MISTRAL
Meteo Italian Supercomputing Portal

Meteo Italian SupercompuTing portal

Deliverable

D5.1 Harvesting and Mapping deploy of input data

Deliverable Lead:	Dedagroup
Deliverable due date	31/12/2019
Status	FINAL
Version	V1.0

his project has received funding from *European Commission*
Connecting Europe Facility 2014-2020
AGREEMENT No INEA/CEF/ICT/A2017/1567101

Document Control Page

Title	D5.1 Harvesting and Mapping deploy of input data
Creator	Dedagroup
Publisher	Mistral Consortium
Contributors	Martina Forconi (DEDAGROUP), Stefano Pezzi (DEDAGROUP)
Type	Report
Language	en-GB
Rights	copyright “Mistral Consortium”
Audience	☐ public ☐ restricted
Requested deadline	

Contents

Structure of Ingestion Flows

The observational data ingestion will be coordinated by Apache NiFi. With this component several data flows will be defined to gather data from providers and store them into the different operational databases:

- DBAll.e (short-term repository)
- ArkiMet (long-term repository)
- STA DB (experimental short-term repository)

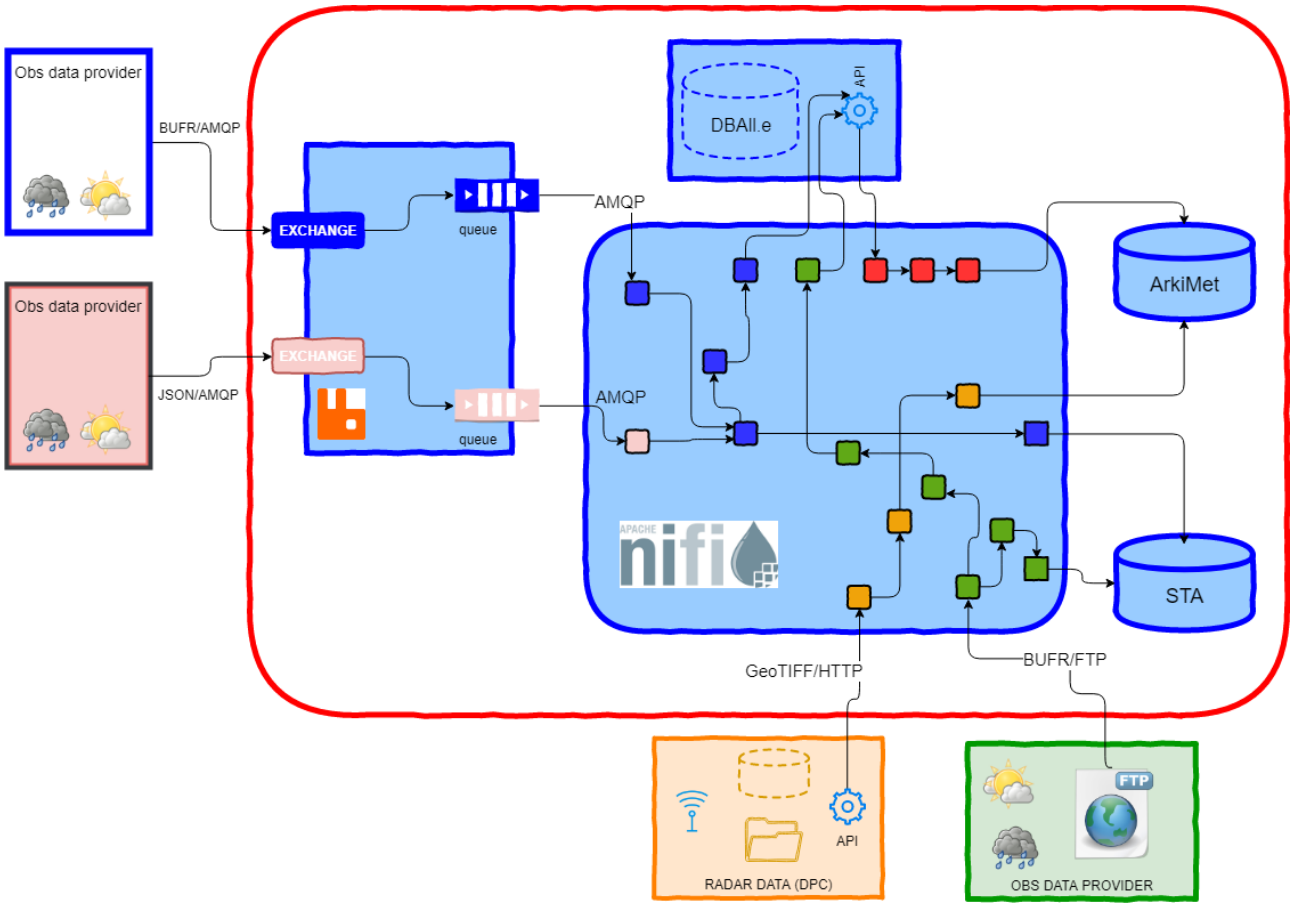


Figure 1 - Data flows managed by Apache NiFi

The four main flows will be the following:

- Radar data ingestion to ArkiMet (orange flow)
- Ground station data to DBAll.e and STA via AMQP (pink/blue flow)
- Ground station data to DBAll.e and STA via FTP (green flow)
- DBAll.e to ArkiMet data transfer (red flow)

Radar data ingestion to ArkiMet

Data types

Several radar data products are available on the DPC platform (WIDE - Weather Ingestion Data Engine) from which they can be downloaded by means of the platform's APIs. The list of available products is:

