SBDIO I4.0
Big Data
for Industry

O5.2

Prototipo dei servizi out per la realizzazione di soluzioni B2B

basate su big data analytics e focalizzati su macchine automatiche che operano al di fuori dello shop floor di una fabbrica

Code	05.2
Date	30/09/2020
Type	Confidential
Participants	UNIFE
Authors	Mauro Tortonesi (UNIFE), Marco Govoni (UNIFE), Federico Frigo (UNIFE)
Corresponding Authors	Mauro Tortonesi

POR FESR
EMILIA-ROMAGNA
2014/2020

UNIONE EUROPEA
Fondo europeo di sviluppo regionale

RegioneEmilia-Romagna

SBDIO I4.0
Big Data
for Industry

Sommario

1. Architettura

1.1. Architettura Carpigiani

In Figura 1 si riprende, per completezza, l'architettura di raccolta e memorizzazione dati descritta nel precedente deliverable 05.1. Questa versione è attualmente distribuita e funzionante all'interno di Carpigiani, ed è costituita dai seguenti componenti:

- **Cleaner**: un modulo software realizzato internamente a Carpigiani che permette di spacchettare i dati dal formato "grezzo" nel nuovo formato JSONB
- **PostgreSQL**: Data Warehouse che conterrà i dati in arrivo dal Cleaner
- Logstash: componente di ingestion, che si occupa di prelevare i dati da PostgreSQL, arricchendoli e applicando alcune modifiche
- **Elasticsearch**: si occupa di salvare i dati indicizzandoli, per ottimizzare la loro fruizione al layer di visualizzazione
- **Kibana**: componente per la visualizzazione, permette di creare nuove visualizzazioni e dashboard al fine di mostrare determinati risultati

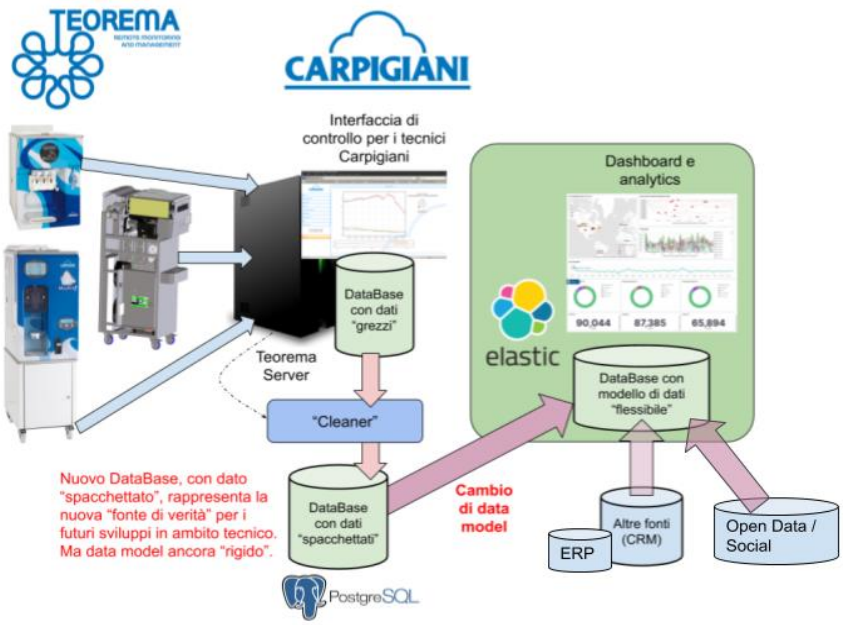

Fig. 1. Architettura della piattaforma di analisi Big Data

1.2. Architettura a Virtual Machines (VMs)

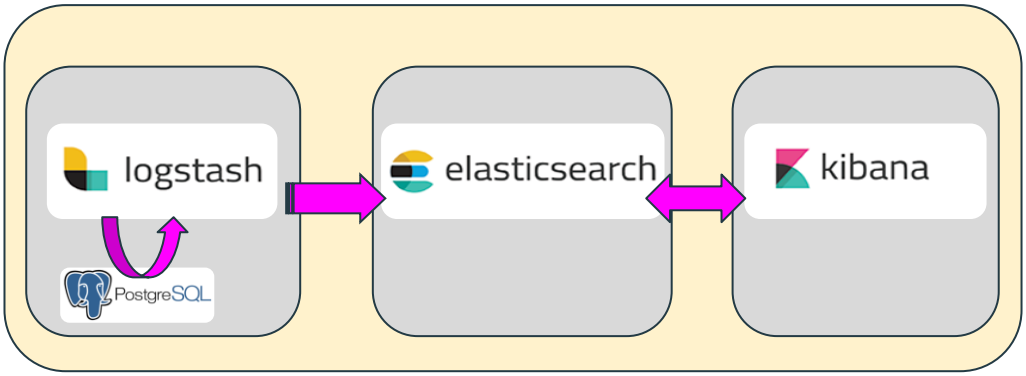

Fig. 2. Deploy dei componenti su piattaforma OpenStack

Nell'ambito di questo progetto, ed in particolare dell'attività Servizi Out, si è deciso di evolvere la distribuzione attuale dell'architettura al di sopra di un'infrastruttura cloud-based. Per questo, sono stati ri-creati e distribuiti tutti i componenti software, a parte il Cleaner che è di competenza Carpigiani, su un'infrastruttura cloud-based resa disponibile dal partner di progetto T3Lab. Questa nuova distribuzione è realizzata tramite uno strumento open source di Infrastructure as a Service (IaaS) chiamato **OpenStack** (Figura 2). In questa versione, come evidenziato in figura, l'unità fondamentale di computazione sono le **macchine virtuali (VMs)**, dove ognuna di queste è basata su sistema operativo **Ubuntu**. La suddivisione dei componenti è la seguente:

- **VM-1**: PostgreSQL v.11 e LogStash v.7.2
- **VM-2**: Elasticsearch v.7.2
- **VM-3**: Kibana v.7.2

Essendo lo stack ELK scritto in Java, l'unico prerequisito necessario sulle VM è la presenza di Java 8, per cui è stata opportunamente installata la JVM OpenJDK-8 per ogni macchina virtuale. La quantità di CPU, RAM e memoria necessaria su ogni macchina dipende chiaramente dal volume dei dati che si intende raccogliere. In particolare uno snapshot di una macchina FDM pesa 3200 byte che, sommato a un overhead di storage, diventa come spazio effettivo su disco pari a 4000 byte. Al giorno mediamente la FDM produce 200 snapshots. Per il momento sono state stabilite le seguenti risorse per ogni VM:

- 4 vcpu
- 8GB RAM
- 100GB HDD

1.3. Architettura a Container

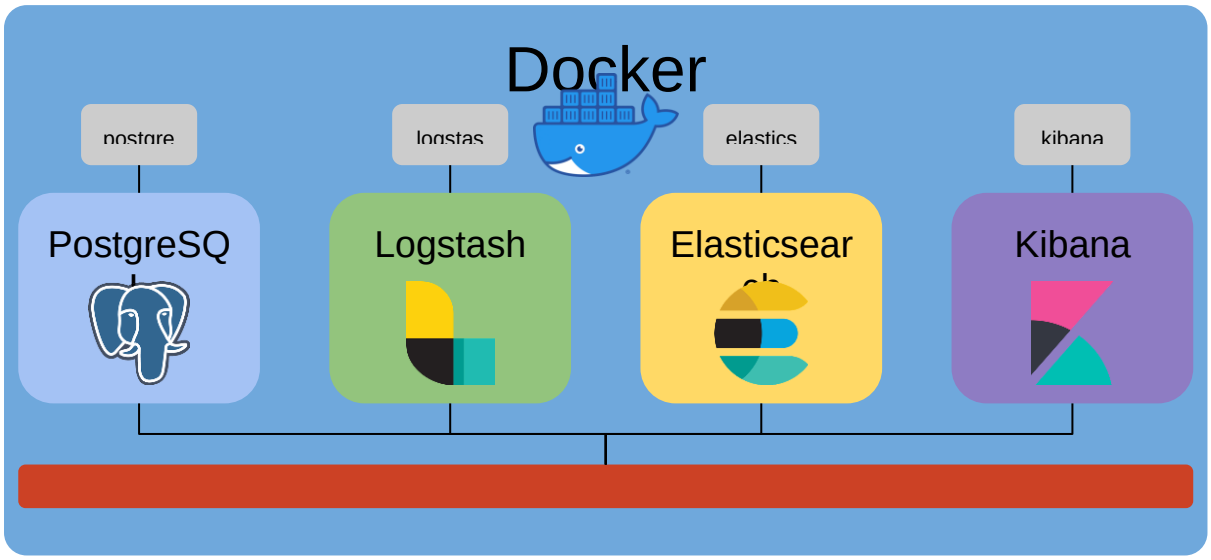

Fig. 3. Architettura della piattaforma su Docker

Attualmente, in fase di sviluppo all'interno di UniFE, è stata ulteriormente evoluta la distribuzione dell'architettura passando da VMs ad unità di computazione basate sui **container Docker** (Figura 3). Questa scelta è motivata dal fatto di velocizzare lo sviluppo ed il testing dei vari componenti sfruttando i benefici dei container Docker.

In questo scenario, è stato impiegato lo strumento **Docker Compose** per la definizione ed esecuzione di applicazioni multi-container. Docker Compose si occupa di automatizzare il download e il setup dei vari componenti, consentendo di eseguire e/o terminare in maniera molto semplice l'intera architettura. La struttura presentata in Figura 3 prevede quattro container contenenti i diversi servizi (PostgreSQL, Logstash, Elasticsearch e Kibana) con i rispettivi layer di storage dedicati ed un network per la loro comunicazione (elk_net).

L'impiego di Docker Compose prevede l'utilizzo di un unico file YAML per la configurazione dell'infrastruttura (*docker-compose.yml*) che definisce il setup di tutti gli strumenti e velocizza la loro esecuzione tramite semplici comandi (Par. 3.7 - Script realizzati). Questo file YAML contiene quindi i parametri di configurazione per ciascun servizio, e saranno descritti nei paragrafi seguenti i dettagli di ogni configurazione.

```
# docker-compose.yml

version: '3.7'

services:
  postgres:
    …

  elasticsearch:
    …

  logstash:
    …

  kibana:
    …

networks:
  elk_net:

volumes:
  postgres_data:
  elasticsearch_data:
  logstash_data:
  kibana_data:
```

2. Descrizione dei dati

2.1. Descrizione dati macchine Carpigiani modello FDM

In questa fase di prototipazione possiamo prendere come riferimento le macchine Carpigiani modello FDM, in quanto sono quelle che forniscono informazioni più complete e dettagliate. Sono anche le macchine più complesse all'interno del portfolio di Carpigiani.

Il Cleaner, come già detto di competenza Carpigiani, prende in ingresso gli snapshots, li decodifica tramite la mappatura della macchina (in questo caso la FDM), trasformandoli nel formato Hash Ruby. All'interno dell'hash sarà infatti presente una parte comune per tutte le macchine (common attributes) e una parte

specifica della macchina (snapshot data), compreso il machine_datetime. I dati decodificati vengono caricati direttamente su PostgreSQL utilizzando un API che li trasforma in JSONB. Una volta che viene caricato l'ultimo snapshot, lo script si salva in memoria il suo ID, in modo tale che all'esecuzione successiva riparte senza effettuare duplicazioni. Il nuovo formato JSONB è il seguente:

```
{
  machine_id: ID della macchina sul DB originale Teorema,
  serial: seriale della macchina,
  material_code: codice prodotto,
  model: modello macchina,
  installation_date: data di installazione della macchina,
  time_zone: time zone della città di installazione,
  teorema_msd_id: ID dello snapshot sul DB originale Teorema,
  location: {
    lat: latitudine
    lon: longitudine
  }
  location_details: {
    restaurant: nome del locale,
    city: città,
    country: paese,
    address: indirizzo
  },
  snapshot_data:{
    machine_datetime: orario macchina
    # campi variabili a seconda della macchina
    . . .
  }
}
```

Come già anticipato, la tabella del database PostgreSQL contenente i nuovi dati si chiama ***snapshots*** ed è costituita dalle seguenti colonne:

1. **document**: è lo snapshots in formato JSONB
2. **created_at**: mostra quando è stato salvato lo snapshot
3. **updated_at**: mostra quando è stata fatta l'ultima modifica
4. **id**: è l'id della sessione di monitoraggio nella quale è stato inviato lo snapshot

Per i test è stato sviluppato uno script in python (snapshot_extracting_job.py) per l'estrazione di dati da un file .csv fornito direttamente da Carpigiani, contenente i dati di una macchina riguardanti il mese

di Gennaio, che deve essere avviato a mano; si tratta di una versione molto semplificata del Cleaner di Carpigiani.

2.2. Dati meteo

2.2.1. OpenWeatherMap

OpenWeatherMap è uno dei principali fornitori di informazioni meteorologiche digitali. Si tratta di una piccola azienda IT con sede in UK, fondata nel 2014 da un gruppo di ingegneri ed esperti in Big Data, elaborazione dati ed elaborazione di immagini satellitari. Offrono delle API gratuite, previa registrazione di un nuovo utente, e anche dei piani a pagamento per dati più precisi e per il supporto tecnico. Per la descrizione dei vari piani è possibile consultare la sezione dedicata (https://openweathermap.org/price) nel loro sito.

