and analysis components is a critical part of the BIGG project. Defining and implementing these components is an iterative process that will span throughout the duration of the BIGG project.

[31] https://github.com/biggproject/Ontology

[32] https://github.com/biggproject/biggdocs

[33] https://github.com/biggproject/biggpy

[34] https://github.com/biggproject/biggr

The AI Toolbox for Buildings (AITB) serves as a comprehensive resource for analyzing building data and constructing AI models to optimize energy conservation within buildings. This toolbox leverages the capabilities of Python and R programming languages, customized with popular libraries like scikit-learn and Caret to meet specific usability requirements.

At its core, the AITB is designed for adaptability, capable of accommodating diverse data formats and seamlessly navigating through harmonized data in alignment with the BIGG data model. While it does not perform data collection, it proficiently manages various data types, including real-time measurements, pulses, and index values — common elements in building energy data. Importantly, each function block within the toolbox possesses standalone functionality, independent of the BIGG data model, making it versatile and compatible with various data sources.

Furthermore, while the toolbox's pipelines are optimized for harmonized inputs and outputs, the foundational building blocks of these pipelines do not inherently demand such coherence. This adaptability to different data formats positions the toolbox as a potent instrument for effective building data management, regardless of the specific presentation.

III.3.3.c. AITB Function Blocks and Modules[35]

The function blocks within AITB are engineered with versatility in mind, offering applicability across a spectrum of building-related scenarios. These function blocks are defined by their inputs, functions, and outputs, forming the cornerstone of distinct modules. These modules are thoughtfully categorized into module blocks, encompassing four primary classifications:

Data Preparation Modules: These function blocks synchronize with initial data management stages, including quality assessments, outlier identification, and timestamp management :

[35] biggpy/ai_toolbox at main · biggproject/biggpy · GitHub

Table 2: Data preparation Modules, their decriptions, their availability & usage

Function Name	Description	Used in Applications (A1 to A5)	Availability
Time Stamps Alignment			
detect_time_step	Infers the minimum time step from input data.	A1, A2, A3, A4	biggr, biggpy
align_time_grid	Aligns input time series frequency with specified aggregation.	A1, A2, A3, A4	biggr, biggpy
clean_ts_integrate	Converts cumulative or onChange measurements to instantaneous.	A1, A2	biggr, biggpy
Outlier Detection			
detect_ts_min_max_outliers	Detects outliers outside allowed range in time series data.	A1, A2, A5	biggr, biggpy
detect_ts_zscore_outliers	Detects outliers based on Z-score threshold in time series.	A1, A2, A3, A4	biggr, biggpy
detect_ts_calendar_model_outliers	Detects outliers using a calendar-based model.	A1, A2	biggr
detect_static_min_max_outliers	Detects outliers in static data (e.g., building areas).	A1, A2	biggr
detect_static_reg_exp	Detects elements satisfying specified regular expressions.	A1, A2	biggr
detect_disruptive_period	Detects disruptive periods in consumption time series.	A1, A2	biggr
detect_holidays_in_tertiary_buildings	Detects holiday periods in highly seasonal buildings.	A1, A2	biggr
Missing Data Management			
fill_ts_na	Imputes values to NA elements in a time series.	A1, A2	biggr, biggpy

Data Transformation Modules: These function blocks delve into data categorization and secondary dataset management, involving elements like calendar and weather data.

Modeling Modules: The focus of these function blocks revolves around constructing, evaluating, and testing data models.

Reinforcement Learning Modules: Function blocks in this category pertain to the creation and training of reinforcement learning agents—a specialized facet of machine learning.

Table 3: Functional blocks and descriptions of data transformation modules of AITB

Function Name	Description	Used in Applications (A1 to A5)	Availability
Profiling			
clustering_dlc	Clusters similar daily load curves based on load curves, calendar variables, and outdoor temperature using spectral clustering. Minimum frequency: hourly.	A1, A2	biggr
classification_dlc	Classifies daily load curves based on clustering or labeled datasets and new data. Minimum frequency: hourly.	A1, A2	biggr
weekly_profile_detection	Returns the weekly profile of the input time series.	A5	biggpy
yearly_profile_detection	Returns the yearly profile of the input time series.		biggpy
add_weekly_profile	Derives the weekly profile of an input time series (hourly or higher frequency) and adds it to the feature set. Can enhance linear model performance.	A3 (transformer version)	biggpy
generate_extended_weekly_profile	Derives the weekly profile of an input time series at hourly frequency, repeating it for multiple years for prediction.	A3	biggpy
Holidays			
add_holiday_component	Adds a public holiday feature based on specified country, province, and state to the input feature set within the time range of the input time series.	A3, A4 (transformer version)	biggpy
Calendar			
add_calendar_components	Decomposes time into various features (e.g., date, day of the year, day of the week)	A1, A2, A3, A4, A5, A6	biggr, biggpy

	considering local time zone. Typically used for modeling user behavior seasonality.		
trigonometric_encode_calendar_components	Encodes calendar components into sine and cosine trigonometric cyclic components. Boosts predictive capabilities of some models.	A3, A4	biggpy
Weather			
degree_days	Calculates degree-days with desired output frequency, considering cooling or heating mode.	A1, A2, A3, A4	biggr, biggpy
degree_raw	Calculates the difference between outdoor temperature and a base temperature, irrespective of the original data frequency.	A1, A2	biggr
get_change_point_temperature	Finds optimal change point temperature based on correlation between energy consumption and outdoor temperature data.		biggr
Autoregressive Processes			
lag_components	Shifts a set of features in time for multi-step predictions in autoregressive models.	A1, A2	biggr
lpf_ts	Computes a first-order low-pass filter for smoothing time series data, used for various purposes including simplifying modeling.	A1, A2	biggr
get_lpf_smoothing_time_scale	Calculates the smoothing time scale parameter of the first-order low-pass filter over an input variable, considering a specific time constant in hours.	A1, A2	biggr
Fourier Series			

fs_components	Obtains components of the Fourier Series in sine-cosine form to linearize seasonal input time series to some output (e.g., energy consumption).	A1, A2	biggr
Modeling			
Model Candidates			
RLS	Custom model wrapper for the R-package caret to train linear models using Recursive Least Squares method. Time-varying coefficients for improved data fitting.	A1, A2	biggr
GLM	Custom model wrapper for the R-package caret to train Generalized Linear Models.	A1, A2	biggr
Cross Validation			
BlockingTimeSeriesSplit	Special time series partitioning for cross-validation, generating disjoint partitions in each iteration.	A3	biggpy
Model Assessment			
evaluate_model_cv_with_tuning	Nested cross-validation with hyperparameter tuning to reduce bias in model selection and generalization error estimation.	A3, A4	biggpy
Model Identification			
identify_best_model	Generalized pipeline for supervised learning to find the best model among different families with specific parameter grids, based on an input time series and scoring function.	A3, A4, A5	biggpy
Model Persistence and Prediction			
serialize_model	Serializes and saves a model instance to a specified file path and format.	A3, A4	biggpy

deserialize_and_predict	Deserializes a model, applies it to input data, and returns predicted values as a time series.	A1, A2, A3, A4	biggpy

III.3.3.d. BIGG sample applications

The AITB has been developed to address specific use cases within the BIGG project, each of which is discussed next.

- **Energy benchmarking of buildings (GitHub)[36]**

This application focuses on the benchmarking and monitoring of energy consumption in buildings. It achieves two primary goals:

First, it evaluates the energy usage of a single building by analyzing its historical consumption and weather data, a process known as longitudinal benchmarking. This involves breaking down total consumption into three components: baseload, heating, and cooling. These components, combined with static building information, lead to the estimation of various Key Performance Indicators (KPIs). These KPIs provide insights into usage patterns, costs, savings, and emissions associated with energy consumption over time.

Second, the application compares the KPIs generated from the longitudinal benchmarking process with those of similar buildings in terms of characteristics and weather conditions. This comparison, referred to as cross-sectional benchmarking in literature, helps identify how a building's energy consumption performance compares to its peers.

Thus, the application can provide valuable insights for optimizing energy usage, reducing costs, and minimizing environmental impact. This contributes to more informed decision-making and improved energy efficiency in buildings.

- **Energy Efficiency Measures (EEM) assessment (GitHub)[37]**

This pilot evaluates the energy, cost, and emissions savings resulting from the implementation of specific EEMs. This assessment hinges on data-driven modeling of a building's energy consumption, comparing consumption before and after EEM implementation. This approach mirrors the energy benchmarking concept, constituting a longitudinal evaluation of EEM effectiveness.

- **Baseline identification for Energy Performance Contracts (GitHub)[38]**

The aim of streamlining the Measurement and Verification (M&V) process, which is a crucial step for Energy Service Companies (ESCOs) in managing Energy Performance Contracts (EnPCs), is pursued by the Energy Performance Contract (EnPC) management application. Typically, the M&V process is manually managed with Excel sheets, leading to time-consuming and error-prone outcomes. This makes the process inefficient and labor-intensive.