- "SRI", Surface Rainfall Intensity [frequency: new data every 5min]
- "SRT1", Surface Rain Total, representing the accumulated rainfall in the last 1 hour by integrating radar network data with data of the pluviometric stations on the ground [frequency: new data every 60min]
- "SRT3", Surface Rain Total, representing the accumulated rainfall in the last 3 hours by integrating radar network data with data of the pluviometric stations on the ground [frequency: new data every 60min]
- "SRT6", Surface Rain Total, representing the accumulated rainfall in the last 6 hours by integrating radar network data with data of the pluviometric stations on the ground [frequency: new data every 60min]
- "SRT12", Surface Rain Total, representing the accumulated rainfall in the last 12 hours by integrating radar network data with data of the pluviometric stations on the ground [frequency: new data every 60min]
- "SRT24", Surface Rain Total, representing the accumulated rainfall in the last 24 hours by integrating radar network data with data of the pluviometric stations on the ground [frequency: new data every 60min]
- "VMI", Vertical Maximum Intensity [frequency: new data every 5min]
- "IR108", Infrared satellite channel processed to obtain cloud cover [frequency: new data every 5min]
- "TEMP", Surface temperature from ground stations [frequency: new data every 60min]
- "AMV", Wind AMV Atmospheric Motion Vector. Vector data file representing wind intensity and direction at high altitude coming from satellite remote sensing [frequency: new data every 20min]

- "LTG", Vector data file representing the location of lightnings calculated with triangulation of the signals received by dedicated ground antennas of the LAMPINET[1] network [frequency: new data every 10min].
- "HRD", Heavy Rain Detection, map of important phenomena, classified according to a Severity Index; also displays their possible trajectory in the very short term [frequency: new data every 5min]
- "RADAR_STATUS", Text message with the list of radar stations that are properly working at the time of the query.

The REST APIs are described by this WADL document http://www.protezionecivile.gov.it/wide-api/wide?_wadl where one can find the following resources:

POST /downloadProduct invoked with a JSON object as payload that specifies the name of the product you want to download and the date/time of the product in milliseconds of unix epoch; optionally you can add a key with the period expressed as an ISO8601 duration.

- GET /existsProduct
  verify the existence of a product of a specific time by indicating its name and time in the query parameter.
- GET /findAvaiableProducts (WARNING: typo in resource name)
  Returns a JSON object with the total count and the names of the available products.
- GET /findLastProductByType
  returns a JSON object that describes the more recent product of the requested type; the same kind of JSON that required by /downloadProduct.

The combinate use of the APIs enables a remote client to search for a certain product of a specific time and then download it.

This means that the client must poll constantly the DPC server in order to understand when a product becomes available for download (if you don't know exactly the timestamp of a product you can't download it).

An alternative method makes use of a notification service implemented with the STOMP protocol over a WebSocket connection that the client can establish with the server. In this way, the client can

[1] 15 IMPACT-ESP (IMProved Accuracy through Combined Technology Enhanced Sensitivity & Performance) sensors are installed all over Italy to form the LAMPINET network of Aeronautica Militare.

receive a JSON message with the description of the product and its reference time soon after the product becomes available.

```
{
  "productType" : "SRT1",
  "time" : 1575151200000,
  "period" : "PT1H"
}
```

The formats of the downloadable data are raster GeoTIFF and vector shapefiles.

GeoTIFF

The raster file CRS (Coordinate Refence System) is EPSG:4326 (WGS84). GeoTiff files have different pixel dimensions, but all of them have only one data band storing a float32 number per cell, expressing the phenomenon measure. No compression has been used to create these files even though several lossless algorithms are available for GeoTiff format. The raster files have also internal overview pyramids that speed up visualization, but at the same time increase the file size. NULL values are represented with the conventional value of -9999.

Product	Raster Dimension	Pixel size (degree)	File size	Period	Retention	# files per day	Total daily size
SRI	1200x1400	~ 0.013 x 0.009 (48″ x 33″)	12MB	5′		288	3.5GB
SRT1	631x576	~ 0.020 x 0.020 (1′ 12″ x 1′ 12″)	4.5MB	1h		24	108MB
SRT3	631x576	~ 0.020 x 0.020 (1′ 12″ x 1′ 12″)	4.5MB	1h		24	108MB
SRT6	631x576	~ 0.020 x 0.020 (1′ 12″ x 1′ 12″)	4.5MB	1h	7d	24	108MB
SRT12	631x576	~ 0.020 x 0.020 (1′ 12″ x 1′ 12″)	4.5MB	1h		24	108MB

SRT24	631x576	~ 0.020 x 0.020 (1' 12" x 1' 12")	4.5MB	1h		24	108MB
VMI	1200x1400	~ 0.013 x 0.009 (48" x 33")	12MB	5'		288	3.5GB
IR108	600x600	0.041 x 0.028 (2' 28" x 1' 40")	4.5MB	5'		288	1.3GB
TEMP	631x576	~ 0.020 x 0.020 (1' 12" x 1' 12")	4.5MB	1h		24	108MB
total size							**8.8GB**

Table 1 - Properties of the radar products

Shapefile

Product	Note	File size	Period	# file per day	Total day size
AMV	*Error in download*	variable	20'	72	
LTG	--	variable	10'	144	--
HRD	--	variable	5'	288	

Estimated annual size of radar data = 9GB x 365 = 3.2TB

In *Figure 1* the Radar data flow is represented by the orange chain of processors. This flow can be implemented in two different ways.