In questo contesto sono state utilizzate le API per le previsioni fino a cinque giorni a partire dalla data corrente, ad intervalli di 3 ore (00.00, 03.00, 06.00, ...). Le informazioni vengono reperite a partire da stazioni meteo presenti in alcune città di interesse (in Italia la copertura è molto buona), indirizzabili tramite il nome o il codice univoco specifico fornito direttamente nel sito; naturalmente, non tutti i piccoli paesini italiani presentano delle stazioni meteo, ma è possibile ottenere previsioni sufficientemente accurate in base alle stazioni più vicine.

Nel caso del piano gratuito le chiamate API sono limitate, 60 al minuto, e sono cumulative, quindi qualsiasi chiamata viene sempre conteggiata (ad esempio, è possibile chiamare 1 previsione al secondo per 1 città, 1 previsione ogni 12 secondi per 5 città, ...); questa limitazione viene estesa o rimossa nel caso di upgrade del piano.

Di seguito si riporta a scopo illustrativo il risultato di una chiamata API per la città di Ferrara:

```
# curl http://api.openweathermap.org/data/2.5/forecast?q=ferrara&appid=API_KEY
# Chiamata eseguita alle 15:20 il 01/09/2020
{
  "cod": "200",
  "message": 0,
  "cnt": 40,
  "list": [
    {
      "dt": 1598972400,
```

```
    "main": {
      "temp": 299.68,
      "feels_like": 299.48,
      "temp_min": 299.55,
      "temp_max": 299.68,
      "pressure": 1011,
      "sea_level": 1010,
      "grnd_level": 1011,
      "humidity": 41,
      "temp_kf": 0.13
    },
    "weather": [
      {
        "id": 800,
        "main": "Clear",
        "description": "clear sky",
        "icon": "01d"
      }
    ],
    "clouds": {
      "all": 0
    },
    "wind": {
      "speed": 1.25,
      "deg": 253
    },
    "visibility": 10000,
    "pop": 0,
    "sys": {
      "pod": "d"
    },
    "dt_txt": "2020-09-01 15:00:00"
  },
  {
    "dt": 1598983200,
    "main": {
      "temp": 294.78,
      "feels_like": 293.41,
      "temp_min": 293.4,
      "temp_max": 294.78,
      "pressure": 1011,
      "sea_level": 1011,
      "grnd_level": 1012,
      "humidity": 59,
      "temp_kf": 1.38
    },
    "weather": [
```

```
      {
        "id": 800,
        "main": "Clear",
        "description": "clear sky",
        "icon": "01n"
      }
    ],
    "clouds": {
      "all": 0
    },
    "wind": {
      "speed": 3.42,
      "deg": 132
    },
    "visibility": 10000,
    "pop": 0.02,
    "sys": {
      "pod": "n"
    },
    "dt_txt": "2020-09-01 18:00:00"
  },
  {
    "dt": 1598994000,
    "main": {
      "temp": 292.98,
      "feels_like": 292.15,
      "temp_min": 292.51,
      "temp_max": 292.98,
      "pressure": 1012,
      "sea_level": 1012,
      "grnd_level": 1013,
      "humidity": 69,
      "temp_kf": 0.47
    },
    "weather": [
      {
        "id": 802,
        "main": "Clouds",
        "description": "scattered clouds",
        "icon": "03n"
      }
    ],
    "clouds": {
      "all": 42
    },
    "wind": {
      "speed": 2.98,
```

```
        "deg": 124
      },
      "visibility": 10000,
      "pop": 0.01,
      "sys": {
        "pod": "n"
      },
      "dt_txt": "2020-09-01 21:00:00"
    }
    ...
  ],
  "city": {
    "id": 3177088,
    "name": "Provincia di Ferrara",
    "coord": {
      "lat": 44.8,
      "lon": 11.8333
    },
    "country": "IT",
    "population": 358972,
    "timezone": 7200,
    "sunrise": 1598934925,
    "sunset": 1598982611
  }
}
```


2.2.2. Rp5.ru

Di fortissimo interesse sono ovviamente anche i dati meteo storici, in grado di arricchire i dati provenienti dalle macchine, che servono per comprendere al meglio alcuni parametri (ad esempio, il motivo per il quale in un giorno specifico sono stati venduti meno coni del solito, che potrebbe essere legato al maltempo). OpenWeatherMap fornisce anche i dati storici, però nel piano gratuito si limitano ad appena 5 giorni prima della data corrente; per questo è stato necessario trovare una fonte di dati alternativa, da qui Rp5.ru (https://rp5.ru/).

Si possono scaricare i dati meteo storici dal loro archivio, scegliendo il periodo di interesse. Ovviamente il formato è diverso, quindi si è reso necessario scrivere uno script in python (meteo_extracting_job.py) in grado di trasformare i dati in un formato il più possibile compatibile con quello di OpenWeatherMap (alcuni campi forniti da quest'ultimo non sono presenti nei dati forniti da Rp5.ru).

I campi a disposizione sono:

```
{
  "datetime": timestamp,
  "temperature": temperatura in °C
  "pressure_sea_level": pressione al livello del mare,
  "humidity": umidità,
  "wind_direction": direzione del vento,
  "wind_speed": velocità del vento,
  "rain": volume delle precipitazioni,
}
```

3. Implementazione

3.1. PostgreSQL

PostgreSQL è un database relazionale utilizzato per mantenere i dati provenienti dalle macchine FDM (salvati in formato JSONB).

Di seguito viene riportata la porzione di interesse nel file "docker-compose.yml" riguardante PostgreSQL:

```
# docker-compose.yml
…

services:
  postgres:
    image: postgres
    container_name: postgres
    restart: always
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: database
    volumes:
      - type: bind
        source: ./postgres/CreateTable.sql
        target: /docker-entrypoint-initdb.d/CreateTable.sql
      - type: volume
        source: postgres_data
        target: /var/lib/postgresql/data
    networks:
      - elk_net
```

```
    ports:
      - 5432:5432
    hostname: postgres

…
```

PostgreSQL viene scaricato direttamente da Docker Hub (https://hub.docker.com/_/postgres) all'ultima versione disponibile (se si volesse una versione specifica è sufficiente scrivere la versione nella voce "image", concorde alla disponibilità di Docker Hub).

"container_name" e "restart" sono utili internamente a Docker per gestire il container. Le variabili di ambiente sotto "environment" definiscono username, password e nome del db, mentre "networks", "ports" e "hostname" servono per far comunicare questo servizio con gli altri, attivi nella stessa rete. Infine è presente il volume virtuale per lo storage persistente dei dati, e viene inoltre caricato lo script "CreateTable.sql" che si occupa di creare la tabella all'avvio del servizio, dove verranno salvati i dati provenienti dal Cleaner (in questa fase di test, lo script "snap_extracting_job.py" si occupa sia di estrarre e rimodellare i dati dal file .csv, sia di salvarli all'interno del database).