To standardize the M&V process, a solution has been developed by the BIGG consortium using the AI toolbox, with the goal of accurately and flexibly identifying baselines. The core modules of the toolbox were employed to construct a pipeline that can adapt to the requirements of any EnPC contract and facilitate the identification of a consumption baseline regression model from historical data, using input data such as weather, occupancy, and calendar information. High flexibility is provided by the developed pipeline regarding the types

[36] https://github.com/biggproject/A1-Benchmarking

[37] https://github.com/biggproject/A2-EEM-assessment

[38] https://github.com/biggproject/A3-EPC-baseline-identification

of models employed for baseline identification, allowing users to customize the resulting models to their needs while ensuring interpretability for non-expert users. The models that are identified can then be used as building blocks for global solutions, such as those developed within the context of the BIGG projects, where the entire M&V process, including savings tracking, assessment of financial benefits for both ESCOs and building owners, reporting, and more, is managed comprehensively from start to finish.

- **Occupancy pattern detection for Comfort and energy optimization (GitHub)[39]**

Addressing the needs of both energy efficiency and occupant comfort presents a challenge for energy experts. Existing building management systems focus on comfort but neglect the growing demand from building owners to reduce energy expenses. Additionally, these systems often fail to consider factors like occupancy, weather forecasts, and special events. To overcome this, HVAC controllers need to incorporate new information sources such as weather data and occupancy patterns to decrease energy consumption without compromising comfort. Notably, accurately identifying occupancy patterns is crucial for determining optimal energy-saving strategies, but manual identification can be intricate.

The AITB gives a solution designed to automatically detect occupancy patterns within buildings. It ingests the activity of people of households via the IoT sensors and considers the holidays. The AITB lightens the workload for energy experts while enhancing the accuracy of collected data. As a result, this toolbox has become an indispensable asset for energy experts striving to enhance energy efficiency and create comfortable building environments.

- **Energy consumption forecasting (GitHub)[40]**

Forecasting energy usage plays a vital role in effective energy management and planning. Precise predictions benefit both energy providers and consumers by guiding well-informed choices. Introducing an intelligent algorithm for energy consumption forecasting significantly enhances forecast accuracy, fostering the growth of sustainable and cost-efficient energy systems.

This algorithm aids energy suppliers in orchestrating production and distribution, optimizing consumption, and preventing power outages. For energy consumers, precise predictions enable efficient energy management, cost reduction, and leveraging off-peak periods. Moreover, the algorithm identifies consumption trends, informing efficient energy policies and systems that work towards a dependable and sustainable energy future. The result is a more resilient and eco-friendly energy system.

- **Gas demand response (GitHub)[41]**

This pilot is dedicated to designing an ingenious demand response (DR) plan by tapping into the adaptability of gas consumption in residential heating. This collaborative approach engages customers, allowing gas providers to sidestep unwarranted expenses and diminish CO2 emissions. The pilot's primary goal is to hit a specific gas consumption target, effectively fine-tuning energy efficiency, curtailing gas usage, and yielding economic savings along with environmental advantages.

The pilot case realizes its ambition through collective management of multiple boilers, orchestrating their gas consumption to meet the stipulated objective. In this orchestration, user comfort remains paramount – residences must not swing to uncomfortable temperature extremes

[39] https://github.com/biggproject/A4-Occupancy-pattern-detection

[40] https://github.com/biggproject/biggpy/tree/usecase14/UC14

[41] https://github.com/biggproject/biggpy/tree/main/gas_demand_response

To realize this vision, a reinforcement learning (RL) approach is adopted, shaping a demand response controller policy. This RL agent learns from historical or simulated data, discerning optimal actions based on raw input data. The distilled policy is subsequently applied in real-world scenarios to execute the demand response strategy. The specific actions of the policy pivot on input data and the DR objectives, whether it involves tweaking boiler operations to align with target gas consumption or refining energy usage in response to evolving conditions.

III.3.4. Output layer components

Regarding the RAF architecture, the output layer components are the components located at the end BIGG processing pipelines. These components are responsible for collecting the added-value information created by the BIGG pipelines to pass them to the end-users or external systems.

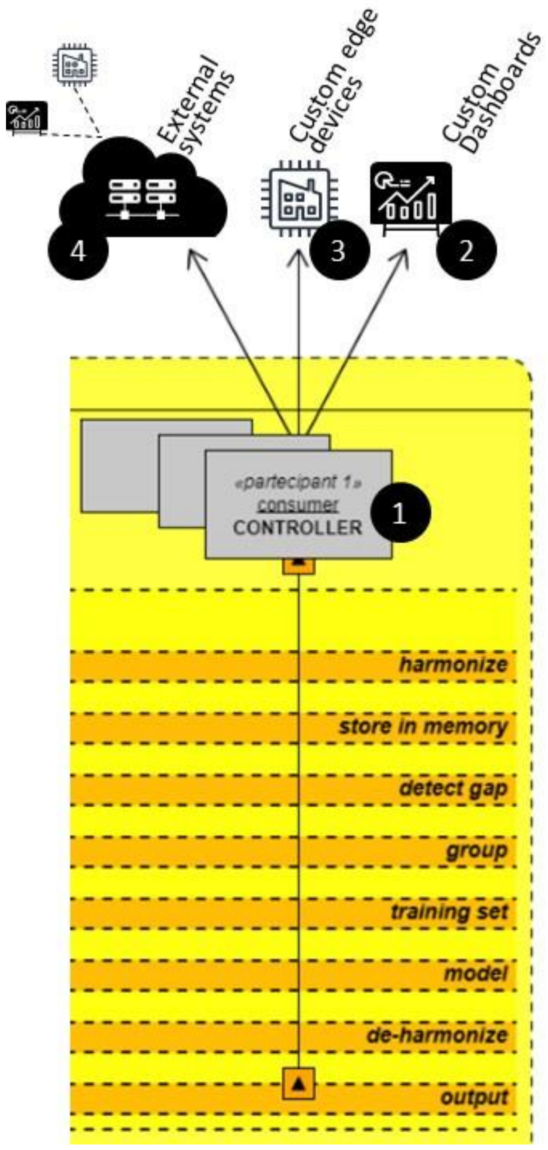


Figure 28) Output layer consumer components positioning in the RAF architecture

The output layer components are controller type-of-components regarding the RAF architecture. They consume high-level output messages from the message bus to process valuable knowledge created by the BIGG system. They are then responsible for providing this information to external systems. To achieve, they consume standard output messages from the communication bus but need to have a custom technical implementation to process them and expose output data in a required presentation for specific end-users or to implement the required protocols to communicate with external systems. Some examples are provided below:

1. A custom dashboard connection can be implemented in a controller as an output of a BIGG pipeline. Such a dashboard is custom because it has to be tailored to clients' business cases which are very specific (e.g., Energis Cloud dashboard).
2. Some business cases implemented with the BIGG architecture can aim at controlling buildings' automations (e.g., triggering the change of a temperature setpoint in a room based on an analysis performed in the BIGG toolbox). In that case, a custom controller needs to be created to communicate with edge devices controlling building systems.
3. In some situations, customer business cases are so complex that they require to be driven by external systems. These systems, that may be pre-existing, need to receive information from the BIGG pipelines to manage complex scenarios where custom dashboards are created or building devices are managed. In such case, a custom controller must be created that captures the knowledge created by BIGG toolbox and transfers this information to an external system, using the appropriate communication channel.

III.3.5. Integration Layer Components

All the components described in this document need to be organised in pipelines to implement specific big data processing scenarios tailored to the particular business cases. This section will describe the components that can help to organise BIGG components, using state of the art architecture patterns such as choreographed architecture or orchestrated architecture. The first paragraph of this section will introduce some enabling components that can be used to make the integration of other BIGG components more efficient and more versatile.

III.3.5.a. Enabler components

III.3.5.a.1. API Gateway

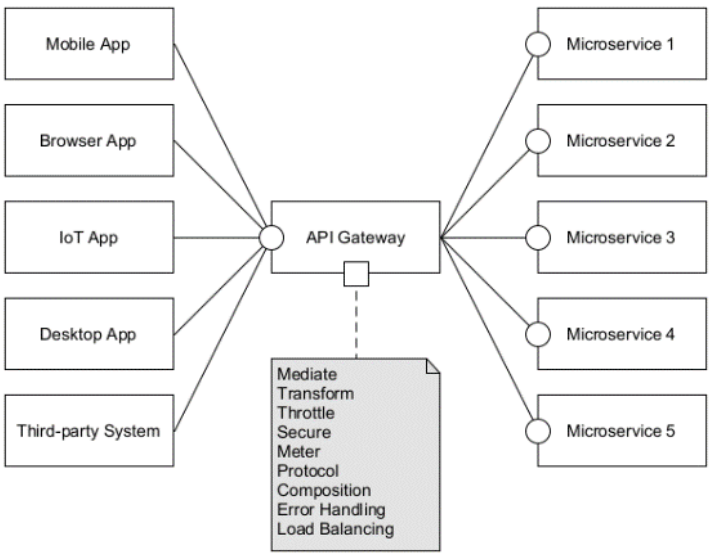


Figure 29) API gateway component

An API gateway is a microservice that has a simple purpose: to be aware of all other microservices in the system and to expose all of their APIs as a unique entry point to external calls:

An external caller could be a mobile app, a web app, an IoT device and a third-party system. In every case, this caller will use a unique front end with a unique API call. Behind the scenes, the API Gateway will receive this call and will use all the necessary microservices to process it and to give a response to the caller. Through this decoupling mechanism, the API Gateway could offer a lot of value-added services: it can, for example,

- transform the content and the protocol of the call;
- offer a security layer protecting the microservices from unauthorized calls;
- throttle the calls in case of performance degradation;
- as a single entry-point, meter the overall performance of the system;
- centralize the error handling;
- balance the load on multiple instances of the microservice to increase throughput.