Polling method

This is the scenario in which NiFi as a list of product types to download and they respective periods of update. Following this information, for every product “TypeX” with nominal update period of “UpdIntX” Nifi will check the presence of an updated version of the product “UpdIntX” seconds after the last download.

Suppose that TypeX = SRT1, a product with a nominal update interval of 1 hour.

The polling mechanism will be like this

1. After 1 hour from the last successful download, NiFi will invoke the following REST service in this way:

```
GET /product/findLastProductByType&type=SRT1
```

2. Suppose that the timestamp of the last downloaded file was 1575216000000 (and that it was stored in some way to let the system understand if a new data file is present); if the WIDE platform returns a message like this:

```
{
  "total": 1,
  "lastProducts": [
    {
      "productType": "SRT1",
      "time": 1575219600000,
      "period": "PT1H"
    }
  ]
}
```

NiFi then would understand that the file has changed;

3. NiFi will ask for the downloading of the new file with this request:

```
POST /product/downloadProduct

{
  "productType": "SRT1",
  "productDate": 1575219600000,
  "period": "PT1H"
}
```

It will as well store the new timestamp and will schedule the next check after an hour.

4. Otherwise, if the “time” key is equal to the old one, NiFi would start polling immediately with a shorter period (i.e. a minute) to become aware of the update as soon as possible.

Listening method

The listening mode is based on the notification service offered by the WIDE platform.

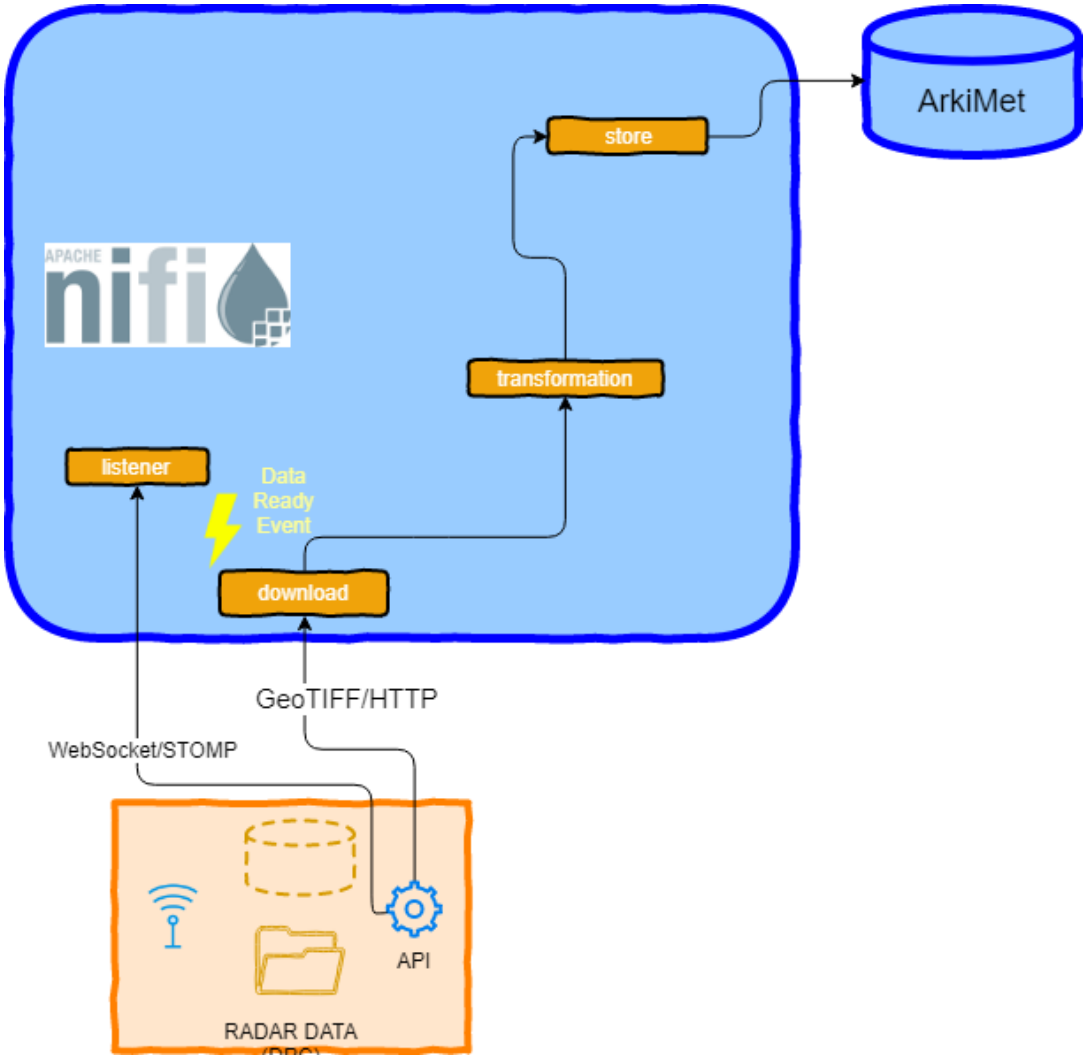


Figure 2 - GeoTiFF flow (listening method)

1. Nifi must open a WebSocket connection at the WIDE server URL:

```
GET ws://www.protezionecivile.gov.it/wide-websocket
```

The connection must be opened specifying that it will be used as a channel for a communication with the STOMP protocol. This is done using dedicated headers.

2. After the successful establishment of the connection (a 101-return code from the server), the channel is ready for bi-directional communication with the server. Periodically the connection could be closed by the server; when this happens, the client should re-open the connection.