3.2. Logstash

Si tratta del componente che si occupa dell'ingestion, ossia recupera i dati da diverse fonti di dati, li modella a seconda delle richieste e necessità per poi inviarli a qualche altro servizio (tipicamente verso Elasticsearch). Tutto questo viene definito tramite delle pipelines, file di configurazione che presentano tre campi:

- Input: definisce la sorgente dei dati e come interfacciarsi ad essa
- Filter: definisce le operazioni eseguite sui dati
- Output: definisce la destinazione dei dati e come interfacciarsi ad essa

Di seguito viene riportata la porzione di interesse nel file "docker-compose.yml" riguardante Logstash:

```
# docker-compose.yml
…

  logstash:
    image: logstash:7.8.1
```

```
    container_name: logstash
    depends_on:
      - elasticsearch
    restart: on-failure
    environment:
      LS_JAVA_OPTS: '-Xmx256m -Xms256m'
    volumes:
      - type: bind
        source: ./logstash/pipelines.yml
        target: /usr/share/logstash/config/pipelines.yml
      - type: bind
        source: ./logstash/pipeline
        target: /usr/share/logstash/pipeline
      - type: bind
        source: ./logstash/jdbc-drivers
        target: /usr/share/logstash/jdbc-drivers
      - type: bind
        source: ./logstash/templates
        target: /usr/share/logstash/templates
      - type: volume
        source: logstash_data
        target: /usr/share/logstash/data
    networks:
      - elk_net
    ports:
      - 9600:9600
      - 50000:50000
    hostname: logstash

…
```

Logstash viene scaricato direttamente da Docker Hub (https://hub.docker.com/_/logstash) all'ultima versione disponibile (se si volesse una versione specifica è sufficiente scrivere la versione nella voce "image", concorde alla disponibilità di Docker Hub).

"container_name" e "restart" sono utili internamente a Docker per gestire il container. La variabile di ambiente sotto "environment" definisce la RAM complessiva utilizzabile da Logstash, mentre "networks", "ports" e "hostname" servono per far comunicare questo servizio con gli altri, attivi nella stessa rete.

Sotto "volumes" è presente il solito storage per i dati di Logstash, ma sono presenti anche:

- “pipelines.yml” e la cartella “pipeline”, dei quali il primo definisce le pipeline disponibili, mentre il secondo contiene i file di configurazione delle varie pipelines
- “jdbc-drivers” è una cartella contenente i driver jdbc per la connessione ai database (in questo caso, solo quello utile a connettersi a PostgreSQL)
- “templates” contiene i file di template, utili a definire il formato dei dati che verranno poi inoltrati a Elasticsearch, in modo che vengano deserializzati correttamente

3.2.1. Pipeline “postgres.conf”

Pipeline per l’ingestion dei dati provenienti da PostgreSQL.

Input

```
input {
  jdbc {
    # Postgres jdbc connection string
    jdbc_connection_string => "jdbc:postgresql://postgres:5432/database"
    # Postgres user
    jdbc_user => "user"
    # Postgres Password
    jdbc_password => "password"
    # Jdbc driver path
    jdbc_driver_library => "/usr/share/logstash/jdbc-drivers/postgresql-
42.2.13.jar"
    # Jdbc driver class name
    jdbc_driver_class => "org.postgresql.Driver"
    # Postgres query
    statement => "SELECT document::text, document->>'teorema_msd_id' AS id,
document#>>'{snapshot_data, machine_datetime}' AS time FROM snapshot WHERE
document->>'model' LIKE '%FDM%' AND document->>'serial' LIKE '%IC%' AND
(document#>>'{snapshot_data, machine_datetime}')::timestamp > :sql_last_value
ORDER BY (document#>>'{snapshot_data, machine_datetime}')::timestamp LIMIT 500"
    # Query scheduled every 1 minutes
    schedule => "*/1 * * * *"
    # Paging to avoid Java heap space error
    record_last_run => true
    use_column_value => true
    tracking_column => "time"
    tracking_column_type => "timestamp"
    last_run_metadata_path =>"/usr/share/logstash/data/.last_run_metadata"
  }
}
```

Il campo “input” della pipeline definisce come connettersi (stringa di connessione al database e locazione del driver jdbc) e come autenticarsi (username e password) con il database, la query da eseguire per recuperare i dati (che vengono paginati per evitare problemi di esaurimento della ram), la schedulazione della query e quale campo salvare per evitare di recuperare dati già letti.

Filter

```
filter {
  # Parsing from text to json
  json {
    source => "document"
    remove_field => ["document"]
  }
}
```

Il campo “filter” definisce la conversione del dato “document” dal formato testo al formato JSON per poi rimuovere quello obsoleto in formato testo e mantenendo solo quello in formato JSON.

Output

```
output {
  # Output to Elasticsearch
  elasticsearch {
    hosts => ["elasticsearch"]
    document_id => "%{id}"
    index => "fdm_snaps"
    template => "/usr/share/logstash/templates/location.json"
    template_name => "location"
  }

  # Debug output
  # stdout { codec => rubydebug }
}
```

Il campo “output” definisce la destinazione dei dati (Elasticsearch), l’id del documento e l’indice in cui salvarlo e il template del formato dei dati con annesso il suo nome.

3.2.2. Pipeline “meteo-historical.conf”

Pipeline per l’ingestion dei dati meteo storici, recuperati da un file .csv tramite lo script “meteo_extracting_job.py”.

Input

```
input {
  # Get input from TCP port 50000
  tcp {
    codec => json
    port => 50000
  }
}
```

Viene definito su quale porta si deve mettere in ascolto Logstash in attesa di una connessione TCP; infatti, lo script “meteo_extracting_job.py”, una volta estratti i dati dal file .csv e dopo averli elaborati e trasformati in formato JSON, li invia tramite il protocollo TCP sulla porta 50000 a Logstash .

Filter

```
filter {
  # Convert string timestamp
  date {
    match => ["datetime", "ISO8601"]
  }

  # Remove unused fields
  mutate {
    remove_field => ["port", "host", "datetime"]
  }
}
```

Esegue la trasformazione della data da testo ad un formato riconosciuto da Elasticsearch, e rimuove alcuni campi inutilizzati.