The implementation of this component can be based on ZooKeeper,[42] an open-source project that "enables highly reliable distributed coordination". ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. ZooKeeper allows distributed processes to coordinate with each other through a shared hierarchical namespace of data registers (these registers are called znodes), much like a file system. It is a necessary component for Kafka, and we can leverage this dependency using some of its functionalities. On top of ZooKeeper, to facilitate the usage of its API's, a tool like Curator[43] can be used. The schema in Figure 30) depicts the usage of Curator library.

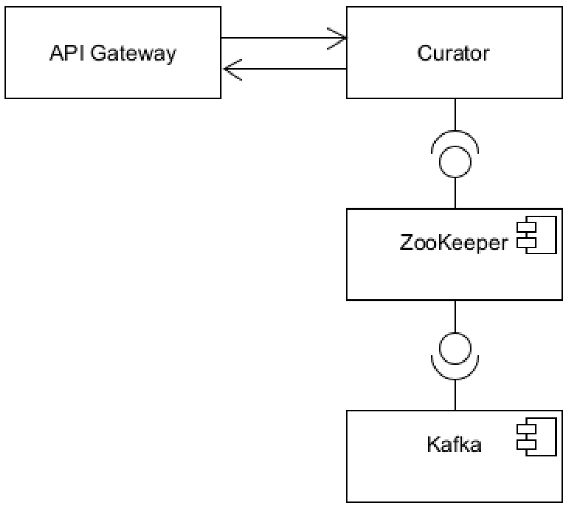


Figure 30) API gateway component technical implementation

With Curator, using ZooKeeper to store a microservice configuration within a ZNode "service path" is very simple:

```
// json object mapper
// to serialize and deserialize configuration
ObjectMapper mapper = new ObjectMapper();
```

[42] https://zookeeper.apache.org/

[43] https://curator.apache.org/index.html

```
// instantiate and start a client pointing to ZooKeeper instance and
// with the desired retry policy
CuratorFramework curatorFramework =
CuratorFrameworkFactory.newClient("localhost:2181", new
ExponentialBackoffRetry(5000, 10, 120000));
curatorFramework.start();

// creating of a node
String node = curatorFramework
.create()
.creatingParentsIfNeeded()
.withMode(CreateMode.EPHEMERAL)
.forPath( "/services/bigg/ingestor", mapper.writeValueAsBytes(<object describing
ingestor service >));

// read the array of bytes stored in path
byte[] result = curatorFramework.getData().forPath(node);

// update of a node
Stat stat=new Stat(); // storage node information
curatorFramework.getData().storingStatIn(stat).forPath(node);
stat=curatorFramework.setData().withVersion(stat.getVersion()).forPath(node,
mapper.writeValueAsBytes(<object describing ingestor service >))

// delete of a node
curatorFramework.delete().forPath(node);
```

As a possible microservice configuration, the following structure can be used:

```
public class ServiceDescriptor {

    private String name;
    private String description;

    private String host;
    private String port;

    private String apiUrl;
}
```

Once all the microservices are auto-registered in ZooKeeper's ZNodes, the API Gateway can easily retrieve their configurations and URLs:

```
List<String> uris = curatorFramework.getChildren().forPath("/services/bigg");
```

Then, for a basic API Gateway implementation, we need to forward the external calls to the proper pipeline of microservices.

III.3.5.a.2. Commander

Exploiting the functionalities offered by the API Gateway, the Commander will be the real orchestrator of the BIGG platform. The main idea is the Commander will send to the API Gateway a call organized in two parts:

- **Pipeline:** a list of services the API Gateway must call in the ordered sequence using the response of the previous service as the request of the next;
- **Data:**the byte array that represents the serialized stream of the input for the first service to call.

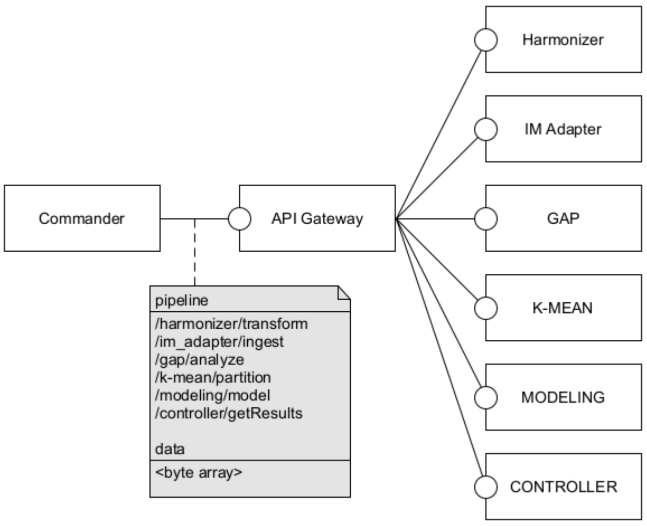


Figure 31) Commander component

In a second version could be implemented other functionalities like persisting a library of predefined pipelines.

III.3.5.b. Components of a choreographed architecture

Just like in a ballet, where each dancer exactly knows his movements and hoto interact with the other dancers, in a choreographed architecture each microservice knows its inputs, its outputs and how to communicate with the other microservices to realize the business process. Choreography is an event-driven process started by the incoming message and the pipeline is determined by the microservices implementation and configuration (in terms of input/output topics to consume from and produce to). Figure 32) shows the implementation of a pipeline using proper topic names and configuring accordingly the microservices.

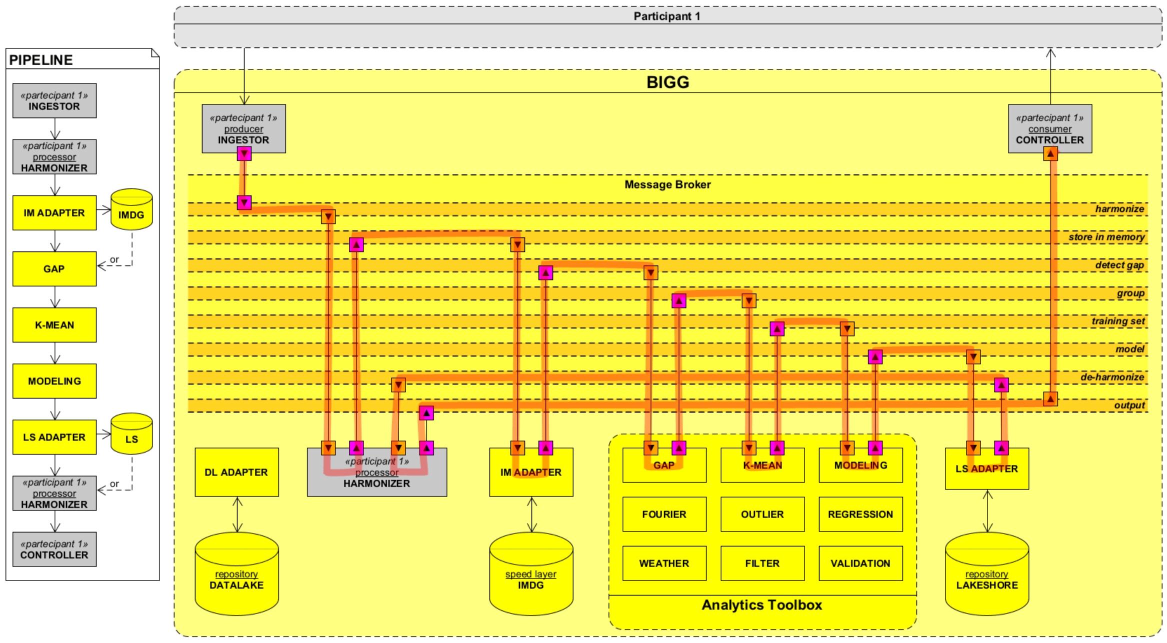


Figure 32) Example of a choreographed flow in the BIGG RAF

III.3.5.c. Components of an orchestrated architecture

If, on the one hand, a choreographed architecture is well suited for the “ingestion” process in the RAF that is, by nature, event driven, on the other hand we need another approach if we want to make the RAF usable by third-party systems or external entities like participant’s users in a request-response fashion. In this second case, a way to dynamically define a “pipeline” of calls and to submit this sequence to the “black-box” BIGG RAF is to be provided as well. The diagram in Figure 33) shows the components that must be present in this scenario.