3. With the WebSocket channel open, the client can start the STOMP communication sending a CONNECT request message.

```
CONNECT
accept-version:1.0,1.1,1.2
```

```
heart-beat:120000,0
```

4. The server will respond with an acknowledge of the connection:

```
CONNECTED
server:RabbitMQ/3.7.14
session:session-r1kegFR1jEeEQ3lsvIEZJQ
heart-beat:0,120000
version:1.2
```

5. Then the client asks to subscribe for notification of a topic "/topic/product" that's the queue where the server send messages about product availability.

```
SUBSCRIBE
ack:auto
id:sub-0
destination:/topic/product
```

6. After the subscription, the client waits for notifications. Also a closure of the underlying WebSocket channel can happen (actually it seems to happen regularly every 35 s) and the client must soon re-open it and start again a STOMP session and subscription.

7. When a product is ready, the server sends a message with the information (type, timestamp and period) of the product using the MESSAGE syntax:

```
MESSAGE
subscription:sub-0
destination:/topic/product
message-id:T_sub-0@@session-sc5BsSSgk-dtdcf_D8_gHA@@1
redelivered:false
content-type:application/json;charset=UTF-8
content-length:74
{
  "productType" : "VMI",
  "time" : 1575152100000,
  "period" : "PT5M"
}
```

More than one MESSAGE during the same session can follow, advertising the availability of other products.

8. Using the information communicated by the server, NiFi can thus request the download of the advertised product.

POST /product/downloadProduct

```
{
  "productType": "VMI",
  "productDate": 1575152100000,
  "period": "PT5M"
}
```

Tranformation

In order to store data in Arkimet, GeoTIFF must be transformed in GRIB file.

This is achieved in four steps with the help of GDAL tools[2]:

1. Harmonization of nodata values

The GeoTIFF file downloaded from DPC has nodata pixels valorized in two different ways:

- The reprojection artifacts on the outer boundary of the image are valorized with 'nan'
- The pixels laying outside of the radar range are valorized with '-9999'

The **gdalwarp** tool is therefore called to harmonize all nodata values to '-9999':

```
gdalwarp original.tif -srcnodata nan -dstnodata -9999 transformed_step1.tif
```

2. Unit conversion

The product downloaded from DPC is the Surface Rainfall Intensity (SRI) which is expressed in millimiters of rain per hour (mm/h), while the requirement is to store it in Arkimet in SI units (Kg/m^2 s). For water this roughly eqates to a conversion to mm/s = mm/h * 1/3600.

The **gdal_calc.py** tool is therefore called to make the necessary unit conversion:

```
gdal_calc.py -A transformed_step1.tif  --outfile= transformed_step2.tif
--calc="A*0.000277778"
```

3. Nodata value conversion

This step is necessary to reset nodata value after the numeric operation performed in the preceding step. The requirement is to set nodata pixels to a value that is customarily assumed in the meteorological community to represent nodata, so to ensure the correct fruition of the data by the widest possible number of custom data consuming softwares.

The **gdalwarp** tool is therefore called again to set all nodata values to '9999':

```
gdalwarp transformed_step2.tif -dstnodata 9999 transformed_step3.tif
```

[2] GDAL is a translator library for raster and vector geospatial data formats that is released under an X/MIT style Open Source License by the Open Source Geospatial Foundation. As a library, it presents a single raster abstract data model and single vector abstract data model to the calling application for all supported formats. It also comes with a variety of useful command line utilities for data translation and processing. https://gdal.org/

4. Format conversion

Finally, the data format needs to be transformed in GRIB and the correct metadata has to be provided[3]. Metadata are part of the GRIB file and are fundamental for the ingestion in arkimet, in order to route data to the correct dataset.

Morover, GRIB format allows for several data encoding options, some of which are designed to pack data and reduce data volume when many nodata pixels are present, as it is our case.

The **gdal_translate** is therefore called to convert data to GRIB format and save matadata about reference time, data origin, product type and encoding:

```
gdal_translate -of GRIB -co "IDS=CENTER=80 SIGNF_REF_TIME=3 REF_TIME=$2
DISCIPLINE=0" -co "PDS_TEMPLATE_ASSEMBLED_VALUES=1 52 8 255 50 0 0 1 0 1 0
0 1 0 0" -co "DATA_ENCODING=COMPLEX_PACKING" -stats transformed_step3.tif
transformed.grib
```

Data storage

After the format transformation, the GRIB file can be stored into the Arkimet repository. The repository is organized in datasets, which are physically represented by folder in the file system. Each dataset is described by a 'config' file that resides in the dataset's folder and lists the attributes (such as data origin, area, product type, etc) that uniquely identify the dataset with respect to the other datasets.

The separation between datasets with respect to the different licence types will be implemented by setting the custom attribute '_licence' in the configuration files of Arkimet's datasets.

The config file of the dataset 'radar_dpc', designed to stor the SRI data form DPC, is the following:

```
name = radar_dpc
description = Surface Rainfall from Radar-DPC
format = grib
filter = origin:GRIB2,80,,8,,50; area:GRIB:Ni=1437,Nj=1155;
step = daily
type = iseg
```

For a detailed description of attributes' meaning refer to Arkimet's documentation[4]