Output

```
output {
  # Output to Elasticsearch
  elasticsearch {
    hosts => ["elasticsearch:9200"]
    document_id => "%{id}"
    index => "meteo_historical"
    template => "/usr/share/logstash/templates/location.json"
    template_name => "location"
  }

  # Debug output
  # stdout { codec => rubydebug }
}
```

Il campo “output” definisce la destinazione dei dati (Elasticsearch), l’id del documento e l’indice in cui salvarlo e il template del formato dei dati con annesso il suo nome.

3.2.3. Pipeline “meteo-nowcast.conf”

Pipeline per l’ingestion dei dati meteo attuali, recuperati tramite le API di OpenWeatherMap.

Input

```
input {
  # Get current meteo for several cities every 10 minutes
  http_poller {
    codec => json
    urls => {
      current => {
        method => get
        url =>
"http://api.openweathermap.org/data/2.5/group?appid=API_KEY&id=3181927,6540120,6
541461,6542002,3169561,6541863,6542120,3176746,6542025&units=metric&lang=it"
      }
    }
    schedule => { every => "10m"}
  }
}
```

Utilizza il plugin “http_poller” per eseguire una chiamata API schedulata una volta ogni 10 minuti; nell’endpoint sono definiti l’API_KEY e le varie città di interesse tramite i loro codici identificativi (reperibili nel sito di OpenWeatherMap).

Filter

```
filter {
  # Remove unused 'cnt' field
  mutate {
    remove_field => ["cnt"]
  }

  # Split city list in single events
  split {
    field => "list"
  }

  # Copy nested fields to the root
  ruby {
    code => "
      list = event.get('list')

      event.remove('list')

      list.each {| k, v |
        event.set(k, v)
      }
    "
  }

  # Convert UNIX timestamp
  date {
    match => ["dt", "UNIX"]
    timezone => "Europe/Rome"
  }

  # Copy only important fields and remove unused fields
  mutate {
    copy => {
      "[name]" => "city"
      "[sys][country]" => "country"
      "[coord]" => "location"
      "[main][temp]" => "temperature"
      "[main][pressure]" => "pressure_sea_level"
      "[main][humidity]" => "humidity"
      "[wind][speed]" => "wind_speed"
```

```
      "[wind][deg]" => "wind_direction"
      "[rain][3h]" => "rain"
      "[snow][3h]" => "snow"
    }

    remove_field => ["coord", "weather", "base", "main", "wind", "clouds", "dt",
"sys", "timezone", "id", "name", "cod", "visibility"]
  }

  # Add rain field if it doesn't exist
  if ![rain] {
    mutate {
      add_field => { "rain" => 0 }
    }
    mutate {
      convert => { "rain" => "integer" }
    }
  }

  # Add snow field if it doesn't exist
  if ![snow] {
    mutate {
      add_field => { "snow" => 0 }
    }
    mutate {
      convert => { "snow" => "integer" }
    }
  }
}
```

Il filtro esegue varie operazioni, tra cui rimuovere i campi inutilizzati, formattare correttamente alcuni campi in modo da permettere il loro riconoscimento in Elasticsearch tramite il template e aggiungere campi mancanti per una corretta elaborazione e visualizzazione su Kibana.

Output

```
output {
  # Output to Elasticsearch
  elasticsearch {
    hosts => ["elasticsearch:9200"]
    document_id => "%{city}_%{@timestamp}"
    index => "meteo_nowcast"
  }
```

```
  # Debug output
  # stdout { codec => rubydebug }
}
```

Il campo “output” definisce la destinazione dei dati (Elasticsearch), l’id del documento e l’indice in cui salvarlo e il template del formato dei dati con annesso il suo nome.

3.2.4. Pipeline “meteo-forecast.conf”

Pipeline per l’ingestion dei dati meteo futuri, recuperati tramite le API di OpenWeatherMap.

Input

```
input {
  # Get 5days/3hours meteo forecast for several cities every 10 minutes
  http_poller {
    codec => json
    urls => {
      forecast5 => {
        method => get
        url =>
"http://api.openweathermap.org/data/2.5/forecast?appid=0b6d9f9ffa7ee67bac8dbd6c59896dd1&id=3181927&units=metric&lang=it"
      }
    }
    schedule => { every => "10m" }
  }
  http_poller {
    codec => json
    urls => {
      forecast5 => {
        method => get
        url =>
"http://api.openweathermap.org/data/2.5/forecast?appid=0b6d9f9ffa7ee67bac8dbd6c59896dd1&id=6540120&units=metric&lang=it"
      }
    }
    schedule => { every => "10m" }
  }
  http_poller {
```

```
    codec => json
    urls => {
      forecast5 => {
        method => get
        url =>
"http://api.openweathermap.org/data/2.5/forecast?appid=0b6d9f9ffa7ee67bac8dbd6c5
9896dd1&id=6541461&units=metric&lang=it"
      }
    }
    schedule => { every => "10m" }
  }
  http_poller {
    codec => json
    urls => {
      forecast5 => {
        method => get
        url =>
"http://api.openweathermap.org/data/2.5/forecast?appid=0b6d9f9ffa7ee67bac8dbd6c5
9896dd1&id=6542002&units=metric&lang=it"
      }
    }
    schedule => { every => "10m" }
  }
  http_poller {
    codec => json
    urls => {
      forecast5 => {
        method => get
        url =>
"http://api.openweathermap.org/data/2.5/forecast?appid=0b6d9f9ffa7ee67bac8dbd6c5
9896dd1&id=3169561&units=metric&lang=it"
      }
    }
    schedule => { every => "10m" }
  }
  http_poller {
    codec => json
    urls => {
      forecast5 => {
        method => get
        url =>
"http://api.openweathermap.org/data/2.5/forecast?appid=0b6d9f9ffa7ee67bac8dbd6c5
9896dd1&id=6541863&units=metric&lang=it"
      }
    }
    schedule => { every => "10m" }
  }
```

```
  http_poller {
    codec => json
    urls => {
      forecast5 => {
        method => get
        url =>
"http://api.openweathermap.org/data/2.5/forecast?appid=0b6d9f9ffa7ee67bac8dbd6c5
9896dd1&id=6542120&units=metric&lang=it"
      }
    }
    schedule => { every => "10m" }
  }
  http_poller {
    codec => json
    urls => {
      forecast5 => {
        method => get
        url =>
"http://api.openweathermap.org/data/2.5/forecast?appid=0b6d9f9ffa7ee67bac8dbd6c5
9896dd1&id=3176746&units=metric&lang=it"
      }
    }
    schedule => { every => "10m" }
  }
  http_poller {
    codec => json
    urls => {
      forecast5 => {
        method => get
        url =>
"http://api.openweathermap.org/data/2.5/forecast?appid=0b6d9f9ffa7ee67bac8dbd6c5
9896dd1&id=6542025&units=metric&lang=it"
      }
    }
    schedule => { every => "10m" }
  }
}
```

Utilizza il plugin “http_poller” per eseguire una chiamata API schedulata una volta ogni 10 minuti per le 9 città selezionate; negli endpoint sono definiti l’API_KEY e la città di interesse tramite il suo codice identificativo (reperibile nel sito di OpenWeatherMap).