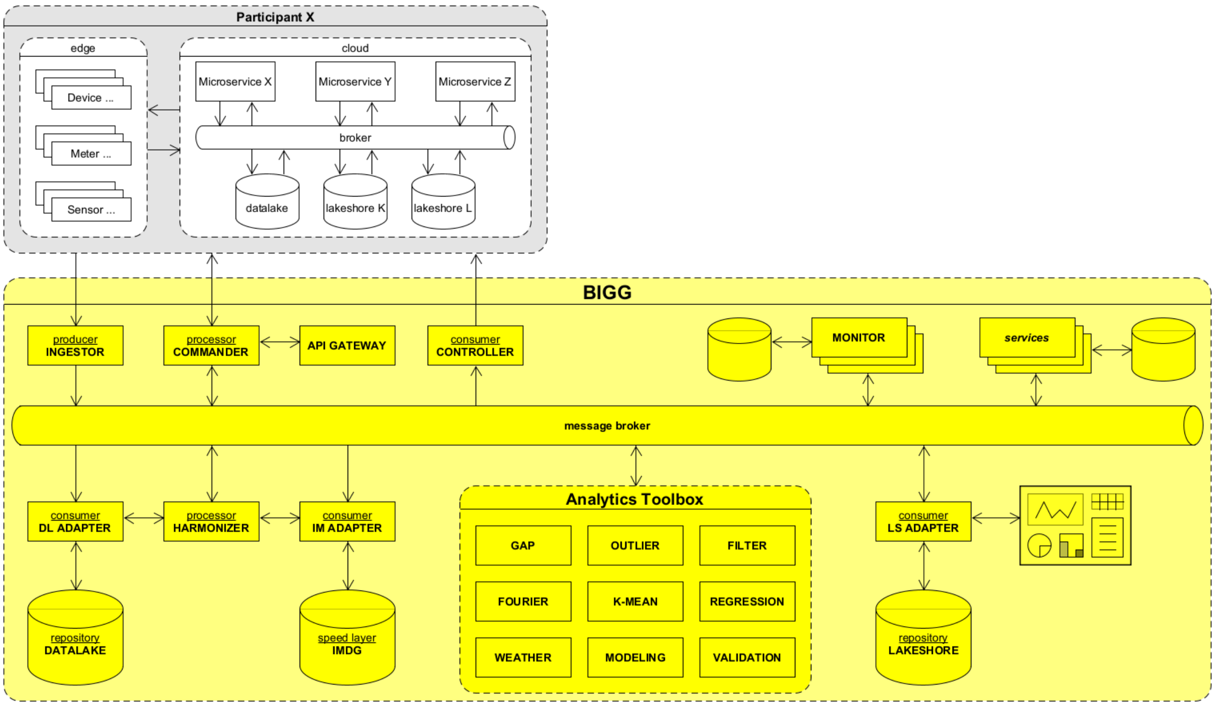


Figure 33) Example of an orchestrated flow in the BIGG RAF

IV. DEMONSTRATIONS IMPLEMENTATIONS

The goal of this section is to present the **different specific architectures** setups for the different BIGG business cases with an emphasis set on the supporting services modules of the BIGG analytics toolbox that are used withing these pilot implementations.

The goal here is to validate that the objectives of the description of work are fulfilled by the business-cases-supporting architectures implemented in BIGG. These objectives from the description of work include:

- 1 ([…] *implement a flexible and open-source big data reference architecture*[…]),
- 3 (*[…]develop an open,[…]building-related data analytics toolbox […]*)
- 4 (*[…] BIGG Data Analytics Toolbox over the BIGG Data Reference Architecture 4 Buildings […] supporting different multi-party business cases […]*)

For every BIGG business case, the flow of information among the different BIGG components in the BIGG RAF is described demonstrating a pragmatic usage and implementation of the BIGG assets.

IV.1. Unified big data demonstration (Catalonia Spain)

For all Spanish Pilots related use cases the same architecture is used. This architecture is deployed on the ENMA[44] commercial big data infrastructure owned by CIMNE.

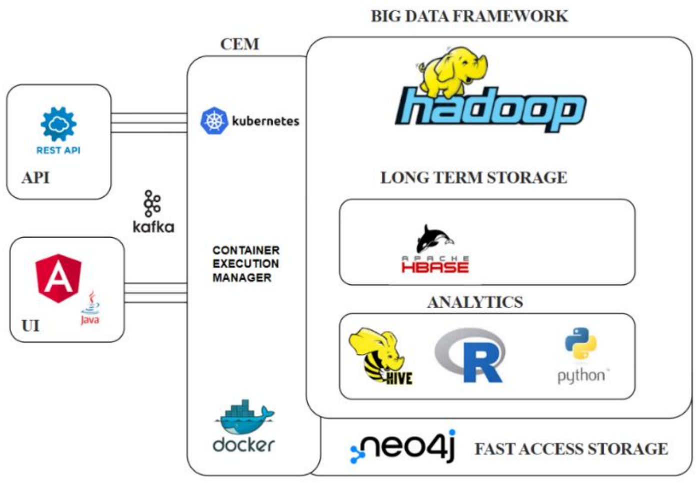


Figure 34) CIMNE ENMA big data infrastructure

The main characteristics of the ENMA infrastructure are the following:

- BIGDATA FRAMEWORK: Hadoop is the "backbone" of the whole system that can develop scalable modules operating in a distributed way, using the Map-Reduce paradigm. It is a Distributed File System (HDFS) where all components reside, e.g., HBase or Hive.
- LONG TERM STORAGE: It is a distributed and scalable database to store data without losing access to read/write on them. HBase is a random-access database capable of holding billions of rows with real time access.

[44] https://www.beegroup-cimne.com/enma-big-data-architecture/

- FAST ACCESS STORAGE: To maintain scalability, the system is provided with a fast access database that will provide the required information to the API and visual interfaces.
- ANALYTICS: The analytics component is the data warehouse to bind to HBase and NEO4J. Hive creates external tables from HBase as Storage Handler (which means HBase holds the data) and then makes HBase tables queryable with a SQL-like syntax.
- CONTAINER EXECUTION MANAGER: Kubernetes[45] is a portable, extensible, open-source platform for managing containerized workloads and services. It is used to keep running and orchestrate together all the required API and User Interfaces. It also manages the execution of analytics modules and any other execution requirement in the system. Each execution will be isolated thanks to the Docker container. Keeping the infrastructure dependency agnostic.

The architecture, components, and pipelines designed and developed in the BIGG project have been implemented on this infrastructure.

Description of the processes implemented for the Spanish pilot UCs

Ingestion process

In the first part of the process, we have the ingestors, which are custom-made applications to collect information from different sources. These sources can be found in different formats such as Excel/CSV files, REST APIs or Web forms. The ingested information is channeled through the Kafka broker to different storage services (raw data) or preprocessing (harmonization). Kafka is the main component in the implementation for this part of the process.

BIGG ingestors are used for all UCs in the Spanish pilot. The implementation of the ingestors in this pilot has been done by integrating them directly into the production code.

It is highlighted that for some sources it is necessary to launch injectors in the MapReduce paradigm to distribute the executions due to the large amount of data and the slowness of the API responses.

Link data and harmonization process

In cases where harmonization-pre-processing is required, the harmonization component collects data and information from the Kafka broker and processes it. This information is harmonized using the harmonizers proposed in BIGG; for some sources, a custom BIGG-derived harmonizer developed by CIMNE is used.

To execute the harmonization process, a custom mapper has been created for each data source. This mapper contains the equivalence between the data in the initial format (raw data) and the data in the format of the BIGG model (harmonized data). The harmonization component collects the ingested data, searches for the source mapper, and performs the transformation of the data.

Data storage

The harmonized data is then published again in the Kafka broker and made available to the consumer which oversees the harmonized data store. The component stores the data in the databases according to the type of data: a Neo4j graph database for static information and an HBase time series database for time series.

The following Figure 35) shows the process followed for the ingestion, harmonization and storage of each data source involved in each use case.

