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

D2.2 - Initial technical specifications and preliminary design of BIGG Architecture building blocks

Responsible:	CSTB
Document Reference:	D2.1
Dissemination Level:	Public
Version:	1.0
Date:	01/12/2021

Contributors Table

DOCUMENT SECTION	AUTHOR(S)	CONTRIBUTOR(S) to results	REVIEWER(S)
Section I, V	Eric Pascual, Nicolas Pastorelly, Alan Redmond, Bruno Fies (CSTB)	N.A.	María Pérez (Inetum BE) Oriol Escursell (ICAEN)
Section II	Eric Pascual, 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)	María Pérez (Inetum BE) Oriol Escursell (ICAEN)
Section III	Eric Pascual, Nicolas Pastorelly, Alan Redmond, Bruno Fies (CSTB) Mario De Marco, Adelfio D'Angiò, Pasquale La Pietra (Intuicy) Stoyan Danov, Eloi Gabaldon (CIMNE)	Nico Vermeir, María Pérez, Théa Gutmacher, Jarne Kerkaert (Inetum BE) Pierre Lehanneur, Riccardo De Vivo, Frederic Wauters (Helexia/Energis)	María Pérez (Inetum BE) Oriol Escursell (ICAEN)
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 (CIMNE) Stratos Keranidis, Polychronis Symeonidis (domX)	Eric Pascual, Nicolas Pastorelly, Alan Redmond, Bruno Fies (CSTB)	María Pérez (Inetum BE) Oriol Escursell (ICAEN)

Table of Contents

Table of Figures

Table of Acronyms and Definitions

Acronym	Definition
AHU	Air Handling Unit
AI	Artificial Intelligence
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
ES ECMS	Energy Conservation Measures (ECMs)
EEM	Energy efficiency measures (EEM)
EFFIG	Energy Efficiency Financial Institution Group
EPC	Energy Performance Certificate .
EPCo	Energy Performance Contract
ESCO	Energy Service Company
EUBSO	EU Building Stock Observatory (EUBSO) and national/regional Energy Performance Certification (EPC)
HVAC	Heating Ventilation and Air Conditioning
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/
Process	A process is a logical grouping of operations manipulating data : retrieval, publication, transformation, derived data extraction,… Processes can be defined in a hierarchical way, for instance to provide a higher level view of a group of more elementary actions.

RAF	Reference Architecture Framework
RES	Renewable Energy Source
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.

I. INTRODUCTION

The BIGG project aims at demonstrating the application of big data technologies and data analytic techniques for the complete buildings 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 will achieved 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 data sets); 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 initial technical specifications and preliminary design of BIGG Architecture building blocks.

This is 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. They are software units with well-defined purposes like for instance retrieving data, transforming data, analysing data or coordinating data flows.

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

The document is organized in the following manner:

- the first section presents the technical approach to ensure modularity and versatility of BIGG software components
- The second section presents the 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). In this chapter different categories of BIGG components are introduced that manage matters like ingesting data, harmonizing data, analysing and improving data, exposing data to external systems, or organizing BIGG pipelines.
- The last section will describe the planned instantiation of the BIGG reference architecture for the different BIGG business cases. This chapter will explain which BIGG components will be used by each partner for the first version of the platform implementation.

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 preliminary 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 analysing 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 flavours:

- 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, 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") able to connect to the Kafka message bus.

It must be noted that these 3 forms are agnostic with respect 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) and with the installed services and software libraries. For instance, an application can require a dependency which is already installed on the target system, but with a version not compatible with what is expected. Trying to install it without updating the dependency will result in a non-working application. Trying to update the dependency has big chances 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**[1]. In a few words, this is a form or virtualization, but without the overhead of stacking a guest OS over the host OS. The 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 flavours of distribution presented above**.

One option for supporting the concept of "component flavour" 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: This is not mandatory to code all versions of the wrapped components: involved development teams are free to choose the best technical implementations that suit the pilots’ requirements and constraints.

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

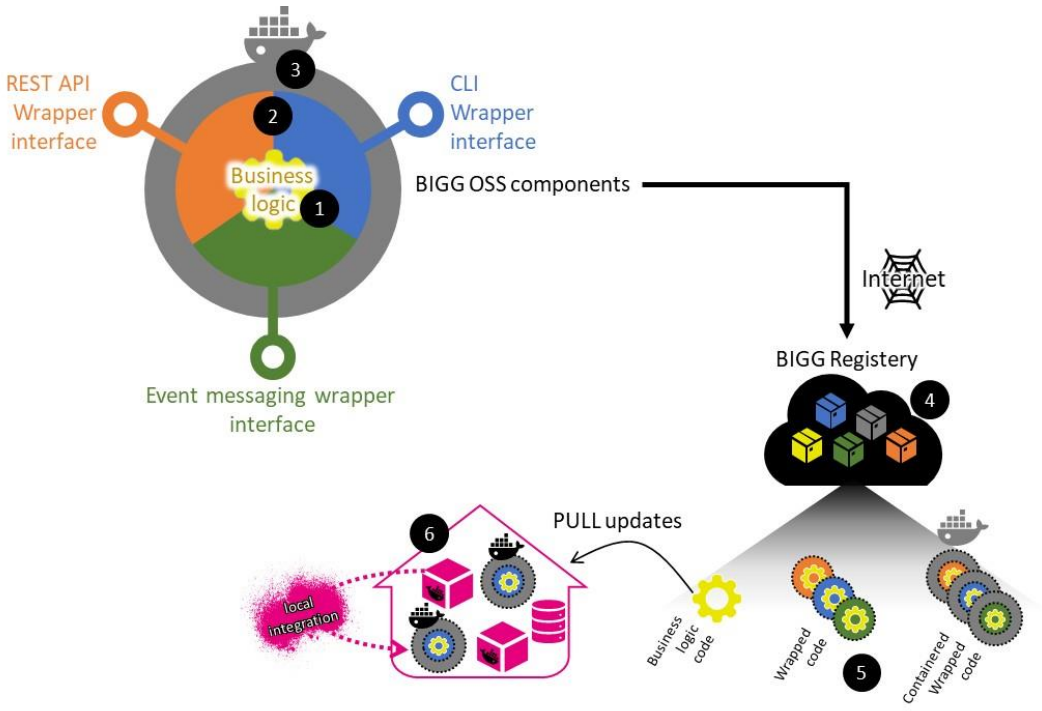

Figure 1) BIGG components flavours, distribution mechanisms and deployment

The figure above summarizes technical solutions to manage and share BIGG components among consortium partners:

1. For every business subject handled by BIGG (ex: ingesting data, harmonizing data, analysing and improving data etc. see §III.3.), business logic software codes must be created. These business logic codes hold the added value mechanisms and features that have been created by BIGG developers regarding the business matters. These “low level” codes must be shared among consortium's developers so that the code can be commonly improved, reviewed and updated in a future. These business codes usually use dedicated technologies and frameworks or libraries (ex: Python + Pandas, R + ggplot2) to offer a software solution to a business matter.
2. Then business logic code can be wrapped in different interface-implementation options depending 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 flavours 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 cases, components in different required flavours will be stored in the repository. The business logic code holds the added value processes that had 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[2] 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

They are expected to use:

- positional arguments by default for identifying data sets (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

[2] 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.

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.

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 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 data sets, its execution must be handled by a HTTP POST request which body contains the parameters encoded as `multipart/form-data`[3]. If no data set is to be passed, the `application/x-www-form-urlencoded`[4] 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 burden on the end-user's side and does not bring a noticeable benefit to the performance side, considering the generally small size of the involved inputs.

When the process is expected to return a data set, it must be the content of the response body and the `Content-Type` header[5] 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[6] 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.

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

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

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

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

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
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.

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

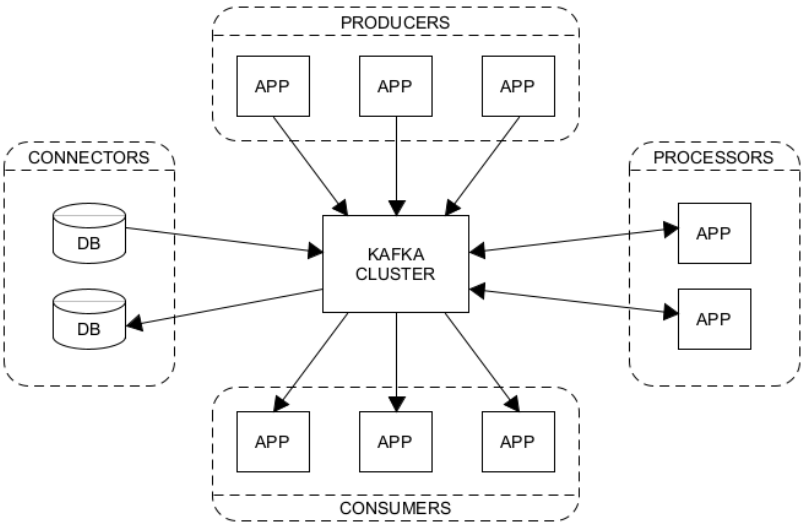

Figure 2) Kafka ecosystem

So, a “cluster” is 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 to start 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 made of all the messages the producers send to and the consumers subscribe to.
- **Partition**
  A topic can contain a huge amount of data. Storing and processing such quantity of data could be scaled dividing all the messages sent to a topic between its “partitions”. As an example, a Kafka cluster could 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 could be used to divide message between the partitions: range, hashing, round-robin, etc.

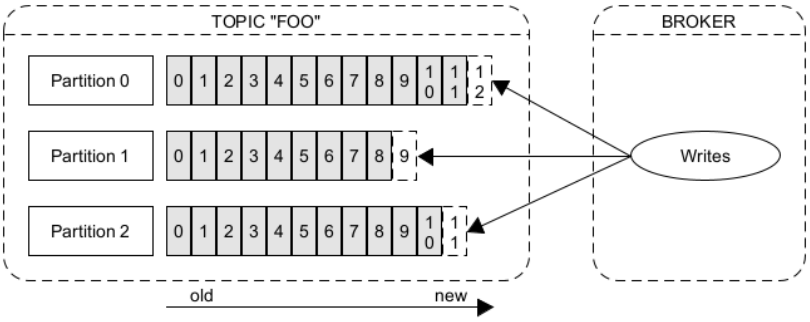

Figure 3) Kafka partitioning

- **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.

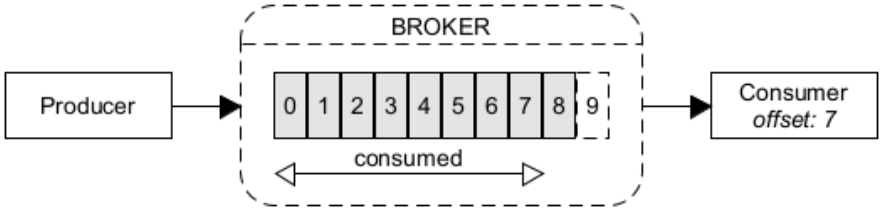

Figure 4) Kafka offsets management

- **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.