Filter

```
filter {
  # Reorganize event
  ruby {
    code => "
      list = event.get('list')
      city = event.get('city')

      event.remove('cod')
      event.remove('message')
      event.remove('cnt')
      event.remove('list')
      event.remove('city')

      event_list = []

      list.each {|el|
        new_event = {
          'city' => city['name'],
          'location' => city['coord'],
          'country' => city['country'],
          'dt' => el['dt'],
          'temperature' => el['main']['temp'],
          'pressure_sea_level' => el['main']['pressure'],
          'humidity' => el['main']['humidity'],
          'wind_speed' => el['wind']['speed'],
          'wind_direction' => el['wind']['deg'],
          'rain' => el['rain'] ? el['rain']['3h'] : 0,
          'snow' => el['snow'] ? el['snow']['3h'] : 0,
        }
        event_list.push new_event
      }

      event.set('event_list', event_list)
    "
  }

  # Split the event list in single events
  split {
    field => "event_list"
  }

  # Convert UNIX timestamp
  date {
    match => ["[event_list][dt]", "UNIX"]
    timezone => "Europe/Rome"
```

```
  }

  # Copy fields to the root and remove unused fields
  mutate {
    copy => {
      "[event_list][city]" => "[city]"
      "[event_list][location]" => "[location]"
      "[event_list][country]" => "[country]"
      "[event_list][temperature]" => "[temperature]"
      "[event_list][pressure_sea_level]" => "[pressure_sea_level]"
      "[event_list][humidity]" => "[humidity]"
      "[event_list][wind_speed]" => "[wind_speed]"
      "[event_list][wind_direction]" => "[wind_direction]"
      "[event_list][rain]" => "[rain]"
      "[event_list][snow]" => "[snow]"
    }

    remove_field => ["event_list"]
  }
}
```

Il filtro esegue varie operazioni, tra cui rimuovere i campi inutilizzati, formattare correttamente alcuni campi in modo da permettere il loro riconoscimento in Elasticsearch tramite il template e aggiungere campi mancanti per una corretta elaborazione e visualizzazione su Kibana.

Output

```
output {
  # Output to Elasticsearch
  elasticsearch {
    hosts => ["elasticsearch:9200"]
    document_id => "%{city}_%{@timestamp}"
    index => "meteo_forecast"
    template => "/usr/share/logstash/templates/location.json"
             template_name => "location"
  }

  # Debug output
  # stdout { codec => rubydebug }
}
```

Il campo “output” definisce la destinazione dei dati (Elasticsearch), l’id del documento e l’indice in cui salvarlo e il template del formato dei dati con annesso il suo nome.

3.4. Elasticsearch

Si tratta del componente che si occupa di salvare ed indicizzare i dati provenienti da Logstash, per fornire un rapido accesso da parte di Kibana al fine di creare delle visualizzazioni.

Di seguito viene riportata la porzione di interesse nel file “docker-compose.yml” riguardante Elasticsearch:

```
# docker-compose.yml
…

  elasticsearch:
    image: elasticsearch:7.8.1
    container_name: elasticsearch
    restart: on-failure
    environment:
      # Use single node discovery in order to disable production mode and avoid
bootstrap checks
      # see
https://www.elastic.co/guide/en/elasticsearch/reference/current/bootstrap-
checks.html
      discovery.type: single-node
      ES_JAVA_OPTS: '-Xms512m -Xmx512m'
    volumes:
      - type: volume
        source: elasticsearch_data
        target: /usr/share/elasticsearch/data
    networks:
      - elk_net
    ports:
      - 9200:9200
      - 9300:9300
    hostname: elasticsearch

…
```

Elasticsearch viene scaricato direttamente da Docker Hub (https://hub.docker.com/_/elasticsearch) all'ultima versione disponibile (se si volesse una versione specifica è sufficiente scrivere la versione nella voce "image", concorde alla disponibilità di Docker Hub).

"container_name" e "restart" sono utili internamente a Docker per gestire il container. Le variabile di ambiente sotto "environment" definiscono il numero di nodi partecipanti al cluster di Elasticsearch (in questo caso solo un nodo è presente) e la RAM complessiva utilizzabile da Elasticsearch, mentre "networks", "ports" e "hostname" servono per far comunicare questo servizio con gli altri, attivi nella stessa rete.

Sotto "volumes" è presente lo storage per i dati di Elasticsearch.

3.5. Visualizzazioni con Kibana

Si tratta del componente che offre agli utenti, anche quelli non tecnici, di creare visualizzazioni sui dati salvati ed indicizzati in Elasticsearch.

Di seguito viene riportata la porzione di interesse nel file "docker-compose.yml" riguardante Kibana:

```
# docker-compose.yml
…

  kibana:
    image: kibana:7.8.1
    container_name: kibana
    depends_on:
      - elasticsearch
    restart: on-failure
    volumes:
      - type: volume
        source: kibana_data
        target: /usr/share/kibana/data
    networks:
      - elk_net
    ports:
      - 5601:5601
    hostname: kibana

…
```

Kibana viene scaricato direttamente da Docker Hub (https://hub.docker.com/_/kibana) all'ultima versione disponibile (se si volesse una versione specifica è sufficiente scrivere la versione nella voce "image", concorde alla disponibilità di Docker Hub).

"container_name" e "restart" sono utili internamente a Docker per gestire il container. I campi "networks", "ports" e "hostname" servono per far comunicare questo servizio con gli altri, attivi nella stessa rete e sotto "volumes" è presente lo storage per i dati di Kibana.

3.6. Casi d'uso

Durante la fase di prototipazione sono nati tre casi d'uso interessanti, con i quali è stato possibile testare la nuova piattaforma e verificarne gli effettivi vantaggi.

3.6.1. Temperatura cabinet FDM

Il primo caso riguarda la temperatura del cabinet della FDM. Carpigiani deve rispettare determinate leggi sanitarie relative allo stato della miscela. Per far ciò deve garantire che la temperatura all'interno del cabinet si mantenga in uno specifico range di valori. Essa potrebbe uscire da questo intervallo o perché è stato lasciato aperto lo sportello della cabinet per troppo tempo oppure per via di qualche problema tecnico dovuto al malfunzionamento dell'impianto. Carpigiani quindi deve essere in grado di dimostrare qual è la causa. Risulta quindi necessario poter monitorare la temperatura della cabinet (t_cab) in relazione con l'allarme cabinet_opened e con una serie di dati che informano sullo stato della macchina.