[45] https://kubernetes.io/

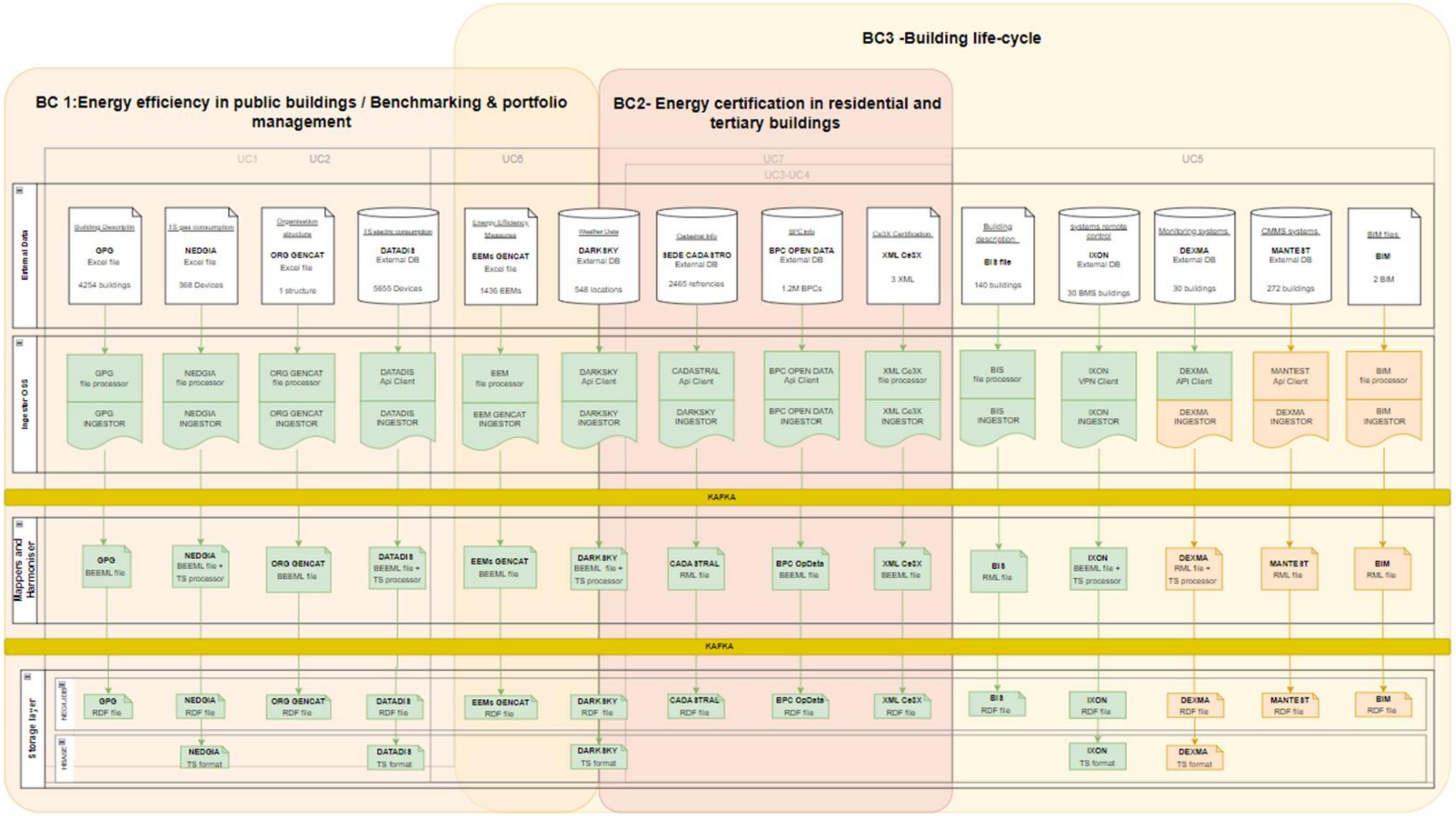


Figure 35) Spanish pilot unified BIGG RAF implementation overview

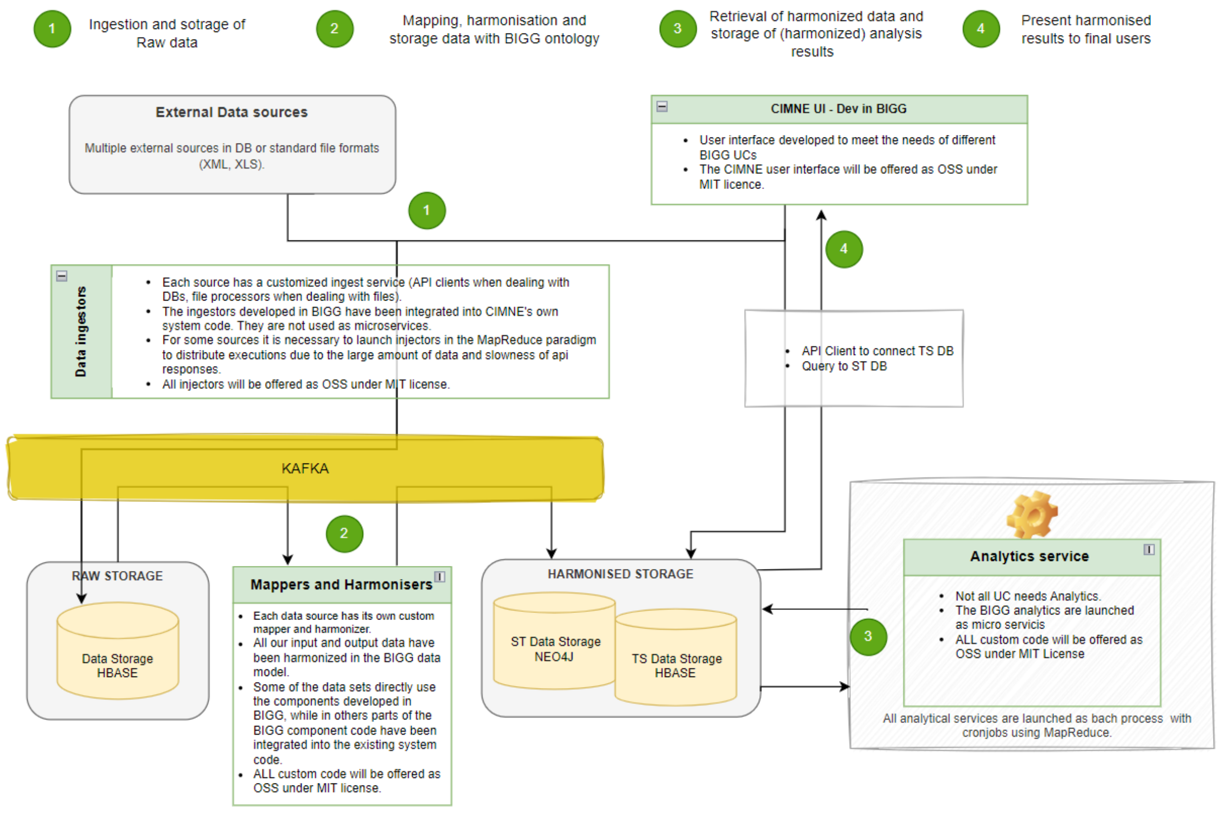
<u>Analytic process</u>

Once sufficient data has been collected and on a regular basis, the various batch analyses are launched. The analytics component obtains the required harmonized data by reading it directly from the databases.

Once the data has been obtained, the Analytics Toolbox provided by BIGG is run. The analytics component generates the harmonized results in the format of the KPI extension of the BIGG ontology and these are stored back in the database.

Finally, the visual user interfaces will fetch the information directly from the databases and display the results.

The following Figure 36) shows a global scheme of the process followed in the Spanish pilot.


1 Ingestion and sotrage of Raw data
2 Mapping, harmonisation and storage data with BIGG ontology
3 Retrieval of harmonized data and storage of (harmonized) analysis results
4 Present harmonised results to final users
External Data sources
Multiple external sources in DB or standard file formats (XML, XLS).
CIMNE UI - Dev in BIGG
1
4
Data ingestors
KAFKA
2
RAW STORAGE
Data Storage HBASE
Mappers and Harmonisers
HARMONISED STORAGE
ST Data Storage NEO4J
TS Data Storage HBASE
3
Analytics service

CIMNE UI - Dev in BIGG

- User interface developed to meet the needs of different BIGG UCs
- The CIMNE user interface will be offered as OSS under MIT licence.

Data ingestors

- Each source has a customized ingest service (API clients when dealing with DBs, file processors when dealing with files).
- The ingestors developed in BIGG have been integrated into CIMNE's own system code. They are not used as microservices.
- For some sources it is necessary to launch injectors in the MapReduce paradigm to distribute executions due to the large amount of data and slowness of api responses.
- All injectors will be offered as OSS under MIT license.

- API Client to connect TS DB
- Query to ST DB

Mappers and Harmonisers

- Each data source has its own custom mapper and harmonizer.
- All our input and output data have been harmonized in the BIGG data model.
- Some of the data sets directly use the components developed in BIGG, while in others parts of the BIGG component code have been integrated into the existing system code.
- ALL custom code will be offered as OSS under MIT license.

Analytics service

- Not all UC needs Analytics.
- The BIGG analytics are launched as micro servicis
- ALL custom code will be offered as OSS under MIT License

All analytical services are launched as bach process with cronjobs using MapReduce.

Figure 36) Spanish pilot demonstration workflow

IV.2. Building utility demonstration (Business cases #4 and #5 in Athens)

The Energis.Cloud platform used for BC4 and BC5 is designed according to the microservice architecture principles. Each microservice is loosely coupled with the others, so that changes in design, implementation or behaviour applied on one would not affect the other components.

However, the microservices can communicate with each other by consuming and producing messages through the Kafka bus, as described in the previous sections. In the "Building utility demonstration" architecture, as depicted in Figure 37) we can distinguish different categories of components:

- **Ingestors**: initiating the data collection process by interfacing directly or indirectly with external systems. In this category, we can differentiate between solicited and unsolicited ingestor. The first one is active and "solicits" an external system that is exposing a service to obtain data, while the second one is passive and receives data based on a pre-defined protocol.
- **Processor**: responsible for harmonizing external data to our system data model. It transforms messages following an arbitrary format into "processed messages", containing measurements organized in a tabular form and complementing them with useful metadata.
- **Adapter**: services that retrieve payloads from the message broker (Kafka) and store its content on a "lakeshore" resource as a DB, a third-party system, a remote host and so on. Optionally, this component can also "adapt" the "processed messages" to make it comply with specific rules before sending the data to the EMS platform, e.g., convert the time stamps of the timeseries to the local time zone.
- **Speed-Layer**: based on Hazelcast, which is an in-memory data grid that caches frequently accessed information to improve the performance of the system.
- **API Gateway**: Intermediate layer between front end of the platform and the back end. It intercepts all the requests from the front end or from external services and redirects them to the corresponding service.
- **Computation service**: the core of Energis Cloud, responsible for performing advanced calculations, aggregations and handling alerts and time-series granularities.