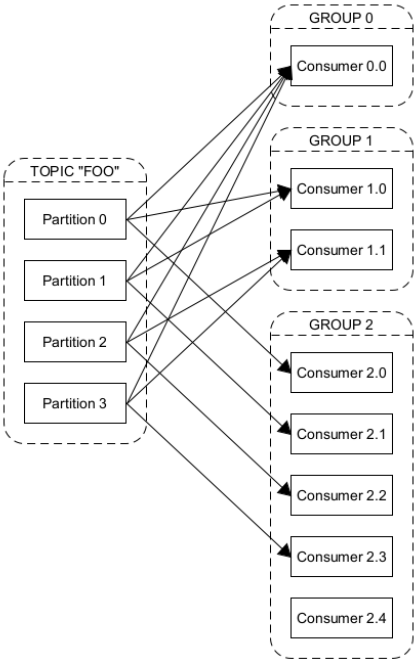

Figure 5) Kafka consumer groups

Moreover, Kafka adopt a rebalancing policy when the partitions are more than the consumers in a group:

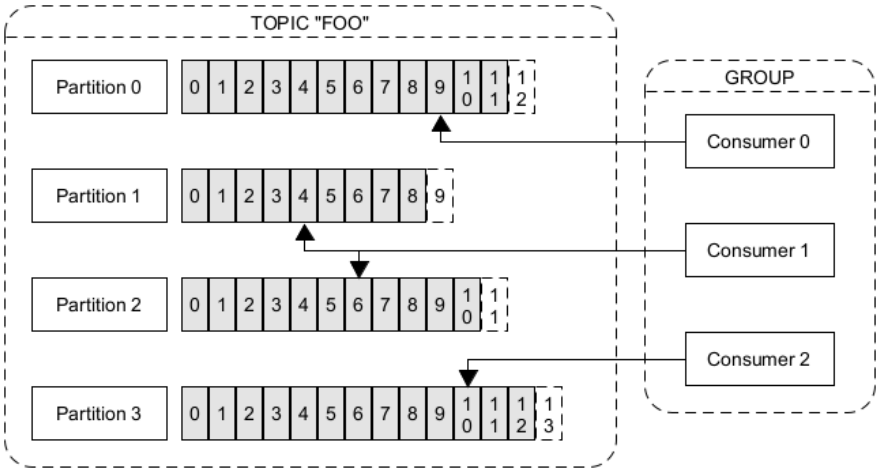

Figure 6) Kafka rebalancing policy

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`[7], that 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
```

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

```
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);


// 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. The following table 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

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)[8]. 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**

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

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.

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.

Here there is an example of different BIGG components integration via a custom integration mechanism:

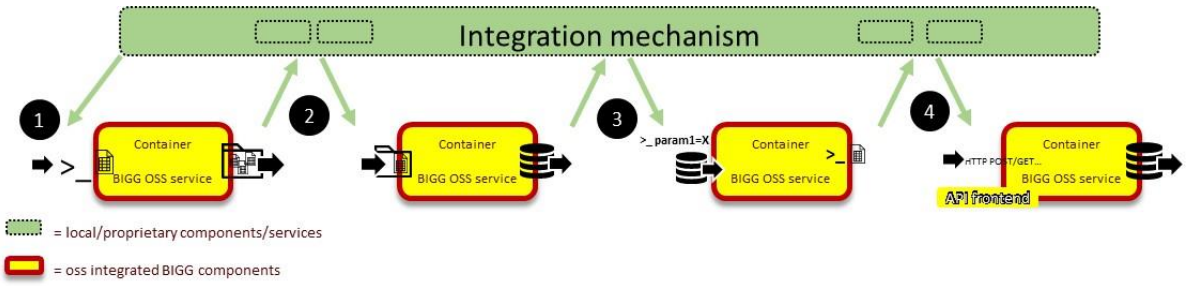

Figure 7) Example of a custom BIGG components integration solution

1. A custom integration mechanism is calling 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 of the next component in the pipeline along with connection parameter to a shared database. The second component is using the shared database to store its results after processing files provided by the first component in the shared folder.
3. The integration mechanism is triggering 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 is 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 (ex: a simple

script to chain several components) to the more complex one using an event stream messaging system and the RAF described in chapter III.

For instance, some data scientists could use some of the BIGG dockerized components in direct local integration with tools like Jupyter Notebook[9] or Matlab[10] via CLI calls.

[9] https://jupyter.org/

[10] 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"[11] and, despite more and more V's are adding over time (arriving up to ten), the standard definition counts 5 V:

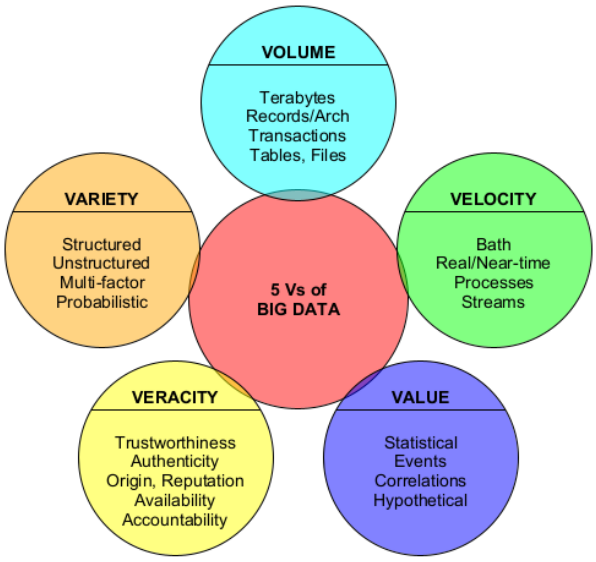

Figure 8) Big Data 5 Vs

1) Volume

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.

2) Velocity

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 best described as 'torrential' and 'unprecedented'. Now, this data should

[11] Laney, D. (2001), "3-D Data Management: Controlling Data Volume, Velocity and Variety"

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

3) 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.

4) 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 data sets, it is important that the validity of the data is checked before proceeding for processing.

5) 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 doesn't conform with the 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 percent 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:

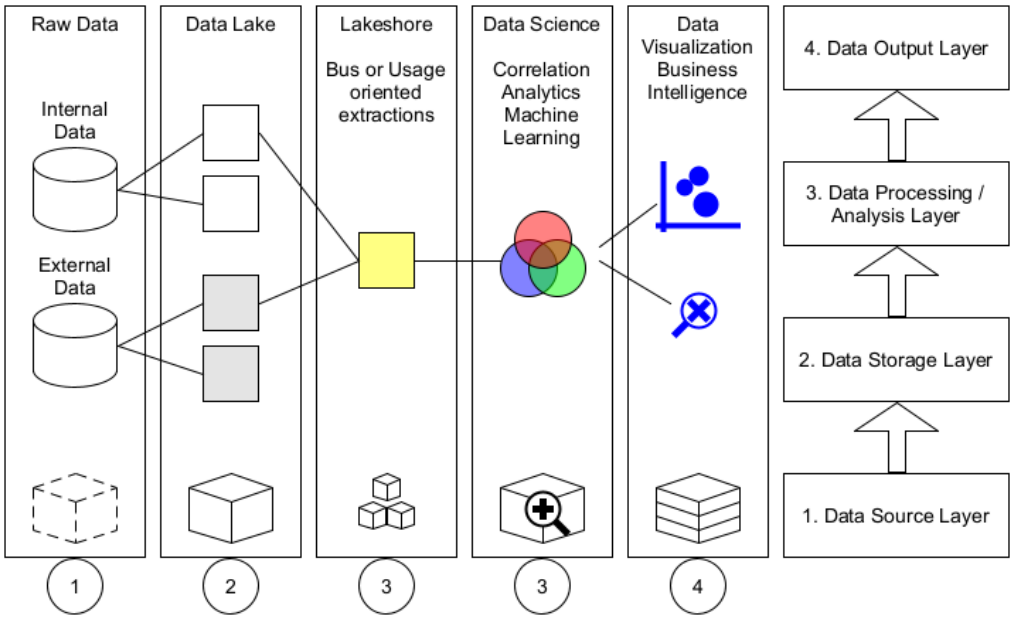

Figure 9) Big Data architecture layers

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, in order 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 persisted. Usually, a generic purpose NoSQL repository is used to store agnostically 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 storage is called “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 has to be enriched with labels and metadata.

3) Data Processing / Analysis Layer

When you want to use the data you have stored to 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, 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 taking action 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[12] 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:

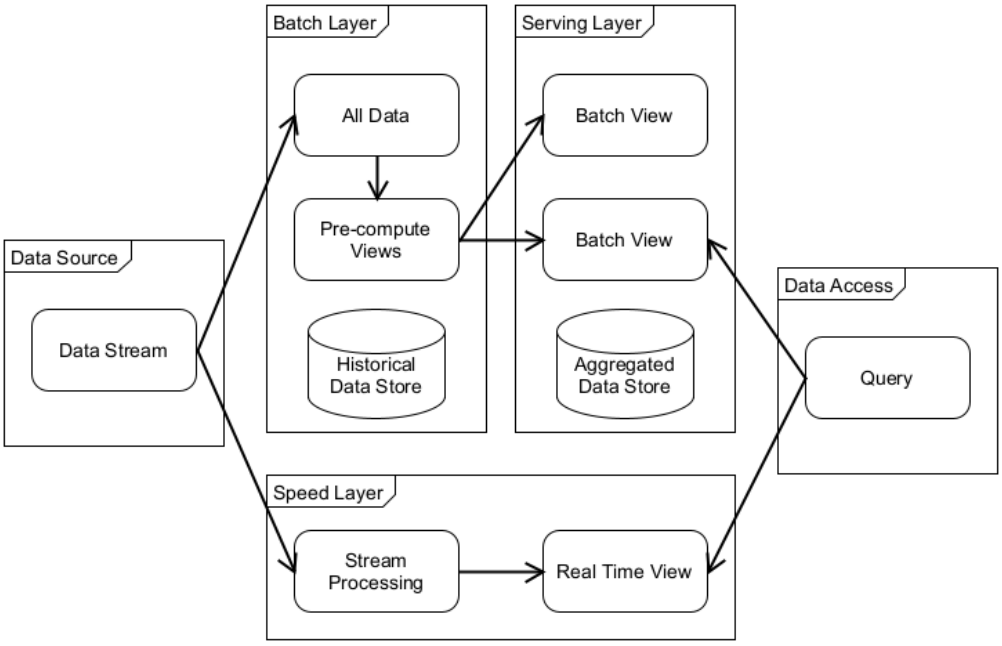

Figure 10) Big Data Lambda architecture

An alternative generic reference architecture was successively designed by Jay Kreps. The Kappa Architecture[13] 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 the last engine in real time and to the replacement of the data stored in the service

[12] http://lambda-architecture.net/

[13] https://www.oreilly.com/radar/questioning-the-lambda-architecture/

http://milinda.pathirage.org/kappa-architecture.com/

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:

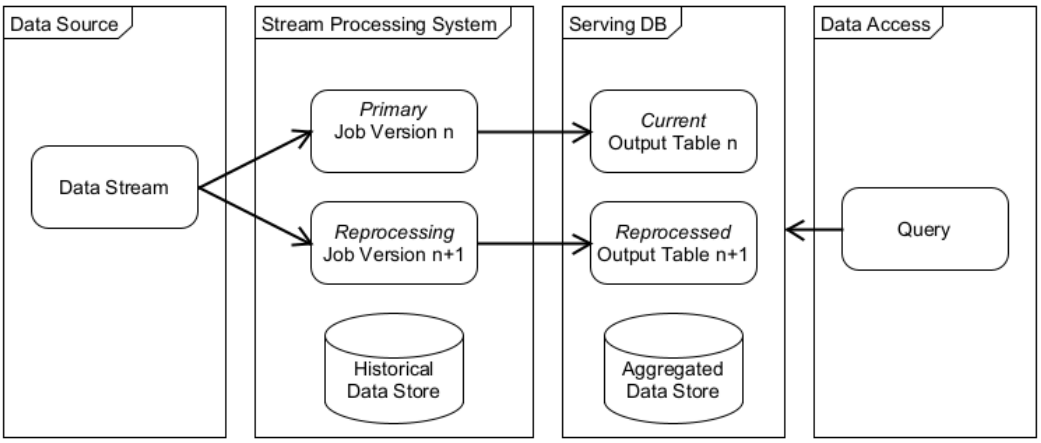

Figure 11) Big Data Kappa architecture

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.