La seguente visualizzazione mostra la posizione delle FDM sulla mappa (tutte le FDM sono geolocalizzate), una heatmap che raffigura gli allarmi "cabinet_open" delle FDM in un range di tempo selezionato, la media delle temperature delle cabinet di ciascuna FDM e globale di tutte le FDM, infine il numero di coni prodotti e la percentuale di permanenza in uno stato da ciascun cilindro delle FDM.

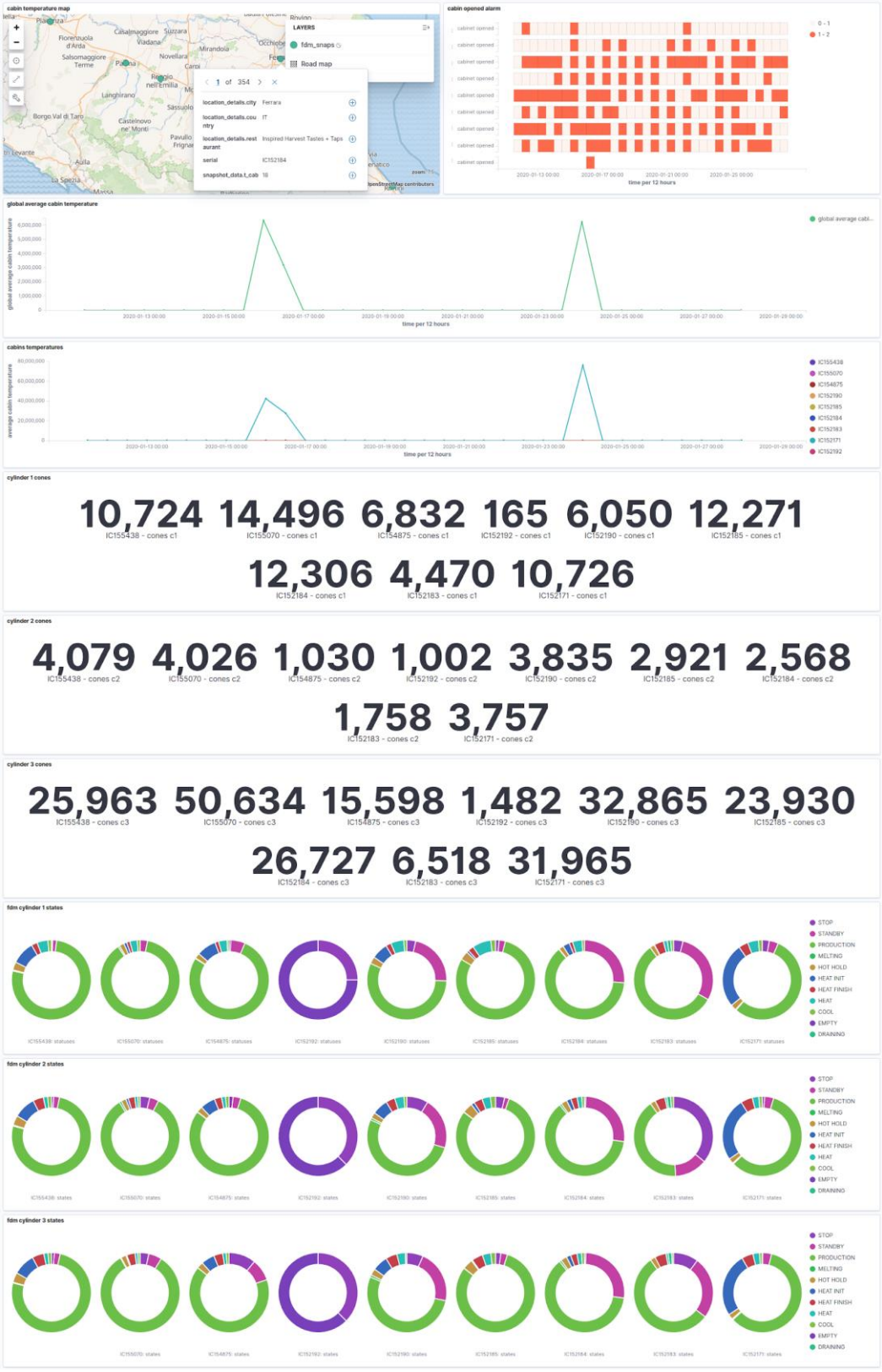

Fig. 4. Dashboard temperature cabinet delle FDM in Kibana

3.6.2. Pressione cilindri delle FDM

Il secondo caso d'uso interessa lo stato della pressione dei tre cilindri (pressure_c1/c2/c3). Risultava che alcune macchine possedevano valori anomali di pressione che si avvicinavano, senza quasi mai raggiungerla, alla soglia che fa scattare l'allarme. Di conseguenza il tecnico, non accorgendosi dell'irregolarità, non interveniva.

Nella seguente dashboard sono stati inseriti tre grafici a torta che mostrano lo stato dei cilindri e soprattutto tre coppie di grafici, appositamente affiancati, che raffigurano nel tempo lo stato degli allarmi di mix_pressure in relazione con l'andamento della pressione dei tre cilindri. Grazie a queste operazioni è stato possibile individuare un legame tra una particolare impostazione che la macchina aveva nella tabella di programmazione, con il comportamento irregolare delle pressioni.

Fig. 5. Dashboard pressione cilindri delle FDM in Kibana

3.6.3. Previsioni meteo

L'ultimo caso d'uso riguarda le previsioni meteo; queste sono utili per comprendere ancora meglio alcuni dati, come ad esempio la scarsità di produzione di coni in giorni specifici (ad esempio a causa del maltempo), o una sovrapproduzione di coni in altri (che può portare ad un malfunzionamento della macchina).

A scopo di test e per semplicità le dashboard per il meteo sono state suddivise in tre, sebbene in produzione sia preferibile unificarle ed utilizzare un unico servizio per i dati meteo, così da potersi man mano costruire la propria cronologia di dati meteo affidabile.

Nella seguente dashboard sono presenti una mappa con i luoghi di residenza delle macchine segnati da dei marker, un grafico che mostra la temperatura media dei giorni selezionati, uno che mostra il volume medio delle precipitazioni e uno delle possibili nevicate, uno che mostra la percentuale di umidità media, una coppia di grafici che mostrano la velocità e la direzione del vento e infine l'ultimo grafico mostra la media della pressione.