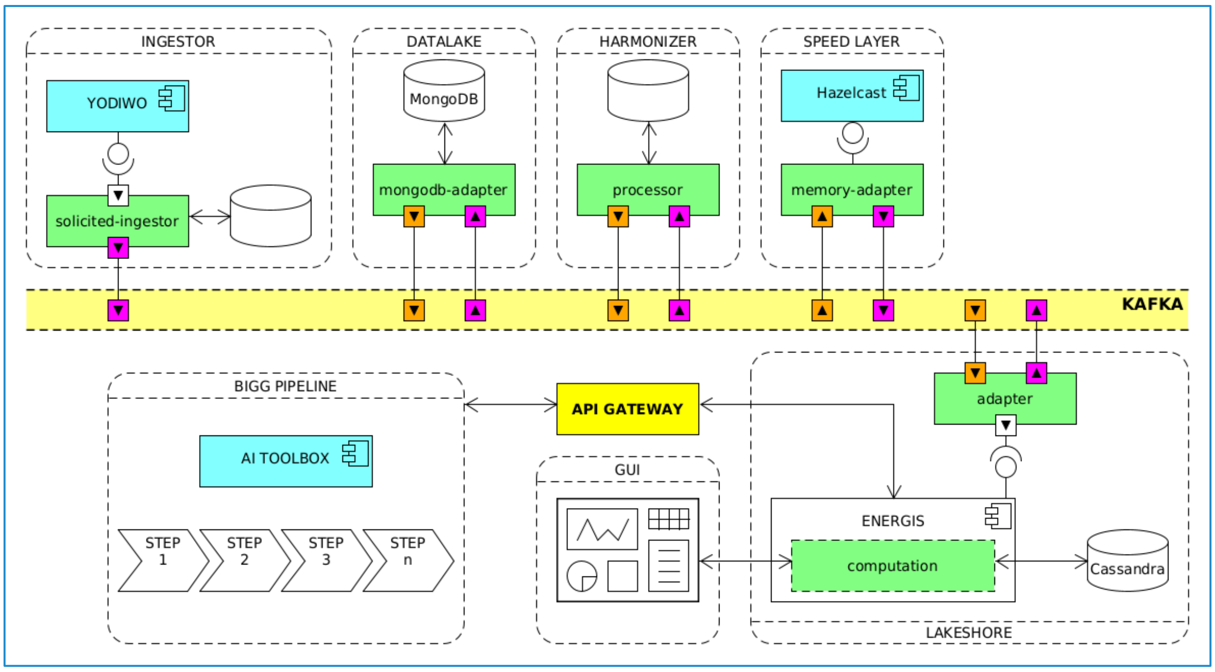


Figure 37) BC4 and BC5 High level architecture diagram

After the brief introduction to our system architecture, we can now describe how the business cases 4 and 5 (BC4 and BC5) were implemented.

First, the initial step was to enable the data collection from our data provider. We integrated a new connector “Yodiwo” into the “solicited-ingestor” to retrieve electricity consumption, sensor and actuators data in a scheduled manner from the Yodiwo platform of Cordia. We also integrated ingestion of weather data from the Weatherbit [46] service as an additional microservice since they are a key element both for the business cases and the related AI Toolbox pipelines.

These data were harmonized according to the Energis.Cloud data model and stored internally using Kairos DB, which is a Java-based time series API that leverages Cassandra as its underlying distributed database. Also, other external services can collect these data from the platform using the APIs offered by the API Gateway. Several improvements of the Energis.Cloud data model were necessary to comply with the BIGG RAF, in particular to add flexibility in the way data is organized and can thereafter be used by the UI and the Computation module, and ultimately to enable scalability of the created solutions. This was materialized under the form of new features called asset roles and meta-metrics.[47]

Once data were available on Energis.Cloud, we used the features offered by the system (using internal data quality KPIs, dashboards, alerts) to make sure that they were flowing correctly and without interruptions. Then, to build the set of solutions required for BC4 and BC5, we:

1. Computed all the necessary KPIs: this was done thanks to Energis.Cloud’s Computation module;

[46] https://www.weatherbit.io/

[47] The Asset roles and Meta-metrics concepts have been later relabeled into Metric groups and Metric definitions, with an additional concept of Metric templates gluing the two concepts. However, for past reference purpose, we included the original names in the text. More documentation about these concepts can be provided on request.

2. Identified the required regression models; this was done using the BIGG AI Toolbox, and more specifically using the pipelines described in the GitHub as Applications A3 and A4, which are further described below;
3. Integrated all these ingredients as a consistent set of dashboards and reports, all linked to each other, which each actor can access through the Energis.Cloud UI.[48]

We now provide more information about the process that led to the creation of the BIGG pipelines deployed for business case 4 and 5, respectively called Application A3 and A4 in the GitHub repository.[49]

We first created a CLI python tool with configurable parameters, as described in Section II.3.1. , to launch each pipeline as an on-demand task.

The python code retrieves the data from the Energis.Cloud platform according to the parameters provided as input and leverages the analytic features of the AI toolbox to perform all the operations needed to generate an ML model.

Specifically, the python tool transforms the input data into the format needed by the adopted data science libraries, e.g., a Pandas Dataframe, performs a data quality check to ensure that a model can be generated for the input dataset, detects and exclude outliers and finally runs an optimizer based on the GridSearch nested cross-validation to select the best performing pipeline between the ones pre-existing in the initial grid.

Other than the final model, also the preliminary steps that create the engineered features are part of the optimization, meaning that also specific parameters of the final model or of the transformers are optimized.

An example of pipeline is shown in Figure 38). Here, before training the model on the initial dataset, several engineered features are added to the data matrix, e.g., public holidays, heating and cooling degree days, weekly profile of the electricity consumption of the building.

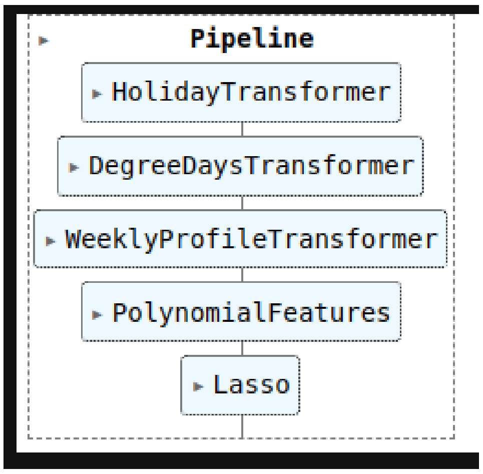


Figure 38) Example of cross-validated pipeline

[48] See Deliverable D6.3 for more information about the solutions created for the business cases BC4 and BC5. Note that the solution delivered for BC5 was also complemented by a Grafana dashboard allowing to manage control actions. See Deliverable D3.2 for more information about the Energis.Cloud UI in general.

[49] See the GitHub repository of the project or Deliverable D5.2 for more information about the pipelines of Applications A3 and A4 or about the BIGG AI Toolbox in general.

Then, the initial data are transformed using the Polynomial Features, where a new feature matrix consisting of all polynomial combinations of the initial features with a degree of at most the specified degree is generated.

Finally, a linear regression model is trained on the transformed data repeatedly using the Grid Search cross-validation framework.

The resulting model is a polynomial equation that is uploaded to the Energis.Cloud platform in a formula metric together with the associated performance indicators using the API Gateway.

IV.3. Gas & electricity demonstration (Business case #6 in Greece)

For business case 6, two different architectures had to be designed and implemented to properly accommodate the needs for the two use-cases it comprises. This was due to the fact that the two pilots were housed in two separate platforms with different prerequisites, different requirements and specific security and data management constraints for each use-case.

Use case 14

The requirements for Use-Case's 14 platform (Electricity DR) pushed us to design a hybrid pipeline based on the Reference Architecture described for the BIGG project. This architecture houses several components for the BIGG Toolbox, on-site and communicated with the AI components of the AI-toolbox through an API Service that served harmonized data to the AI Toolbox that then proceeded to consume, process and analyse the electricity consumption data needed to fulfil the scope of the Electricity DR.