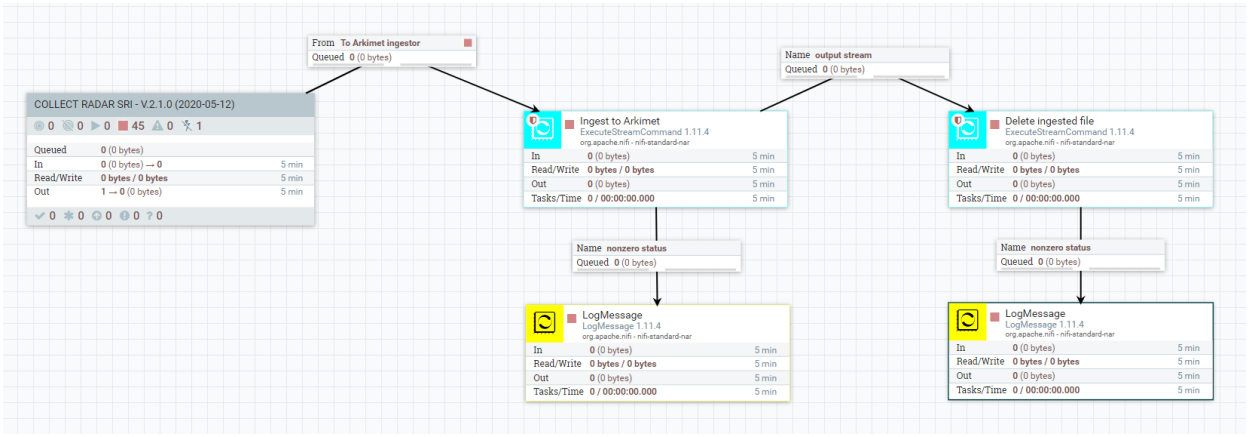
The storage itself is triggeret by a simple call to the arki-scan tool:

[3] https://gdal.org/drivers/raster/grib.html
[4] https://arpa-simc.github.io/arkimet/index.html

```
arki-scan --dispatch=config trasformed.grib > /dev/null
```

Implementation

The ingestion flow described in the previous paragrapsh has been implemented operationally with Apache NiFi[5], a very powerful and versatile tool for data flow design and management.



The implemented flow has two main components[6]:

- **COLLECT RADAR SRI**: this module implements the polling approach, it checks for the most recent availability of the SRI product and downloads each new release as soon as it's availability is verified. This module also performs the conversion of the downoladed GeoTIFF data to a GRIB file and the rescaling of the data content into a more suitable unit.

- **INGEST to ARKIMET**: this module is the recipient of the outputs of the previous two modules. Its task is that of sequentially ingest the GRIB files provided by the other modules into ArkiMet by calling the arki-scan command. The module takes also care of deleting the original files, once ingested in Arkimet.

The input parameter necessary to this flow are:

- The URL of DPC's API resources from which the radar data are fetched.
- The connection parameters (host, port, database, username and password) to the postgres db used as back log resource by the flow, together with the schema and table name of the log table itself.

[5] https://nifi.apache.org/

[6] A third component was also developed, but just as an experimental prototype, capable of receiving GRIB datasets directly from a Rabbit MQ queue and saving them as files, potentially then feeding the file to the 'Ingest to Arkimet' module.

- The working directories where files are downloaded and transformed.

The log table for the radar flow has a very simple structure, storing the information on:

- Product code ('SRI' for Surface Rainfall Intensity)
- The timestamp of reference for the observed data
- The timestamp of download
- The filename where the downloaded data was stored

The aim of the log table is to keep track of what data has been already downloaded and where in the file system. Moreover, the NiFi flows uses the log information to avoid making multiple downloads of the same data.

The SQL create statement for the log table is the following:

```
CREATE TABLE mistraldev.rad_downloads (
          prod_code varchar NOT NULL,
          prod_creation_date timestamptz NOT NULL,
          prod_download_date timestamptz NULL,
          prod_filename varchar NULL,
          CONSTRAINT rad_downloads_pk PRIMARY KEY (prod_code,
prod_creation_date)
);
```

Ground station data to DBAll.e and STA via AMQP

There are two short-term repositories in Mistral: one is implemented by a **DB-All.e** and the other is the PostgreSQL database that backs the **STA** server. DB-All.e can be backed by different DBMSs among which we also have PostgreSQL, and this is the one chosen for Mistral platform. Short-term repositories store observational data for a brief period (2 weeks) in order to speed up the retrieving of the most recent data by Mistral consumers. Every day, the older data are transferred to the long-term repository, slower, but able to store efficiently a huge quantity of observations.

The STA repository is meant to offer a new kind of standard service to consumers and it is kind of an experimental component.

Ground station data may be delivered by providers in two different ways:

- enqueuing BUFR messages in an AMQP endpoint exposed by Mistral
- enqueuing JSON messages in an AMQP endpoint exposed by Mistral

To implement the AMQP channels, Mistral deploys a message broker component implemented by RabbitMQ where “exchanges” and “queues” will be defined respectively to receive messages from providers and to publish them for the data flow engine NiFi. This component will implement the flows to the repositories.

RabbitMQ provides a robust implementation of AMQP protocol so the architecture of the ingestion module includes this component for exposing the exchange endpoints.

One more requirement rises from a legal point of view: observational data are provided to Mistral platform with two kind of licenses that cannot mix together. This means that there will be two separate datasets in DBAll.e where the two types of licensed data will be ingested into and this dichotomy will be preserved up to the final long-term repository Arkimet.

The experimental STA repository will deal with only one type of license.