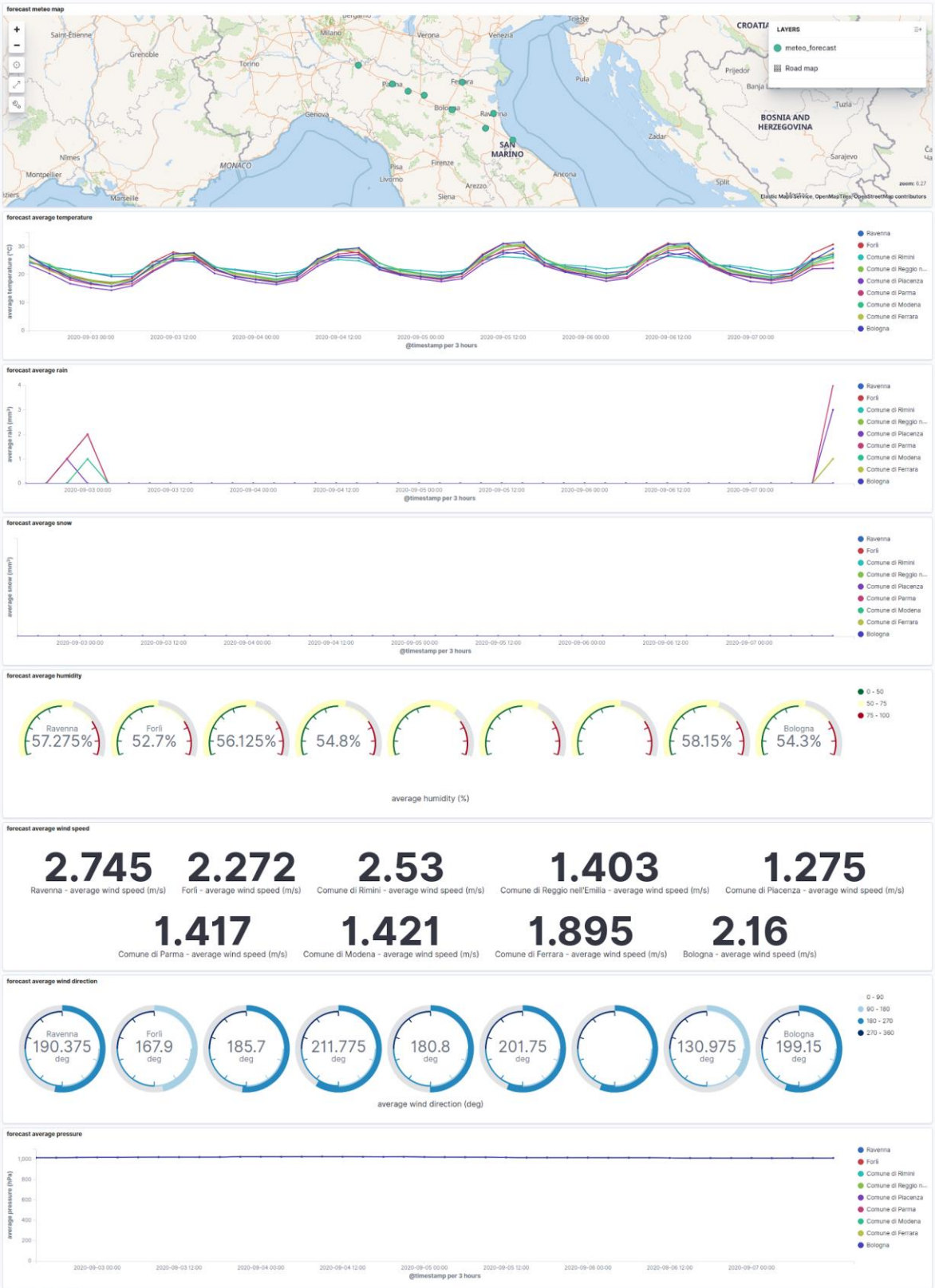

Fig. 6. Dashboard previsioni meteo in Kibana

3.7. Script realizzati

Di seguito si riportano tutti gli script costruiti ed utilizzati all'interno di questo progetto:

install-prerequisites.sh	Installa tutti i prerequisiti necessari per far funzionare gli script python per l'estrazione dei dati e per l'interfacciamento con PostgreSQL e Logstash
meteo_extracting_job.py	Estrae i dati meteo contenuti in un file .csv e li invia tramite protocollo TCP sulla porta 50000 a Logstash
snapshot_extracting_job.py	Estrae i dati delle FDM da un file .csv, li elabora e li formatta per poi salvarli all'interno di una tabella in PostgreSQL
create-index-pattern.sh	Script utile a creare un "index pattern" in Kibana utilizzando le sue API invece dell'interfaccia grafica
export-objects.sh	Estrae tutte le dashboard con le rispettive visualizzazioni da Kibana e le salva in un file .ndjson (formato necessario in quanto usato di default da Kibana) utilizzando le API di Kibana
import-objects.sh	Importa tutte le dashboard con le rispettive visualizzazioni a partire da un file .ndjson utilizzando le API di Kibana
CreateTable.sql	Script utile a generare una tabella in PostgreSQL direttamente al suo avvio

3.8. Esecuzione dei servizi

La cartella del progetto avrà il seguente contenuto:

```
- data-extractor
  - data
    - city.json
    - example-data_from01-01-20.csv
    - meteo-january-2020.csv
  - scripts
    - install-prerequisites.sh
    - meteo_extracting_job.py
```

```
    - snapshot_extracting_job.py
- kibana
  - data
    - export.ndjson
  - scripts
    - create-index-pattern.sh
    - export-objects.sh
    - import-objects.sh
- logstash
  - jdbc-drivers
    - postgresql-42.2.13.jar
  - pipeline
    - meteo-forecast.conf
    - meteo-historical.conf
    - meteo-nowcast.conf
    - postgres.conf
  - templates
    - location.json
  - pipelines.yml
- postgres
  - CreateTable.sql
- docker-compose.yml
```

A partire dalla root del progetto, l'esecuzione dei servizi è molto semplice con Docker Compose, che prevede l'utilizzo di un comando semplicissimo:

```
docker-compose -f "docker-compose.yml" up -d --build
```

Per l'installazione dei prerequisiti, basta eseguire:

```
bash data-extractor/scripts/install-prerequisites.sh
```

Per l'estrazione dei dati delle FDM dal file .csv ed il loro salvataggio in PostgreSQL si esegue:

```
python data-extractor/scripts/snapshot_extracting_job.py
```

Per l'estrazione dei dati meteo storici e il loro inoltro a Logstash si esegue:

```
python data-extractor/scripts/meteo_extracting_job.py
```

Per l'importazione di dashboard, visualizzazioni e index pattern in Kibana si esegue:

```
bash kibana/scripts/import-objects.sh
```

A questo punto sarà sufficiente collegarsi all'indirizzo http://localhost:5601 per accedere a Kibana e dalla sezione "Dashboard" presente nel menu drawer si arriverà infine alle visualizzazioni.