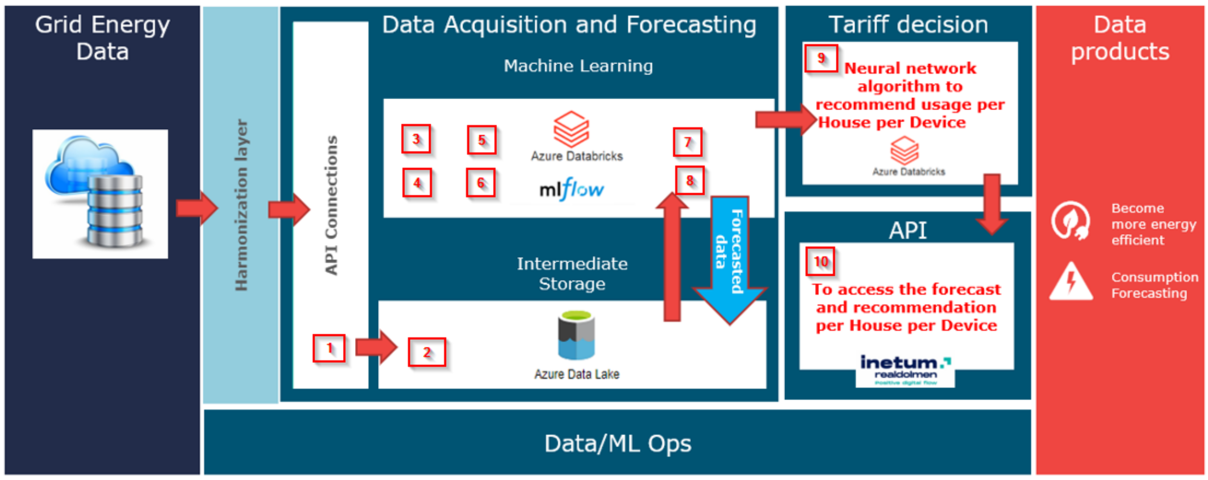


Figure 39) UC 14 High level data architecture diagram of the solution with the corresponding tools/resources that is used.

The electricity demand response pipeline is described in a step-by-step approach:

1. Data Acquisition and Storage:
The process begins with the collection of data from IoT devices installed in households in Greece, which continuously transmit meter readings and smart device readings every 30 seconds. For UC14 we only focus on smart meter readings only. These readings are ingested via Heron's database using the ingestors mentioned in Section III. The data format and structure are detailed in Section III.3.1.c. . This telemetry data serves as the foundational information for energy consumption analysis. To do the pre-processing and storing the meta-data of the following steps, we use an Azure Data Lake (an component external to the RAF). The Azure Data Lake is a deviation from RAF but provides additional flexibility of exporting the data to 3rd party tools for doing analytics and API deployment (see step 10).

2. Pre-processing and Cleaning:
Upon data retrieval, a series of pre-processing steps are executed. This involves data clean-up, alignment of values in time, and addressing any inconsistencies or missing data points.

The AITB modules used for this step can be found in Section III.3.3.c. . The goal is to ensure a standardized and reliable dataset for further analysis.

3.Outlier Detection:
An essential part of the process involves the identification and handling of outliers within the dataset. The outlier detection module, part of the AITB (See Table 2, Table 3), is employed to recognize and manage anomalous or irregular data points that might significantly impact forecasting accuracy.

4.Feature Engineering:
Calendar components, such as day of the week, hour of the day, and potentially other temporal features, are extracted and integrated into the dataset. For the description of the functions, one can consult Section III.3.3.c. . These features aim to uncover correlations between energy consumption patterns and temporal factors, thus enhancing the accuracy of predictions.

5.Energy Profiles and Clustering:
A distinctive step in the pipeline involves the generation of energy profiles, utilizing clustering techniques to identify similar consumption patterns among households. This “double” clustering procedure groups households based on their energy profiles, resulting in an encoding for each profile derived from its proximity to cluster centroids. These profiles and associated clustering distances are integrated as additional input features for the neural network model.

6.Machine Learning Model Training:
Multiple machine learning models (recurrent neural network, RNNs) are trained using the prepared dataset, incorporating power consumption data, device information, calendar components, and energy profiles. These models predict hourly energy demand for individual households.

7.Data Augmentation for Forecasting:
To extend the forecast duration to 24-48 hours, a data augmentation technique is used. Predictions for the next hour, along with the preceding 23 hours of data, are iteratively employed to generate subsequent predictions. This iterative process enhances the accuracy and extends the forecasting horizon.

8. Model Evaluation and Selection:
Following model training, an evaluation process is conducted to determine the most accurate model for residential power consumption forecasting. This involves comparing predicted power consumption values against actual consumption, selecting the model that provides the most precise and reliable forecasts.

9. Tariff Integration and Cost Calculation:
Once forecasting is complete, the system incorporates the green tariff for upcoming hours to calculate the overall cost of consumption. The green tariff is not exactly the cost of the energy but the greenness of the energy i.e. the ratio of energy generated from renewable and non-renewable sources throughout the day. This data is again obtained from an API provided by Heron. This calculation accounts for the customer’s energy usage and the corresponding rates for the given time period. This step is an addition to the RAF to attain UC14 solution requirements.

10. Recommendation API
This step is also a deviation from RAF. However, this step needed to be incorporated so that the providers can seamlessly integrate the output of the recommendations into their own framework to send these recommendations to the customers. As due to GDPR requirement

only the energy providers (in this case Heron) have the confidential information of the customers, this step was necessary to make the pipeline scalable for larger customer base.

These steps explain all the functional components of the pipeline.

Next lines will provide more detailed exploration of the API structure, as this step is pivotal in opening the potential for integrating the end-product of the pipeline into our partner's internal system. For this use case we have developed two sets of APIs. Below we discuss the structure of the two APIs. This detailed examination will unravel the complex framework behind their operational dynamics, shedding light on the precise technical components orchestrating the smooth integration within software ecosystems.

A. Two Deployed Sets of APIs:

- **First API for Next Day Recommendations:**

Functionality: These API offer recommendations for the upcoming day, available a day in advance, aiding proactive energy management.
Objective: To anticipate and provide suggestions for optimizing energy usage based on predictive models.
Timing: Recommendations are accessible a day prior to the specified consumption day, facilitating early access for planning.

- **Second is for Historical/Archival Recommendations:**

Functionality: Provides access to past recommendations that has been provided by the pipeline.
Objective: Serves for retrospective analysis, trend identification, and understanding energy recommendation patterns over time.

B. Priority Levels (P1 and P2):

- **P1 (Priority Level 1):** Represents critical recommendations, crucial for immediate interventions or optimizations corresponding to two lowest ISP scores of hours of renewable to non-renewable ratio.
- **P2 (Priority Level 2):** Offers important but less urgent recommendations compared to P1, prioritizing less critical suggestions.

C. Priority-Based Access and Usage:

- **Order of Priority:** Categorization into P1 and P2 levels provides a structured hierarchy based on urgency and importance. An SMS is sent, based on the priority level.
- **Usage Scenarios:** Users make decisions and allocate resources based on these priority levels, addressing critical recommendations first. The smart devices are used to validate if the user has followed up on the recommendation or not.

D. Implementation and Utilization:

- **API Access and security:** Access to the APIs, based on their requirements for energy optimization and planning is currently anonymous. However, a user ID is required to deploy the API from the Azure Function apps. This corresponds to a Tier 2 security layer. Since the data is anonymized, this level of security is sufficient to comply with the ISO 9001 protocol.

- **Interpretation of Recommendations:** Use recommendations for implementing strategies, managing consumption, and planning energy optimization activities.

E. Potential Applications:

- **Real-time Decision-Making:** Use P1 recommendations for immediate intervention in energy management systems.
- **Long-term Planning and Analysis:** Leverage both P1 and P2 recommendations for historical analysis, long-term planning, and identifying consistent energy consumption trends and customer behaviour.

Conclusion about UC14:
This comprehensive pipeline, provided in the repository as an OSS application,[50] incorporates AITB's modules, from data ingestion to model training and evaluation, leveraging advanced machine learning techniques. This system provides a versatile approach to managing energy consumption, offering both proactive and retrospective recommendations for energy optimization and planning. The integration of IoT data, feature engineering, clustering, and model selection culminates in an accurate and reliable predictive model for residential power consumption.

Use case 15

The Natural Gas DR Use-case (15) demonstrates and exploits the flexibility potential of buildings in natural gas. The focus is on characterizing the availability and distribution of flexible loads in the buildings by analyzing a plurality of data captured from heterogeneous sources like devices deployed at consumer premises smart meters, heating controllers, IoT, sensors, etc.

The whole architecture is based on microservices using Docker containers and a container orchestrator service. The microservice architecture provides the option to include all the BIGG components needed on-site and refrain from having to communicate through the Internet to produce recommendations and data. To this end, all the required BIGG Components are dockerized and deployed on a platform depicted in Figure 40).