III.2. RAF overview

III.2.1. Message Broker

Whatever the generic reference architecture to be used between Lambda and 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 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

In the use of this tool, 3 fundamental service archetypes can be identified:

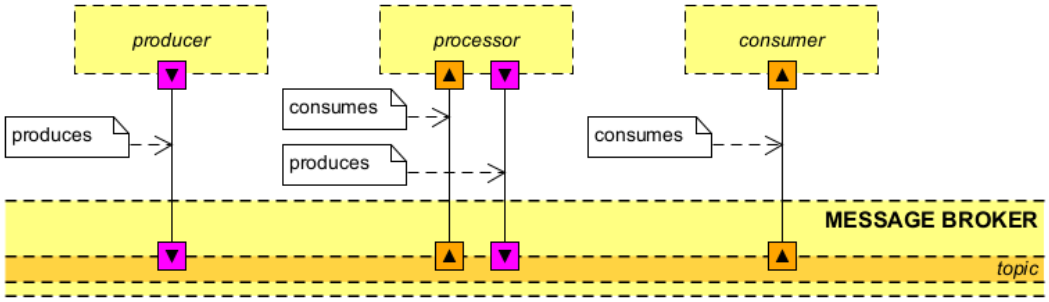

Figure 12) 3 fundamental service archetypes for a Big Data architecture

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.2.2. RAF description

III.2.2.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) could have the 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, harmoniser, 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.2.2.b. Design

The following diagram shows the BIGG reference architecture:

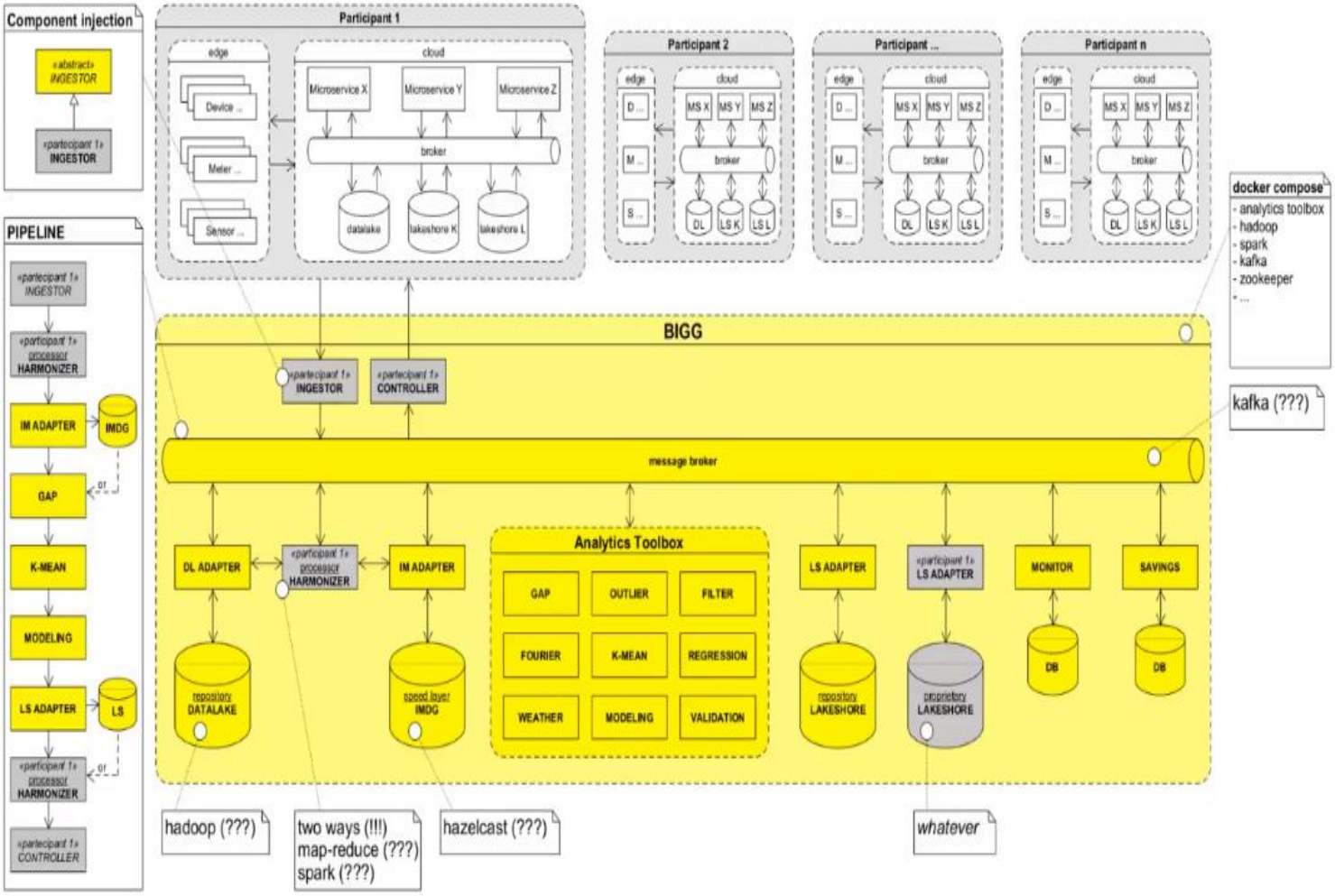

Figure 13) Overview of BIGG reference architecture

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.

Harmoniser (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 persist all the messages it retrieves from the broker into the datalake. It is also used to retrieve the same messages from the datalake. 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 transaction per second. Redis and Hazelcast are two tools that would 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 datalake 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 timeseries 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.2.2.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 Harmonized) 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” could 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**[14], 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.

[14] aka “AI Toolbox”

III.3. Components

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.

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 as less computational resources as possible.
- **Scalable and elastic:** allowing more instance of the micro-service to run in the system on different nodes to (possibly linearly) increase the throughput; making 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 Harmoniser microservice:

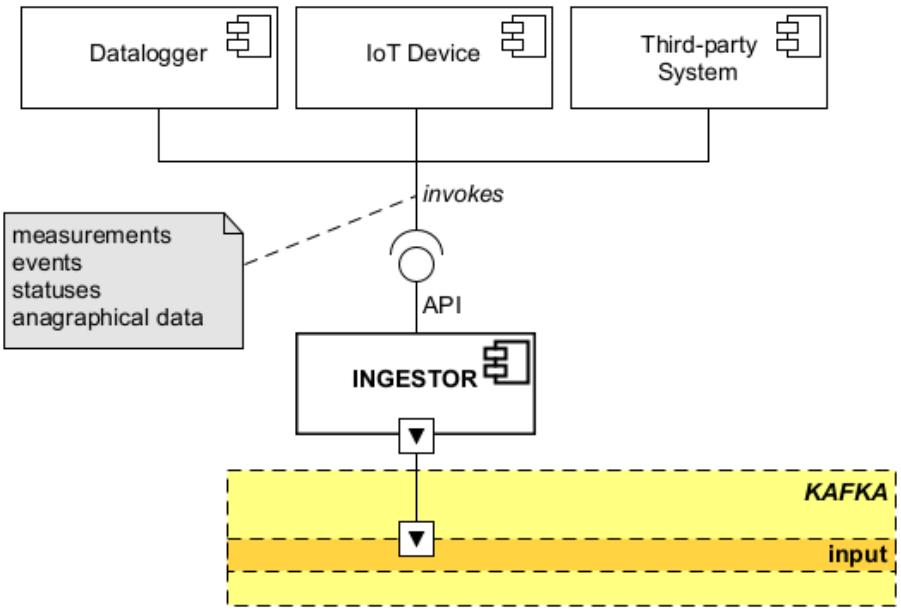

Figure 14) Ingestor artefact

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's a text string that must be enough to 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

At present day, 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”)

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 a metadata with one of these names:

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

If such a 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 does not currently need a GUI, thus it has none.

III.3.2. Harmonization components

III.3.2.a. Roadmap overview

The Harmoniser has the objective to transform the data input from external systems into the BIGG-designed harmonised data format that will make them compatible with the Analytics Toolbox services. The Harmoniser component will be developed gradually throughout the project, initially creating the necessary artifacts and testing them in the first phase, and further evolving them as an integrated components at a second stage.

III.3.2.a.1. Harmoniser component artifacts (V1)

In V1 the initial artifacts for development of the component will be produced and tested. This stage covers the research and initial development of the BIGG Standard Data Model 4 Buildings, the producing of initial Mapping Template, and the mapping of the data sources supporting the Business Cases to it. These are developed in an iterative process in order to ensure the necessary data fields and data relations are in place for each Business Case, reveal and document the exact content of each data source in terms of variables and relations, and perform the initial semantic matching to the BIGG Data Model. These artifacts will be developed in a way ensuring further extension and connection to new data sources in the future, and will also provide the necessary understanding for the implementation of the automated operation of the Harmoniser envisaged for the second stage.

The artifacts developed in V1 are the following:

- BIGG Standard Data Model 4 Buildings, consisting of:
  - UML diagram
  - Data fields definition and description
  - Enumerator taxonomies' definition and description
- Mapping Template
- Individual mapping of each data source over the Mapping Template.

At this stage these artifacts are developed by using general purpose tools, such as Excel, draw.io, etc. They are meant to be agnostic of the technologies, structures and formats in which they will be implemented in the Harmoniser and the external systems. Proposed artifacts are open enough to enable specific technical implementations for harmonisation mechanisms, according to the prescriptions of the overall BIGG RAF.

At operational level, in order to ensure the overall testing of the BIGG RAF in V1, the Harmoniser component artifacts will be used as a base for implementing customised transformation of the raw data sources to the harmonised data structure. Dedicated custom harmonisers will be implemented supporting each BIGG Business Cases. The development of the communication between the Ingestor and Harmoniser will then be demonstrated and implemented from the start of the project. The schema below represents the logic of this first implementation:

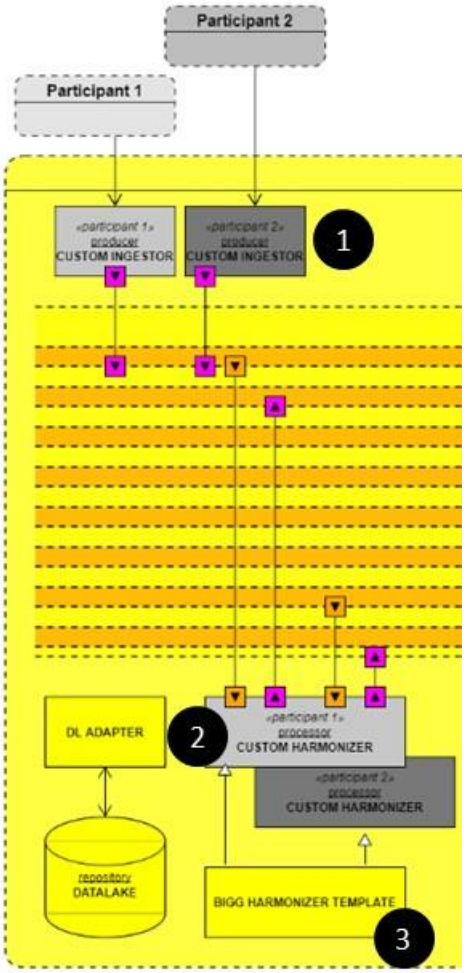

Figure 15) V1 version of the architecture of the ingestion/harmonization process

1. Every participant will locally set the required ingestors up in order to feed into the system implementing the BIGG RAF their local datasets which have a specific format related to their business case.
2. The raw data collection event created by the custom ingestor will be pushed over the message bus and intercepted by a custom harmoniser which will transform participant's data into a normalised data. The custom harmoniser will be the first implementation of the harmonisation artefacts provided by WP4 work.
3. From a technical point of view, a boilerplate code can be provided that describe generic harmoniser behaviour so the BIGG consortium's developers can use it as a base to implement their custom harmoniser.