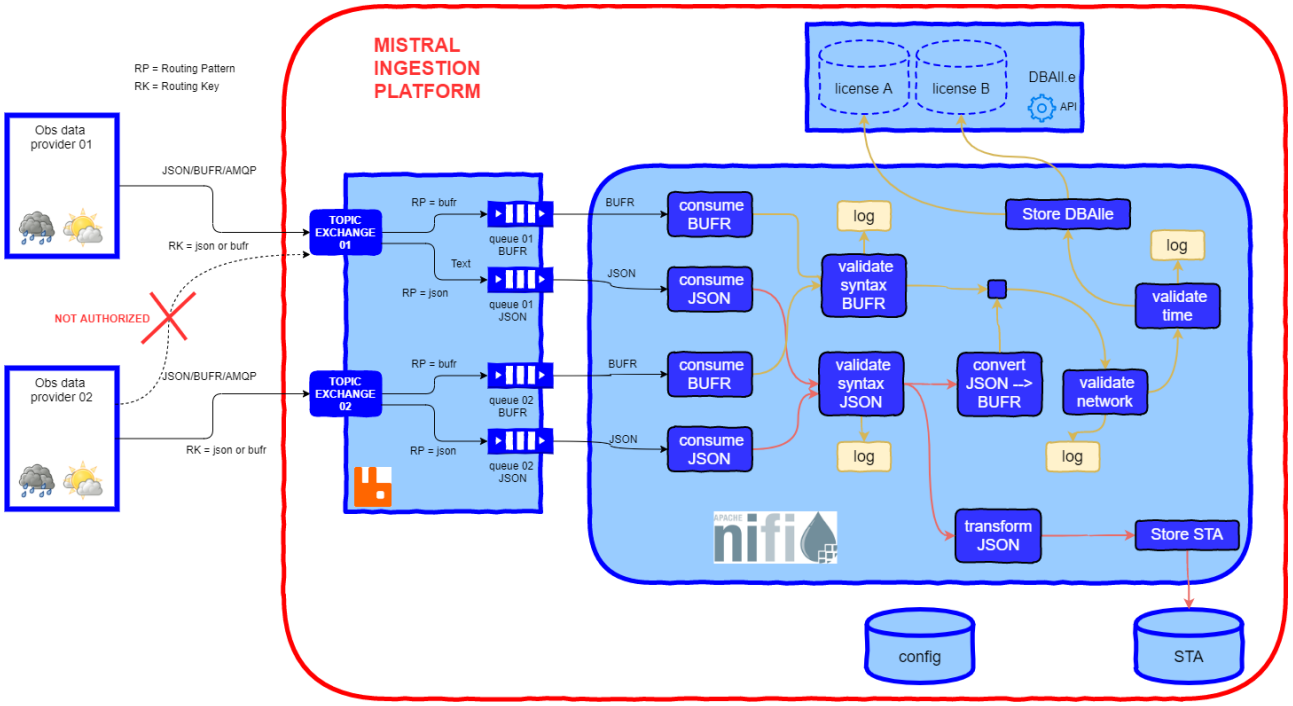


Figure 3 - Observational data ingestion flow

Authentication

Every provider will have a dedicated AMQP endpoint exposed by Mistral platform, that is what RabbitMQ calls an **exchange**[7]. To write messages into this dedicated exchange, a provider must use an **authorized account**. Here messages can be post in both the allowed formats: JSON or BUFR. The format used must however be specified as part of the **topic**[8] of the message[9] and will be configured as **routing key**[10] since these exchanges will be **topic exchanges**. A topic exchange is an exchange which routes messages to queues by applying rules based on the wildcard match between routing key and routing pattern specified during the binding of the queue.

As a result, each exchange will be bound to two queues one for JSON messages and one for BUFR messages.

[7] Exchanges, queues and bindings are called AMQP entities.
[8] A message in an AMQP communication is composed by a label and a payload; there are no origin or destination. The label includes the name of the *exchange* to which the message is directed and, optionally, a topic tag. The exchange is no the real destination, since the consumers are waiting on the queues and the messages reach the queues by mean of a routing strategy defined by the bindings.
[9] The default topic will be "json"
[10] A routing key is a list of word concatenated and separated by dots with a maximum length of 255 bytes.

All data supplied by a single provider will be linked to only a type of license, so each queue will contain homogeneous messages from both the format and the license points of view.