[50] https://github.com/biggproject/biggpy/tree/usecase14/UC14

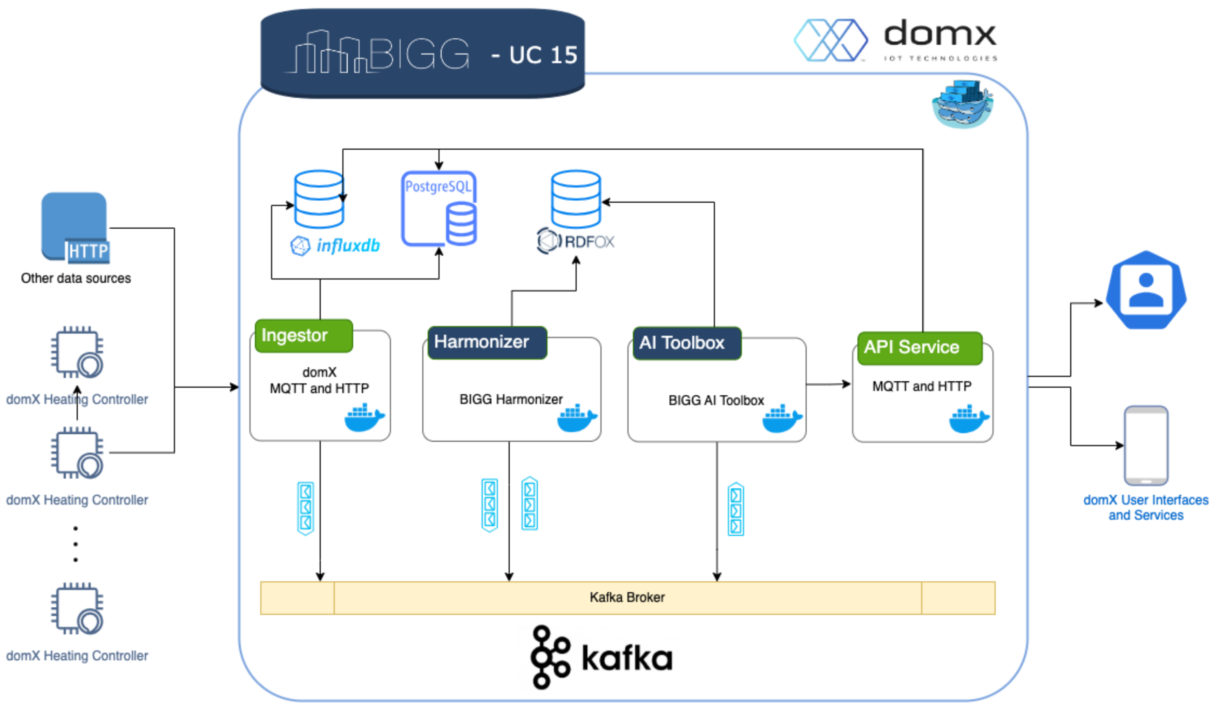


Figure 40) High level Architecture diagram for Use Case 15 – Natural Gas

Notably, the architecture building blocks used for this specific use case are as follows:

- **Kafka:** Kafka is high performance, high-throughput distributed event store and stream-processing platform. From the start of the BIGG Project, KAFKA was deemed as an integral component of the whole Reference architecture it was decided to be integrated in the domX Platform as a way of improving the performance and complying with the RAF. On top of KAFKA we used KAFKA Connect which is a component that allows for hassle-free integration and communication with databases and external systems.

- **Ingestors:** The ingestion process for the domX Platform can ingest two types of data, real-time data that are generated from the IoT Devices used to control the heating elements in the households, and are consumed through an MQTT Broker, and statics data that are being handled through some HTTP APIs and are data that complement and enrich the real-time data. (weather data, home characteristics data etc.). The BIGG Ingestor "pattern" has been used to evolve existing domX core components that had an optimized capability of ingesting Use-case data from MQTT, HTTP and to publish data to a Kafka component out of the box while not requiring any modifications to be deployed and work seamlessly in production. We modified the ingestor that we were using to connect to Kafka and to publish the data consumed to a KAFKA topic ("*bigg_input_mqtt*").

- **DB Storage**: The storage we are using is split into two categories. Raw data storage and Harmonized data storage. The Raw data storage was being used in the domX platform before the BIGG project and it relies on InfluxDB and PostgreSQL. InfluxDB is a timeseries database that allows for the storage of datapoints that are being generated by the devices and are subsequently ingested by the Ingestion service. PostgreSQL is used to store Structured data that cannot be stored in a TimeSeries DB, such as building and user data and metadata.
  For the Harmonized data storage, an RDF storage platform for a variety of OSS

Implementations had to be chosen. Some of the candidates we evaluated were Neo4j, Stardog, GraphDB, Apache Jena, RDFox, etc. We decided to use RDFox mainly because it provided a docker container and was easy to integrate into our existing platform architecture.

- **Harmonization:** Harmonizing the data to a common format is the element of success of the BIGG Project and a pre-requisite to create common Tools for Ingesting, processing, and analyzing the data collected. The BIGG Harmonizer is the component in the BIGG RAF that brings the pieces of the data and the AI Toolbox together. The harmonization process happens in real-time as soon as the data are published by the devices and ingested by the ingestor services. To integrate the BIGG Harmonizer into domX's Platform, we had to dockerize the BIGG Harmonizer script, since otherwise we would not have been able to access the measurements and data published by Kafka since they all resided inside a Virtual Private Network that isonly accessible by container services. The dockerization required a Python base container and we also had to include a volume with a harmonization configuration file that was created specifically for Use Case 15 and Gas/Heating-related data. In addition, we had to create a wrapper around the original Harmonization Script developed in Python to stream data to it and from it using Kafka. After the harmonization process is finished all the generated data points are published to a Kafka topic ("*bigg_harmonized"*) and then are consumed by the DB Service and the AI Toolbox.

- **AI Toolbox:** The AI Toolbox developed in WP5 by Imec is the core component of the whole architecture. It receives real-time data from the devices and uses a series of containers to implement the Demand Response-Event functionality. The AI Toolbox requires roughly 1 month of historical data from each building/apartment to create a specific model. Afterwards it requires re-training every 24-hours to provide the best possible outcome. The training process happens by a training-container that is fired by a cron-job and requests data stored from the DB for every household. The DR-event process happens in real time, the service receives data from the Kafka topic and then processes and evaluates the data based on the DR requirements set through the User Interface and the scenario (increase/decrease consumption). Households are then graded based on their profile and the action required by the DR Scenario. The DR Event Container then proceeds to send DR Control Outputs directly to the devices through an HTTP API.
- **User Interface:** The User Interface which is used to display the control outputs produced by the AI Toolbox is the domX smartphone app and the domX Supplier web-based Dashboard. The smartphone application informs the user about upcoming DR Events and provides him/her the capability to opt-in or opt-out from the specific DR Event. The Supplier dashboard is used to monitor the entire DR Portfolio of the Supplier and can provide valuable information to the Portfolio Manager in terms of consumption and climate on a portfolio or device basis.

Conclusions about UC15:

The RAF pipeline was implemented successfully for the purposes of Use Case 15. DomX managed to use or evaluate all the components developed in the scope of the BIGG Project while also incorporating meaningful improvements. The UC-15 pipeline[51] manages to provide DR control over residential buildings for Natural Gas boilers and can demonstrate scenarios where the AITB can control a portfolio-wide consumption increase/decrease based on the supplier requirements.

51 https://github.com/biggproject/biggpy/tree/main/gas_demand_response

V. Conclusions

This document has presented the technical specifications and design of the BIGG architecture building blocks. This work has leveraged outputs of BIGG Deliverable 6.1, which studied the pilots' use cases from a technical perspective and emphasized the fact that a cloud-only solution should not be the only target of a BIGG Reference Architecture. In fact, if we want the BIGG architecture and more widely the BIGG components, to be exploited and deployed after the project, the BIGG components integration options must be versatile.

Thus, the Big Data Reference Architecture is a blueprint for designing and implementing big data solutions. It provides a high-level overview of the key components and processes involved in big data management, from data collection and ingestion to processing, analysis, and visualization. BIGG RAF is designed to be flexible and scalable, allowing organizations to adapt them to their specific needs. The BIGG RAF is specifically designed for the building energy domain: it provides a pipeline for collecting, harmonizing, and processing building energy data from diverse sources. The harmonization process is enabled by the BIGG Standard Data Model 4 Building Ontology, which provides a structured and standardized representation of data meaning and interrelationships. The harmonized data can then be used by the BIGG AI Toolbox to generate insights and predictions related to building energy consumption and performance.

To provide this high-level overview, the initial section (§II) of this document has presented a technical approach to ensure modularity and versatility of BIGG software components. It has proposed to structure the code of the BIGG components in different layers: (1) the business logic core, embedded in (2) an exposing interface (CLI, Web service or event messaging) which is (3) constrained using Docker technology. The components' code bases and deployment artifacts need to be centralized in a repository shared among users. Every user is then able to pull the components versions that fits the best his local architecture and update the components for future shared improvements.

The next section (§III) has presented a Reference Architecture Framework (RAF) describing state-of-the-art techniques to coordinate BIGG components, may the actual architecture deployment be local (on client's infrastructures) or in the cloud (on centralized shared infrastructures). The RAF provides architectural workflows for collecting, harmonizing, and processing building energy data from diverse sources. The harmonization process is enabled by the BIGG Standard Data Model 4 Building Ontology, which provides a structured and standardized representation of data meaning and interrelationships. The AITB leverages data analytics and AI/ML technologies to process the harmonized data and produce insights and predictions. The RAF enables data-driven excellence in the building energy domain, transforming data into an asset that propels the operations of various business cases.

Finally, the last section (§IV) has presented actual RAF technical implementations on various generic topics like Unified Big data services for ICAEN/ICAT, building utility services for Helexia/Cordia and Gas & electricity demonstration applications for Heron. This section has demonstrated the "pick and choose" nature of the BIGG assets and how BIGG components were used and integrated by each solution –provider within the project to build commercial-grade building energy solutions with BIGG OSS components.

As a final word, the BIGG RAF is an open-source framework, which means that it is freely available for anyone to use and modify. This makes it a valuable resource for organizations that are looking to develop their own big data solutions for the building energy domain.