III.3.2.a.2. Harmoniser component (V2)

Version two of the harmoniser component pushes the genericity further. A harmoniser will be implemented as a containerised component that can be installed on cloud or integrated into a local system and will be able to receive input data messages and transform them into a common harmonised data message format directed to any system component that can benefit from it.

The Harmoniser will implement a "Mapper" structure that describes the transformation from the original to the harmonised format and will be operated as a microservice.

The Mapper will be created from a core module named “Transformer”. As an input the Transformer 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 hamoniser will enable the translation of the incoming messages from the mapped data sources emitted from the Ingestors into the BIGG common message format.

The Harmoniser 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 Harmoniser 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 as less computational resources as possible.
4. **Elastic and scalable**: allowing more instances of Processor to run in the system on different nodes to increase (possibly linearly) the throughput; making easy to add, remove or move instances in the system.
5. **Robust**: coping with error and with erroneous inputs without compromise the operation of the system.

The following schema depicts the overview of a workflow involving the harmoniser V2 component:

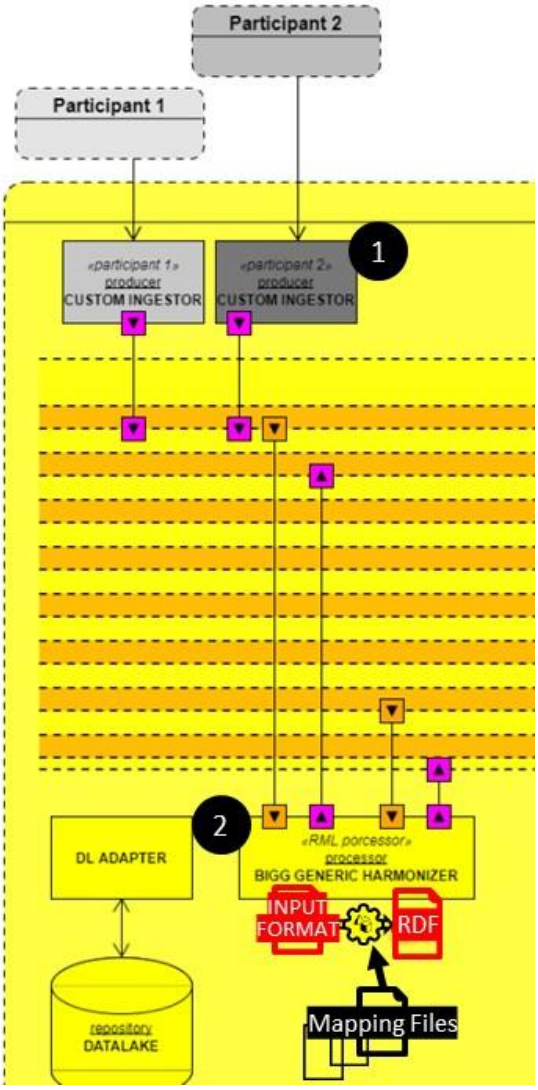

Figure 16) V2 version of the architecture of the ingestion/harmonization process

1. Custom ingestors will get connected to external participant's sources to feed participant's raw data into the BIGG system
2. The harmoniser V2 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.b. Architecture

The Harmoniser 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 Harmoniser 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 Harmoniser delivers data to them, in a loosely-coupled way, based on Kafka topics.

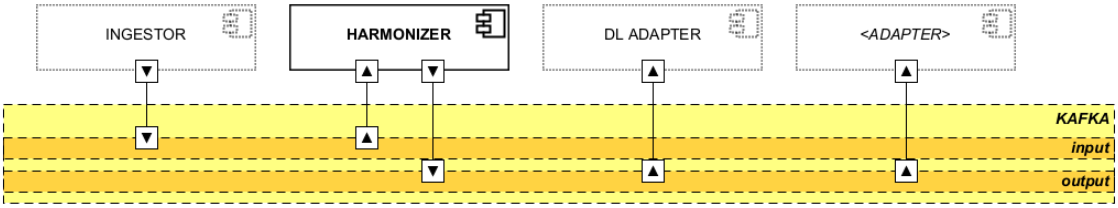

Figure 17) Harmoniser artefact

III.3.2.c. 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's 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 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.3.2.d. Database

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

III.3.2.e. Configuration

The Harmoniser has to 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.f. Implementation

The implementation of the Harmoniser 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:

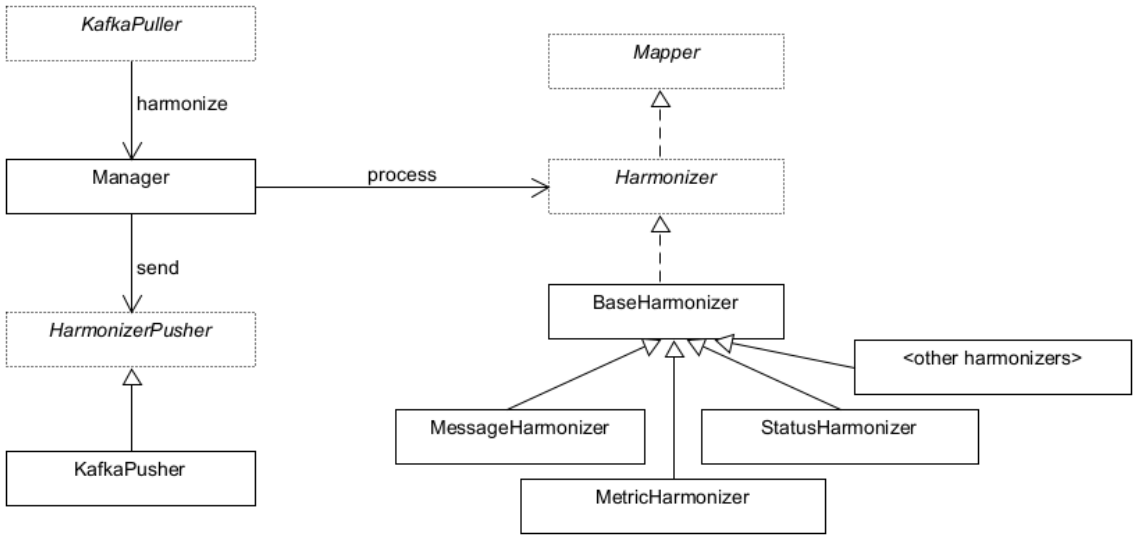

Figure 18) Main components of a BIGG harmoniser

- **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 “Harmoniser” implementation (I.e. MessageHarmoniser, MetricHarmoniser, etc.) and, once received the corresponding harmonized message, propagating it to the “pusher”
- **Harmoniser**
  the “harmoniser” interface will be an extension of the “mapper” interface described above. Every partner will develop the set of harmonisers 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.g. GUI

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

III.3.3. Data processing/analysis components

This section will present the data processing and analysis components that will be created in the BIGG project. These components are presented here but are more deeply detailed and documented on biggproject/biggdocs GitHub[15].

The BIGG GitHub has been separated in three separate sections:

1. BiggDocs : Where all the functions are defined and described in details
2. BiggPy: Where the functions are implemented in Python language
3. BiggR: 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.

At the time of writing this document only a snapshot of these components’ definitions can be provided. It is recommended to directly go to the GitHub repository in order to find more detailed and more up to date descriptions of the WP5 toolbox components.

III.3.3.a. Data preparation

III.3.3.a.1. Data Preparation / Time Stamps Alignment

a. detect_time_step

The function infers, i.e. automatically deduces from the input data, the minimum time step (frequency) that can be used for the input time series.

b. align_time_grid

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

The function aligns the frequency of the input time series with the output frequency given as an argument using the specified aggregation function.

c. clean_ts_integrate

The function converts a cumulative (counter) or onChange (delta) measurement to instantaneous.

III.3.3.a.2. Data Preparation / Outlier Detection

a. detect_ts_min_max_outliers

Detect elements of the time series outside a minimum and maximum range.

b. detect_ts_zscore_outliers

Detects elements of the time series out of a Z-score threshold, applied on the whole time series or a rolling window of predefined width.

c. detect_ts_calendar_model_outliers

Detects elements of the time series out of a confidence threshold based on linear model of the calendar variables (month, weekday, hour).

d. plot_outliers

This function prints out the outliers detected in the former 4 functions.

e. detect_static_min_max_outliers

Detect which numerical elements are outside the min-max range.

f. detect_static_reg_exp

Detects which string element satisfy the regular expression.

III.3.3.a.3. Data Preparation / Missing Data Management

a. fill_ts_na

This function sets values to Not Available (NA) elements of a time series, based on the outliers estimation made the functions implemented in Outlier Detection module block of this library.

III.3.3.b. Data Transformation

III.3.3.b.1. Data Transformation / Profiling

a. clustering_dlc

Cluster similar daily load curves based on the load curves itself, calendar variables and outdoor temperature.

b. classification_dlc

Classify daily load curves based on the outputs of a clustering and a new set of data.

c. weekly_profile_detection

The function returns the weekly profile of the input time series.

d. yearly_profile_detection

The function returns the yearly profile of the input time series.

e. trend_estimation

This function detects general trends in a given time series. A trend would be defined as a long time tendency agnostic of short-term variations. (to be developed in the AI Toolbox v2)

III.3.3.b.2. Data Transformation / Weather

a. Degree_Days

Calculate the degree-days with a desired output frequency and considering cooling or heating mode.

b. Degree_Raw

Calculate the difference between outdoor temperature and a base temperature, without considering the frequency of the original data.

III.3.3.b.3. Data Transformation / Autoregressive processes

a. lag_components

This function time shifts a set of features in order to be used in the training and prediction of the models. It is an important step for the multi-step prediction of Autoregressive models, where the estimated output is directly used in the subsequent predictions.

b. lpf_ts

This function computes the first-order low pass filter for smoothing a time series.

c. get_lpf_smoothing_time_scale

Physical transformation of the smoothing time scale parameter to consider no affectance to the output variable after a known number of hours.

III.3.3.b.4. Data Transformation / Calendar

a. calendar_components

Decomposes the time in date, day of the year, day of the week, day of the weekend, working day, non-working day, season, month, hour, minute, ...

III.3.3.b.5. Data Transformation / Fourier Series

a. fs_components

Obtains the components of the Fourier Series, in sine-cosine form. It is useful for linearising the relationship of a seasonal input time series (e.g. solar azimuth, solar elevation, calendar features, ...) to some output (energy consumption, indoor temperatures, ...). It basically decomposes a cyclic time series into a set of sine-cosine components that are used as inputs for the modelling of some output, each of the components linearly depends to the output.