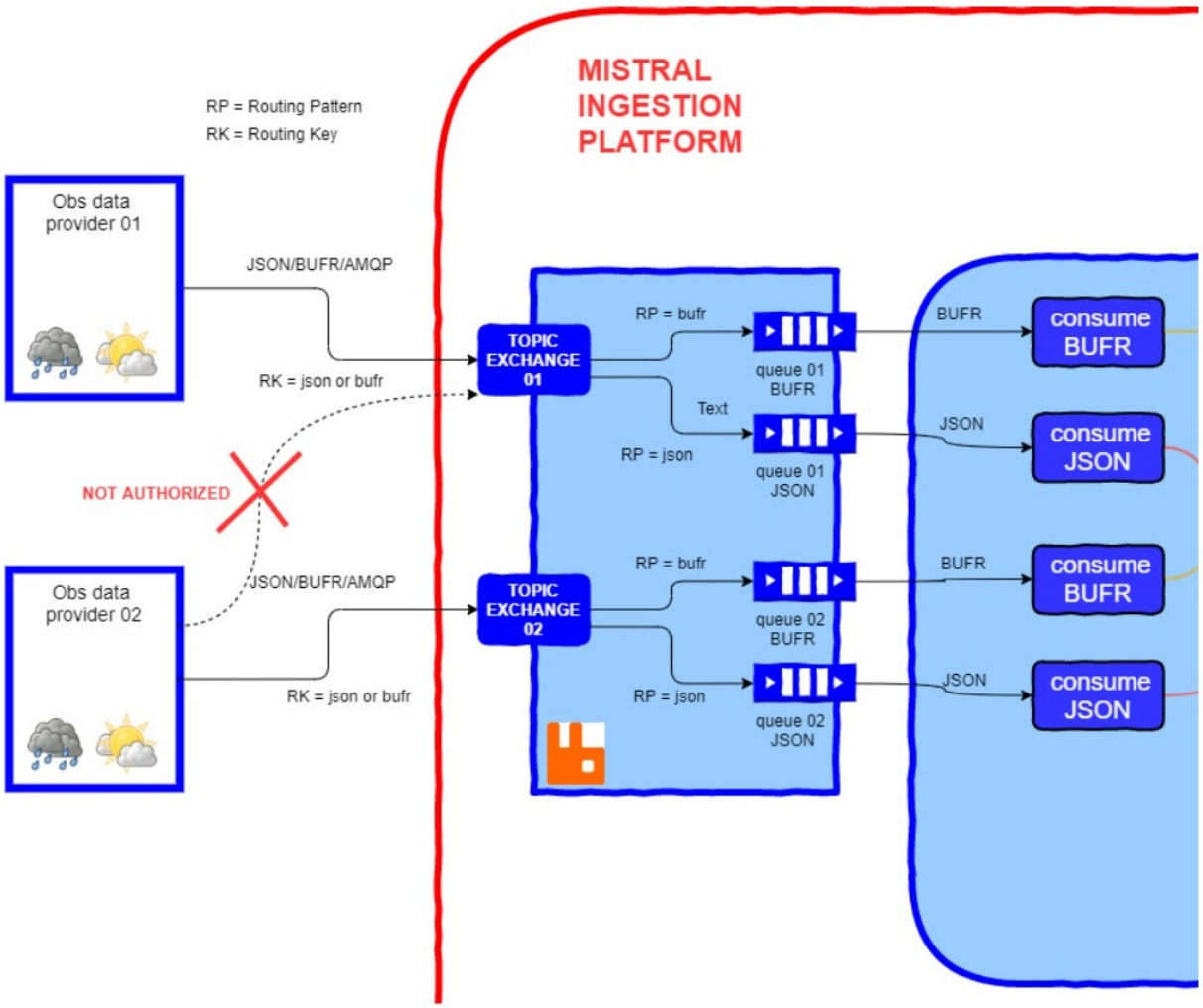


Figure 4 - Detail of the AMQP exchanges and queues

The conversion process

In the DBAll.e repository only BUFRs will be stored, so the JSON messages must be converted along their flow.

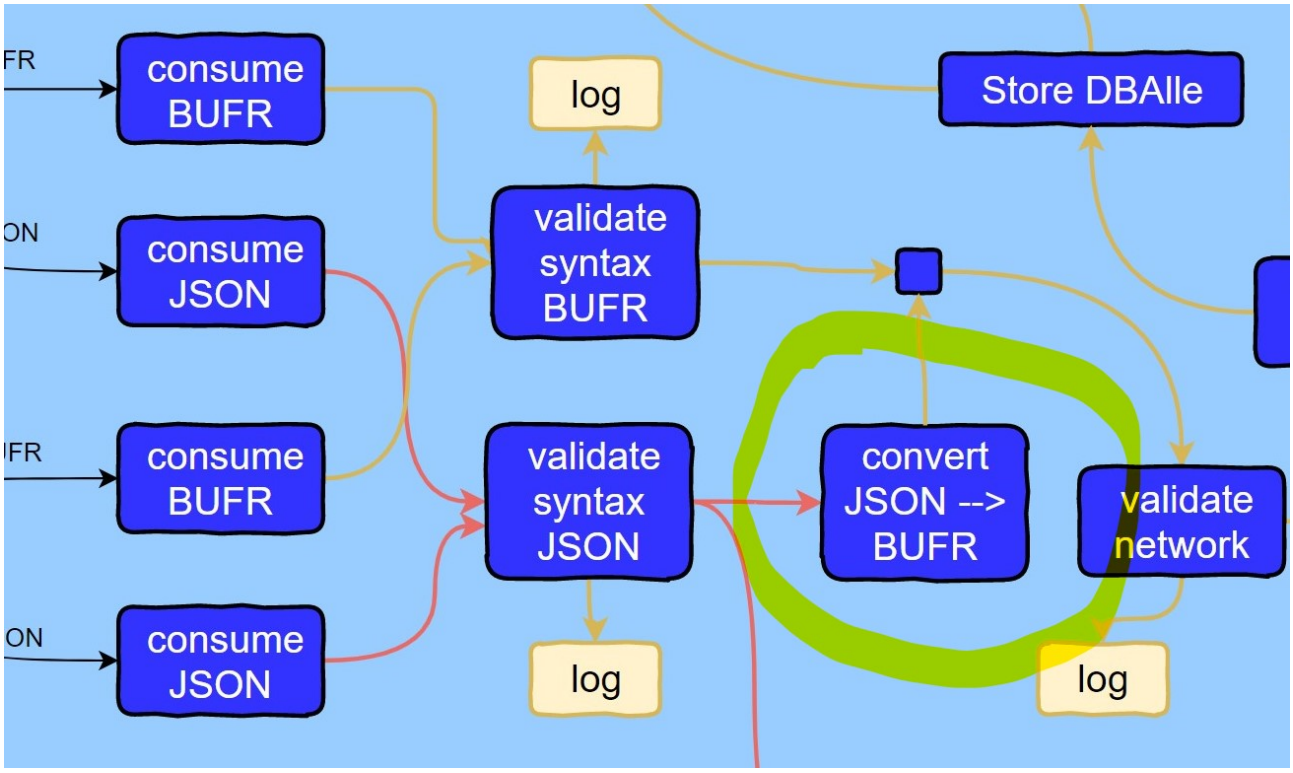


Figure 5 - Detail of the conversion process

In *Figure 5* the conversion process is highlighted: one can see that it applies only to the JSON branch of the flow and only after the validation of the correct syntax of the message. After the conversion, the messages take the same route of the native BUFR messages and go through the other types of validation.