III.3.3.c. Modelling

III.3.3.c.1. Data Modelling / Cross Validation

a. k_fold

This function is already part of an existing analytics package of Python named Sci-kit Learn. It provides train/test indices to split data in train/test sets. Split dataset into k consecutive folds (without shuffling by default).

Each fold is then used once as a validation while the k - 1 remaining folds form the training set. (Source: Scikit-Learn.org)

b. time_series_split

This function is already part of an existing analytics package of Python named Sci-kit Learn:

It provides train/test indices to split time series data samples that are observed at fixed time intervals, in train/test sets. In each split, test indices must be higher than before, and thus shuffling in cross validator is inappropriate.

This cross-validation object is a variation of KFold. In the kth split, it returns first k folds as train set and the (k+1)th fold as test set.

Note that unlike standard cross-validation methods, successive training sets are supersets of those that come before them.(Source: Scikit Learn)

c. train_test_split

This function is already part of an existing analytics package of Python named Sci-kit Learn:

It Split arrays or matrices into random train and test subsets

Quick utility that wraps input validation and next(ShuffleSplit().split(X, y)) and application to input data into a single call for splitting (and optionally subsampling) data in a oneliner. (Source: Scikit Leaarn)

III.3.3.c.2. Data Modelling / Model Assessment

a. Cross_validate

This function is already part of an existing analytics package of Python named Sci-kit Learn: This function Evaluate metric(s) by cross-validation and also record fit/score times.Data Modelling / Model Identification.

(Source: Scikit Learn)

b. Cross_val_score

*This function is already part of an existing analytics package of Python named Sci-kit Learn.*It Evaluate a score by cross-validation.

(Source: Scikit-Learn)

c. evaluate_model_cv_with_tuning

This function performs a nested cross-validation (double cross-validation), which includes an internal hyper-parameter tuning, to reduce the bias when combining the two tasks of model selection and generalization error estimation. However, the purpose of this function is not to

select the best model instance of a model family but instead to provide a less biased estimate of a tuned model's performance on the dataset.

III.3.3.c.3. Data Modelling / Model Identification

a. identify_best_model

This function implements a complete generalized pipeline for supervised learning to find the best model among different model families, each one associated with a specific parameter grid, given an input time series and a scoring function.

III.3.3.c.4. Data Modelling / Model Persistence and prediction

a. serialize_model

This procedure serializes a model, according to the specified file format and saves it on the file system following a specific convention (tbd).

b. deserialize_and_predict

This function deserializes a model, inferring the file format from the file name, applies the model on the X_data and returns the predicted values in the form of a time series.

c. test_stationarity_acf_pacf

This function tests the stationarity and plot the autocorrelation and partial autocorrelation of the time series.

d. split_train_test

This function splits the time series into train and test datasets at any given data point.

e. param_tuning_sarimax

This function performs an exhaustive search on all the parameters of the parameter grid defined and identifies the best parameter set for a sarimax model, given a scoring function.

f. param_tuning_prophet

This function performs a search on all the parameters of the parameter grid defined and identifies the best parameter set for a prophet model, given a MAPE scoring.

g. fit_sarimax

This function trains and fits a SARIMAX model

h. test_sarimax

This function gets the prediction of the sarimax model.

i. fit_prophet

This function trains and fits a PROPHET model

j. test_prophet

This function gets the prediction of the prophet model.

k. evaluate_forecast

This function calculates evaluation metrics for the prediction.

l. schedule_optimizer

This function gives alternative schedules for households to answer to the flexibility request.

III.3.3.d. Reinforcement Learning thermal model

Description of the thermal model of the building that can be used to estimate future gas flexibility. This is a sequential model to estimate the next time steps room temperature and gas consumption given the boiler and room temperate set points.

III.3.3.d.1. Thermal Model / Dynamics

a. RoomT_next

This function calculates the room temperature for next time step. This is based on the space heating model.

b. BuildingT_next

This function calculates the building temperature (temperature of the thermal mass of the building) for next time step. This is based on the space heating model.

c. BoilerInletT_next

This function calculates the boiler inlet temperature for next time step. This is based on the space heating model.

d. BoilerOutletT_next

This function calculates the boiler outlet temperature for next time step. This is based on a decay/growth model for the boiler temperature, where a1 and a2 are the variables that control the rate of change of boiler temperature.

e. Gas_modulation

This function calculates the Gas modulation at the next time step. This is based on the boiler model. b1 and b2 are parameters used for the internal model of the boiler

III.3.3.d.2. Thermal Model / PhysicsCell

a. PhyCell

Description function for the Class **PhyCell** for space heating.

b. PhyCell. forward()

Forward method for the **PhyCell** class.

c. PhyCell. set_param()

Forward method for the PhyCell class.

d. PhyCell. get_param()

Function that returns the dictionary of the parameters of the current instance of the PhyCell object.

e. PhyCell. set_param_grad()

The forward method for the PhyCell class.

f. PhyCell. param_loss()

Function to calculate the loss for the parameters of the PhyCell. This loss is calculated using the weight loss function defined in the training section of Flexibility Identification.

III.3.3.d.3. Thermal Model / DenseNet

a. DenseNet

A function to create a DenseNet neural network with given layers.

III.3.3.d.4. Thermal Model / Model

a. thermalmodel

Thermalmodel class, can be used to create a thermalmodel object. This object can be trained, validated and tested using data

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 in order to pass them to the end-users or external systems.

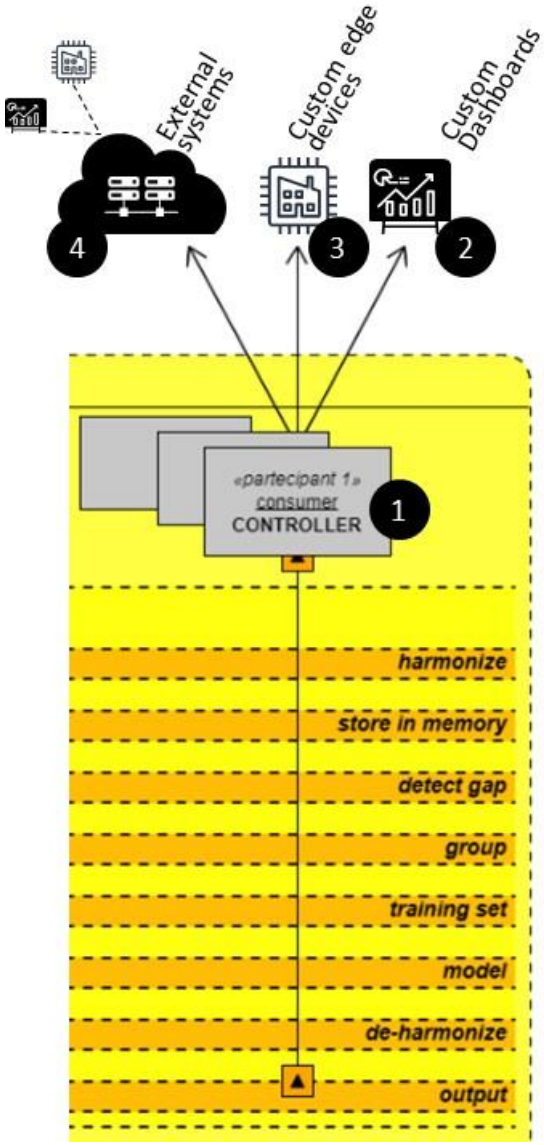

Figure 19) Output layer consumer components positioning in the RAF architecture

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 to provide this information to external systems. In order to do that, 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 (ex: 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 in order to manage complex scenarios where custom dashboards are created or building devices are managed. In that case a custom controller must be created which 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 like 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

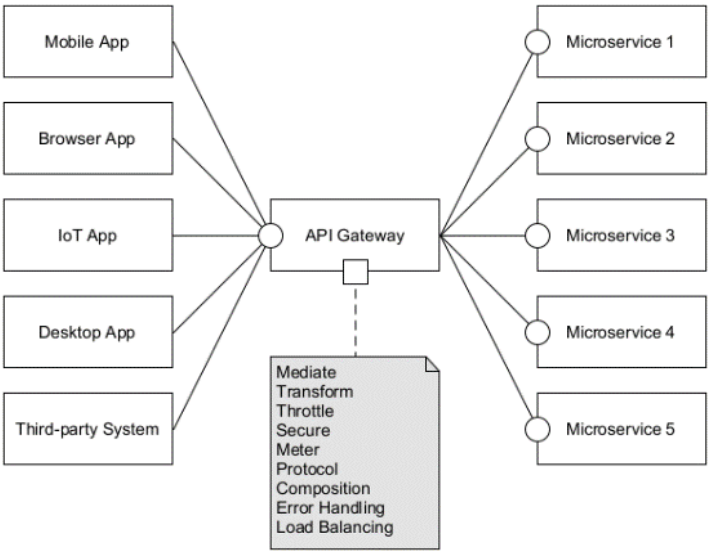

Figure 20) API gateway component

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 scene, 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 could be based on ZooKeeper[16], an open-source project which 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 name space of data registers (these registers are called znodes), much like a file system. It is a necessary component for Kafka, and we could leverage this dependency using some of its functionalities. On top of ZooKeeper, to facilitate the usage of its API's, a tool like Curator[17] could be used. The following schema depicts the usage of Curator library:

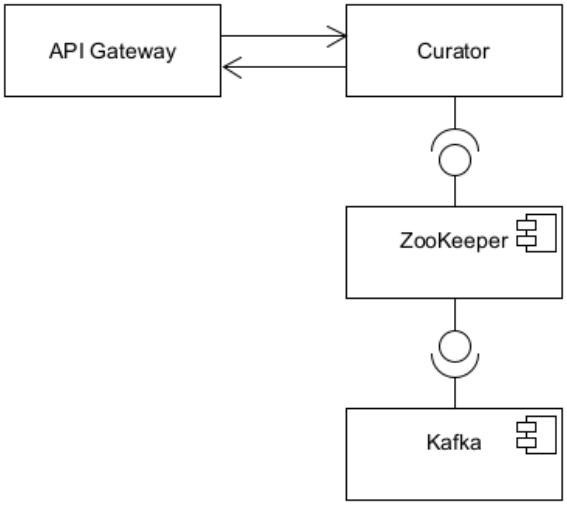

Figure 21) API gateway component technical implementation

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

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

// instantiate and start a client pointing to ZooKeeper instance and
```

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

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

```
// 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 could 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 could 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

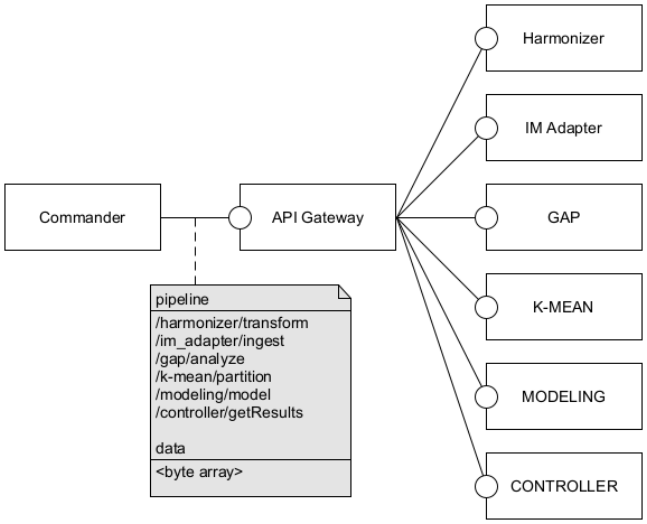

Figure 22) Commander component

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, each dancer knows exactly his movements and how he must interact to the other dancers, in a choreographed architecture each microservice knows its inputs, its outputs and how to communicate with the other microservices to enact 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). The following diagram shows the implementation of a pipeline using proper topic names and configuring accordingly the microservices:

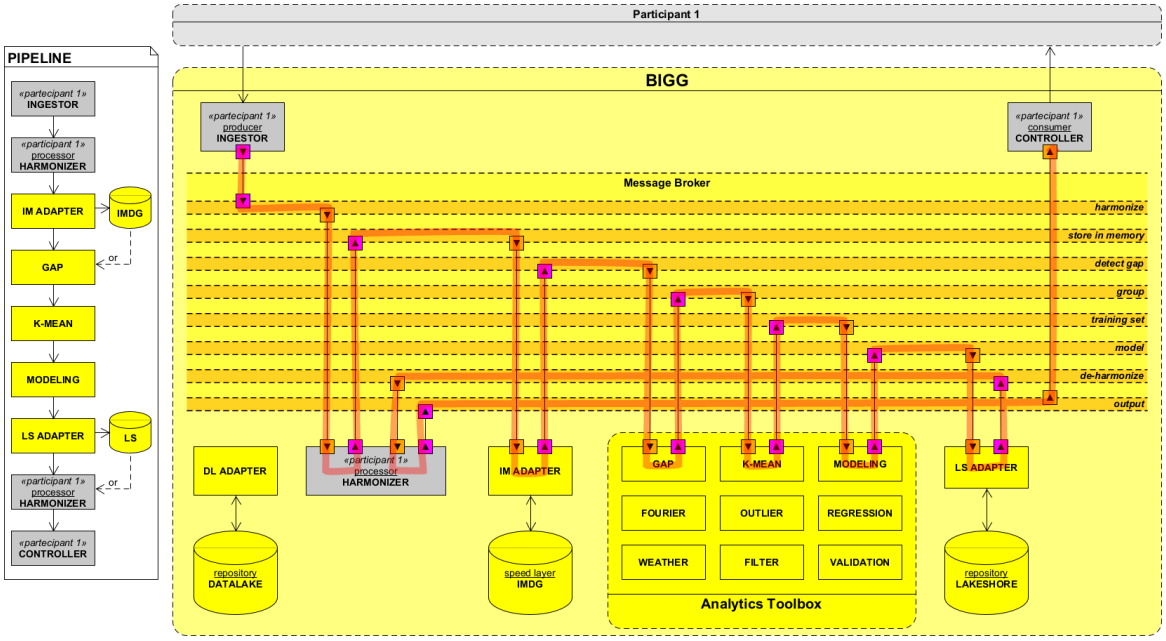

Figure 23) Example of a choreographed flow in the BIGG RAF

III.3.5.c. Components of an orchestrated architecture

If, on one hand, a choreographed architecture suits well 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 following diagram shows the components that must be present in this scenario:

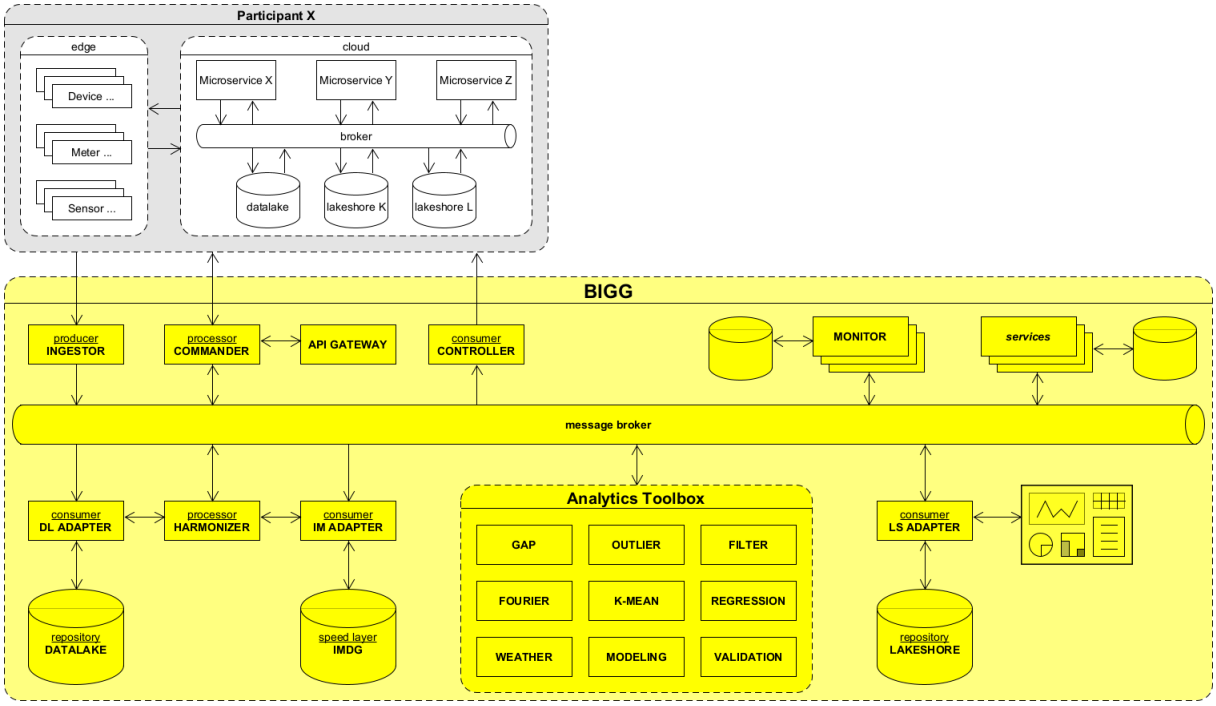

Figure 24) Example of an orchestrated flow in the BIGG RAF

IV. MAPPING OF FRAMEWORK COMPONENTS WITH BUSINESS CASES TECHNICAL ARCHITECTURES

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 are for instance:

- 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 […]*)

This section starts by presenting the role of the overall BIGG KPI dashboard shared by pilots, meant to evaluate the BIGG platform usage & performance.

Then, for every BIGG business case, the flow of information among the different BIGG components in the BIGG RAF is described for a pragmatic implementation in version 1. These flows will be updated for the version 2 of the implementation of the BIGG platform.

IV.1. Role of the BIGG KPI dashboard

The BIGG KPI dashboard will serve as a monitoring tool to provide a general overview of pilot status and progress. The platform will be centrally located and fetch data from distributed pilot sites, through remote data sources or through an API and local storage of data.

The dashboard will be built using the Grafana[18] framework, which is easily deployed, customized and provides a generic set of visualization panels and a multitude of data source connectors out of the box capable of fetching data from various types of data sources such as PostgresDB, MysqlDB, InfluxDB, OpenTSDB, CSV etc.

The dashboard will host data relevant to the Business and Use cases, show general metrics of the data provided to the BIGG platform and the KPIs that were decided in WP6 for each Use Case. Additionally, target goals set by the pilots will be visualised to better monitor the progress of each pilot.

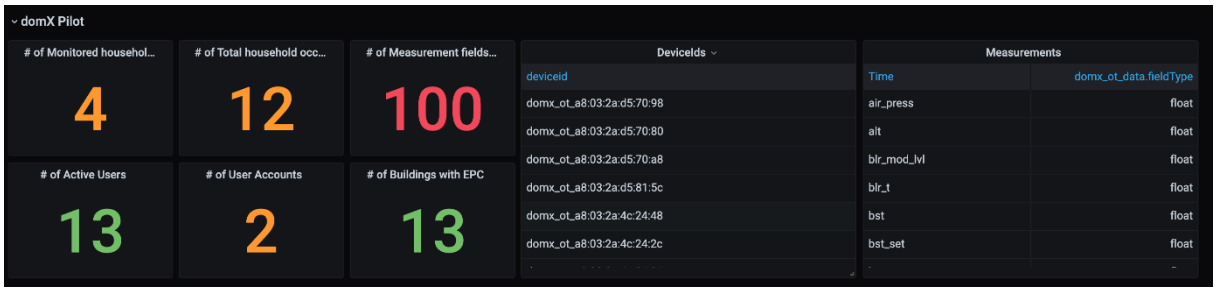

Figure 25) BIGG PKI dashboard user interface prefiguration

[18] https://grafana.com/

IV.2. Business cases #1, #2 and #3 - Case Study Area: Catalonia (Spain)

These business cases will implement service modules from the BIGG analytics toolbox for UC 1 to 7, gathering and harmonizing data from different sources (utilities, energy management systems, existing databases) to the internal standard data model of the BIGG architecture. The ingestion and harmonization process will be managed by a Kafka message broker. At least for the first version the harmonisers will be developed ad hoc for each data source, these components will be opened as OSS to be used for other BIGG users.

The use cases included in these 3 Business cases (1 to 3) will use most libraries and analytic components developed in BIGG. These components will be dockerized and executed through a Kubernetes orchestrator (Argo). In the following section the analytics components use in these cases are presented:

- Data preparation over harmonized data all, the type of process launched depends on each resource. The threads that can be launched are:
    - Time stamp alignment
    - Outlier detection
    - Missing data management
- Data transformation over prepared data in UC 1, 2, 3, 4. The type of process launched depends on each resource. The threads that can be launched are:
    - Profiling
    - Autoregressive process
    - Calendar
    - Fourier Series
- Data modelling over data transformed in UC 1, 2. The type of process launched depends on each resource. The threads that can be launched are:
    - Cross validation
    - Model Assessment
    - Model identification
    - Model persistence and prediction

The following figure shows the execution pipeline and the integration of the BIGG components for these Business Cases:

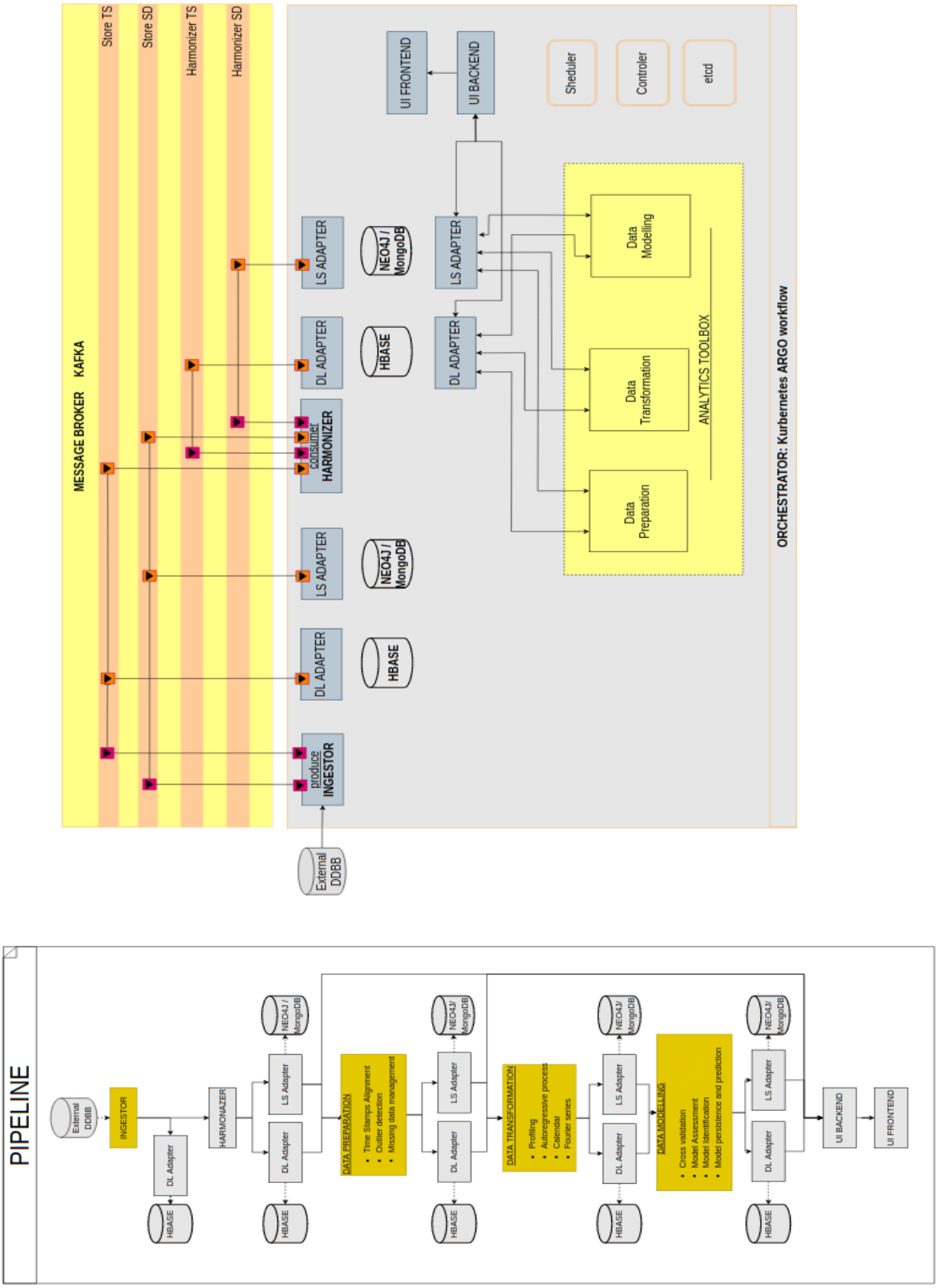

Figure 26) Business cases #1, #2 and #3 envisioned technical implementation

Here are the technical components that will be setup for V1 implementation of the BIGG RAF for these business cases:

Pipeline component	Input	Output	Implementation	Standard BIGG component
Ingestor	Query API external data sources	Raw Data	Ad hoc script developed for each source	Ad hoc for each resource. Offered in BIGG as OSS.
Harmoniser	Raw data CVS, XML, json	BIGG format harmonized data:	Ad hoc script developed for each source	Ad hoc for each resource. Offered in BIGG as OSS.
Data lake	JSON Payload format	JSON Payload format	HBASE	No, CIMNE environment
Lakeshore	RDF format	JSON Payload format	MongoDB /Neo4J	No, CIMNE environment
Analytics toolbox	Harmonised data from Lakeshore or Data Lake	Harmonised data results to Lakeshore or Data Lake	GitHub public library	Yes
Controller			Computation service	No, CIMNE environment
User Interface	API REST	Final user dashboard	Backend and frontend implemented in JavaScript	No, CINE environment. Offered in BIGG as OSS.

IV.3. Business cases #4 and #5 - Case Study Area: Athens (Greece)

From the architectural point of view, these Business Cases will be implemented in a first version, for the most part, using proprietary microservices derived from the base BIGG components. The process will use the Analytics Toolbox through the API Gateway and the Commander and will retrieve results and commands to forward to the IoT devices for optimization purposes by the Controller:

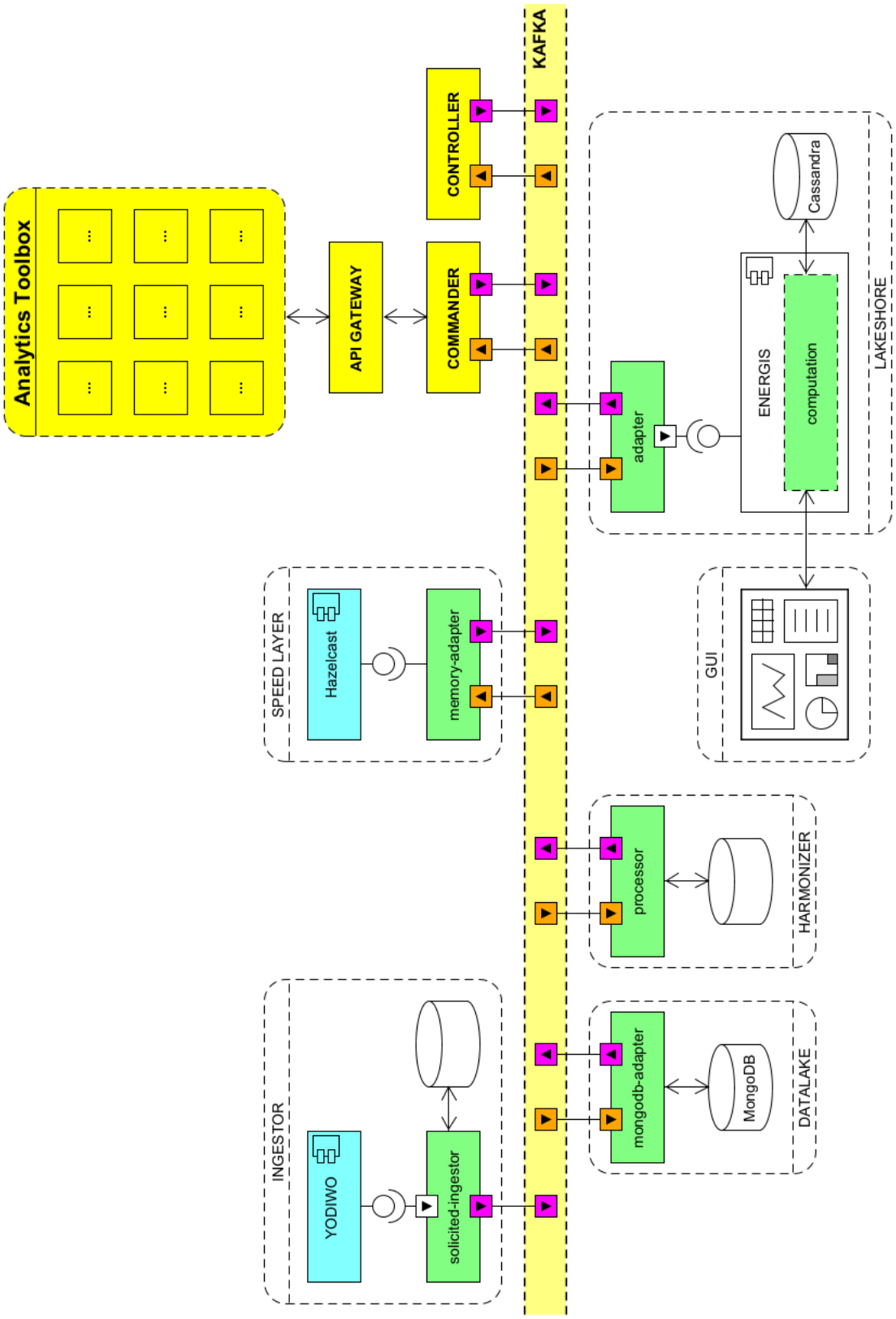

Figure 27) Business cases #4 and #5 envisioned technical implementation V1

A second, more integrated version of the architecture will use a “bigg-adapter” microservice to communicate with a shared BIGG RAF environment instantiated on the cloud:

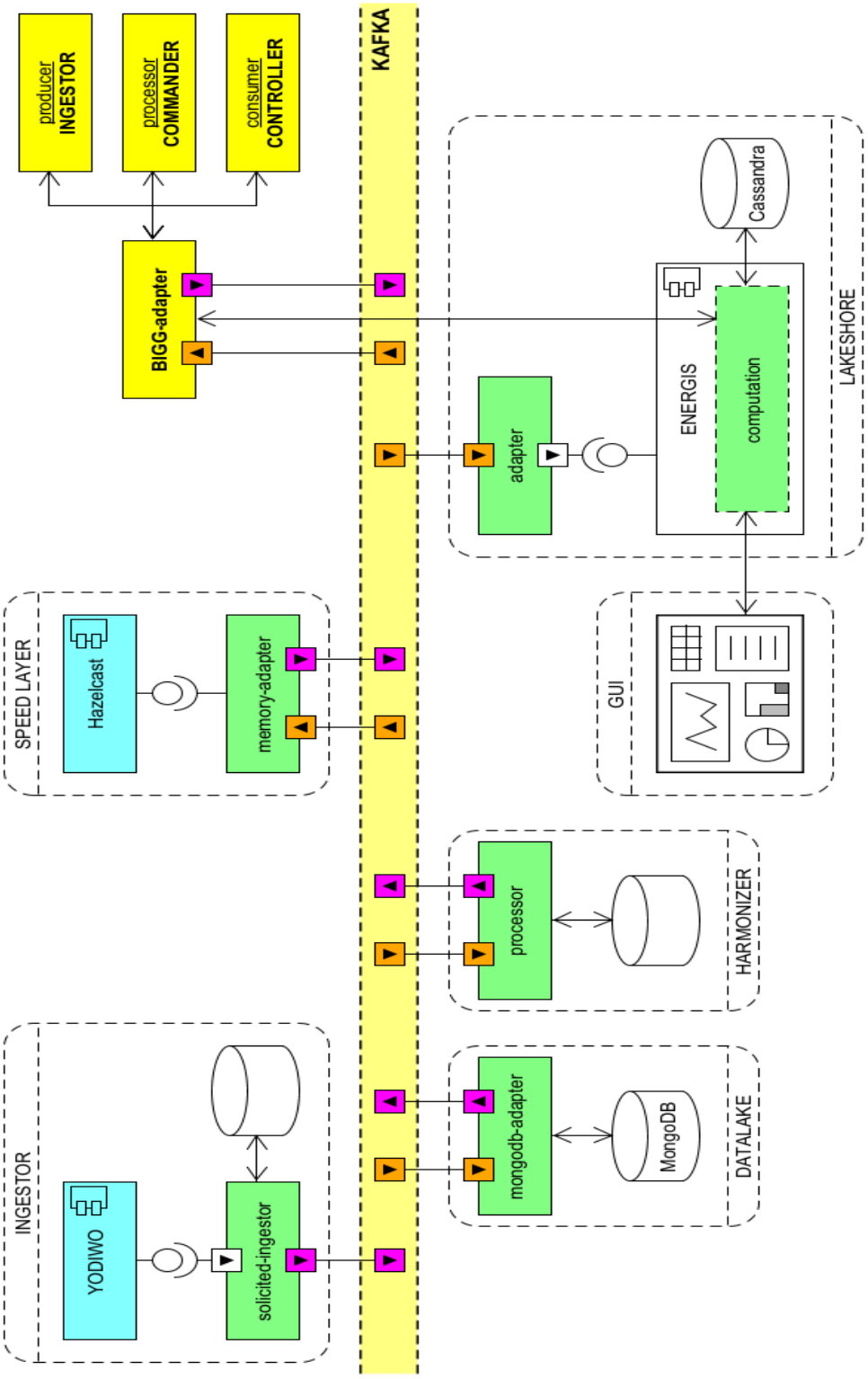

Figure 28) Business cases #4 and #5 envisioned technical implementation V2

Supporting services modules from the BIGG analytics toolbox for UC 8 to 13

- **For Business case #4**

  Monitoring service module collecting data from BMS, IoT sensors, utility invoices …, computing relevant KPIs which can be viewed in dashboards and reports. Identification of baseline models which are mathematical equations giving dependencies between consumption and influencing factors such as weather and occupancy. The model accuracy is assessed according to the IPMVP protocol.

  Contract Management service module to manage EPC and maintenance contracts. Involved stakeholders are notified when milestones are reached. Reports are generated for follow-up purposes.

Here are the technical components that will be setup for V1 implementation of the BIGG RAF for this business case:

Pipeline component	Input	Output	Implementation	Standard BIGG component
Ingestor	Query API Energis connect	KAFKA Message	Yodiwo connector	No, Energis.Cloud
Harmoniser	KAFKA Message	Harmonized KAFKA Message	Mapping at data source level	No, Energis.Cloud
Data lake	KAFKA Message	Storage in Mongo DB	Mongo DB	No, Energis.Cloud
Lakeshore	Harmonized KAFKA Message	Storage in Cassandra	Cassandra	No, Energis.Cloud
Analytics toolbox	Get via Energis.Cloud Middleware API	Post via Energis.Cloud Middleware API	Github public library Energy modelling	Yes
Commander			New microservice orchestrating different calls (pipeline)	Yes
Dashboard for clients			M&V Savings dashboard	No, Energis.Cloud

- **For Business case #5**

  Optimisation service module addressing the different goals related to energy efficiency, renewable energy usage, comfort and cost of the building together and not in an isolated way. Weather conditions have a direct impact on the energy demand of buildings (e.g. necessity of heating/cooling). Forecasted weather conditions will allow to proactively match energy demand and supply (e.g. heat less if large solar gains are expected later in the day). Multi-objective function resolution to support building optimization

Here are the technical components that will be setup for V1 implementation of the BIGG RAF for this business case:

Pipeline component	Input	Output	Implementation	Standard BIGG component
Ingestor	Query API Energis connect	KAFKA Message	Yodiwo connector	No, Energis.Cloud
Harmoniser	KAFKA Message	Harmonized KAFKA Message	Mapping at data source level	No, Energis.Cloud
Data lake	KAFKA Message	Storage in Mongo DB	Mongo DB	No, Energis.Cloud
Lakeshore	Harmonized KAFKA Message	Storage in Cassandra	Cassandra	No, Energis.Cloud
Analytics toolbox	Get via Energis.Cloud Middleware API	Post via Energis.Cloud Middleware API	Github public library Occupancy modelling Thermal modelling	Yes
Commander			New microservice orchestrating different calls (pipeline)	Yes
Controller			Configuration of rules on edge device	No, Energis.Cloud
Dashboard for clients			Comfort dashboard	No, Energis.Cloud
Edge devices	Get via Energis.Cloud Middleware API	Commands towards HVAC devices	Rule execution engine	No. Raspicy

IV.4. Business case #6 - Case Study Area: Several cities (Greece)

UC 14 to 15 will leverage both services developed by DOMX and HERON partners, as well as libraries and analytic components developed in BIGG. These components will be dockerized and orchestrated using Docker Swarm and are presented below:

Data ingestion

Gathering of data from different sources:

- Real-time data of IoT devices (electricity submeters, heating controllers, sensors) from time-series DB
- Static data from local SQL DB (user and building data)
- Static data from remote DBs (EPC registry, billing, consumption) and files (CSV, JSON)
- Real-time meteorological data through remote APIs

For the first version, the ingestion components will be developed as Microservices customized for each different data source.

Analytics toolbox

The UCs 14 and 15 will use libraries and components of the BIGG Analytics toolbox. Key required components include the following:

- Fast-clustering – to group datasets
- Regression – to find correlation between variables
- Modelling – to compute predictive models and timeseries digital twins
- Validation – to validate datasets and timeseries
- Forecasting - to forecast the energy consumption of building assets

The following processes will also be executed through the Analytics toolbox

1. Data preparation. Data cleaning, preparation and quality check service will be executed for all collected real-time data sources and implemented based on BIGG OSS components that will be used to transform existing Python scripts into the BIGG architecture. The existing implementation currently includes the following steps:
    - Time stamp alignment
        - Resampling all real-time data to 1sec
        - Interpolation for nan values
            - Resampling again to 10s
    - Outlier detection
        - Filtering out extreme values with predefined thresholds for potential sensor errors
    - Missing data management

- Detecting and zeroing-out disconnection windows.
- Data transformation over prepared data for UCs 14 and 15 will include the harmonization of data from different sources to the internal standard data model of the BIGG architecture.

2. Data modelling. Data modelling over data transformed for UCs 14 and 15 will include the following steps:

- Cross validation
- Model Assessment
- Model identification
- Model persistence and prediction

User interface

The UCs 14 and 15 will use both custom partner dashboards, as well as the common BIGG dashboard of BIGG. The custom dashboards will be used to visualize the output of the Energy efficiency and Flexibility services. The common BIGG dashboard will be used to visualize in real-time the status of participating households IoT sensors, smart meters, status per connected device, user settings. Pilot level calculated KPIs will be visualized both through dashboards and reports.

The following figure shows the execution pipeline and the integration of the BIGG components for UC 15.

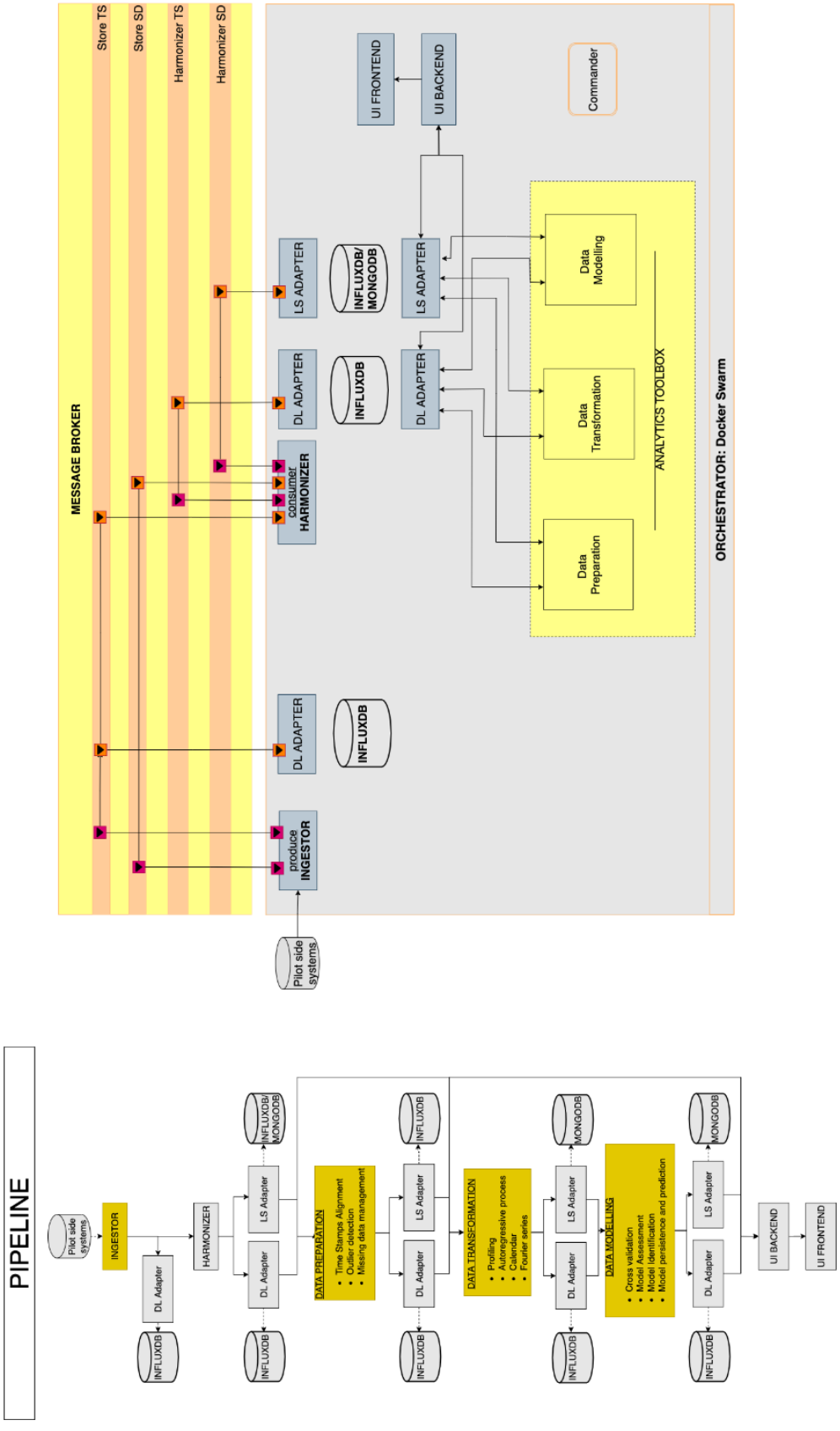

Figure 29) Business cases #6 envisioned technical implementation

Here are the technical components that will be setup for V1 implementation of the BIGG RAF for this business case:

Pipeline component	Input	Output	Implementation	Standard BIGG component
Ingestor	Query local DBs Query API external data sources Import local files	Raw data (JSON)	Microservice customized for each different data source	No DOMX and HERON tools
Harmonizer	Raw data (JSON format) - Broker Message	BIGG format harmonized data:	BIGG OSS components	Yes BIGG OSS components.
Data lake	MQTT Message (JSON Payload)	Storage in Timeseries DB	InfluxDB	No DOMX & HERON tools
Lakeshore	BIGG Harmonizer output	Storage in Timeseries DB	InfluxDB	No DOMX & HERON tools
Analytics toolbox	Harmonized, Processed, Cleaned TimeSeries data	Storage in InfluxDB/Mongo DB	GitHub public repository	Yes
Commander	REST API	MQTT broker message	Cloud device management service	No DOMX and HERON tools
User Interface	REST API	Web dashboards	Backend and frontend implemented in JavaScript	Yes both existing partner dashboards and the common BIGG dashboard.
Edge device	MQTT message	Commands towards: 1. natural gas boilers 2. Electric water heaters	MQTT broker	No. DOMX heating controller HERON meters and actuators

V. CONCLUSIONS

This document has presented the initial technical specifications and preliminary design of the BIGG architecture building blocks. This work has leveraged outputs of the deliverable 6.1 that has technically studied the pilots' use cases and that has emphasised the fact that a cloud-only solution shall 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, during and after the project, the BIGG components integration options must be versatile.

The first section 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 codes 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 second section 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). BIGG possible pipelines were described where information flow from data acquisition to knowledge exposition once the data analysis has been performed on harmonised data by dedicated WP5 components.

Finally, the last section has pragmatically described the planned instantiation of a first BIGG reference architecture for the different BIGG business cases. It has explained which BIGG components will be used by each partner for the first version of the platform implementation.

Indeed, Big Data heterogeneous use cases are complex to harmonize in a single architecture, and it is mandatory to confront architecture envisioned options with the business reality of the different consortiums' partners. The first pragmatic implementation of the BIGG concepts in every pilot site will provide an experience which will lead to the update of the BIGG RAF presented in this document.