The Validation process

After withdrawing messaging from each queue, NiFi has to validate every message based on three main rules (other can be added if necessary). These rules are:

1. each provider can supply only data belonging to certain networks. This is necessary to prevent a provider overwriting other providers' data.
2. data that arrive into the platform must refer to a specific time range and undershoots or overshoots respect this range must be discarded. This is done mostly because of the existence of the subdivision in short- and long-term repositories: when a BUFR message is transferred to Arkimet it is assumed that there are no more observations for that range of time and that station, and latecomer data are not accepted.
3. the message must be correct from the syntactic point of view.

This validation steps take place on the incoming messages soon afterwards the conversion phase that transforms JSON messages into BUFR; to be more precise, since this conversion can take place only if the JSON message is correct, then the BUFR validation is done in the BUFR branch of the flow before the two streams merge.

The configuration DB

There is some important information that the ingestion system must be aware of to correctly manage the flows, for instance to apply a specific validation rule. This information eventually can change during the operational life of the system (like a new data provider joining the platform or a change in the license policy) and thus it's better to have them stored in a DataBase schema created inside the PostgreSQL instance of the platform. This Ingestion Config database will contain this kind of information:

- the list of data providers
- which exchange and queue are assigned to which provider
- whose network a provider can provide data
- the tolerable error in the time coordinate of the observations
- the maximum delay of the observations (correspond to the time depth of the short-term repository)

Constraints and performance issues

Message order

Messages coming from the same provider must be stored in same order in which they arrive at the AMQP endpoint. Sometimes, in fact, a provider sends soon after a certain message, a correction to this message that fixes some kind of error contained in the previous one. The fixing message has exactly the same "dimensions" that characterize the wrong message, in order to overwrite it when stored in DBAll.e. However, during a step of the flow implemented by a multithreaded elaboration, the natural order of the messages could be altered by the random nature of the timing of the concurrent threads.

If the correct message would reach DBAll.e before the wrong message, then this latter would overwrite the former, resulting in a loss of information.

So, it's very important that, despite any concurrent elaboration that could take place during the flow, the original order of the messages of the same provider, is maintained when messages are stored in DBAll.e.

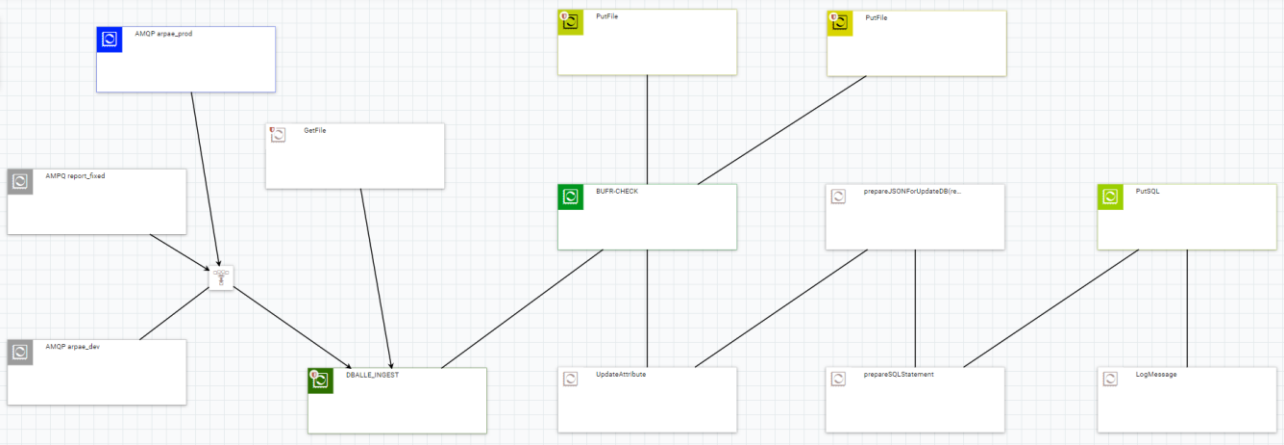
Performance

As a very high throughput of incoming messages (minimum 1000 msg/min) is expected, attention must be paid to Python scripts that transform message format or store it into DBAll.e. NiFi's ExecuteScript creates a new ScriptEngine for each one of the tasks specified in the Max Concurrent Tasks property and reuses those engines for each flow file. This behaviour should be tested and assured to avoid loading libraries and creating connection for each flow file, just to trigger a simple task that could be the only repetitive duty of the processor.

Log

Messages that don't pass validations or make a processor fail should not be wasted. They must be saved somewhere (DB, filesystem) in order to easy debug operations. A limit for the maximum number of discarded messages can be set.

Implementation



The implemented flow reads BUFR-formatted data from a RabbitMQ queue and feeds it to a python script that verifies the data compliance in terms of the network and timerance validation constraints and ingests valid data into DBAll.e.

Depending on the outcome of the compliance validation the data is saved as file in two different physical locations. Moreover, any validation error is logged into aspecific table of NiFi support DB (postgres).

An experimental extension of this flow has also been implemented in order to test the ingestion of ground station data formatted in JSON Line format. The flow converts to GRIB an input file in JSON Line format by calling the 'dbamsg convert' tool of DBAll.e and then continues with the ingestion flow described above.