POLITECNICO DI TORINO

Master's Degree in Data Science and Engineering

Politecnico di Torino

Master's Degree Thesis

An end-to-end Data Engineering Architecture in the energy domain

Supervisors

Prof. Paolo GARZA

Prof. Cristina CORCHERO

Candidate

Michele APICELLA

October 2022

Summary

Globalisation, never-ending economic growth and natural resources exploitation have effects all over the world and in multiple fields. Some of them include climate crisis and geo-political tensions. These two phenomenons, with natural resources exploitation, have directly or indirectly consequences to the energy paradigm, with all of the actors involved, from energy producers, transporters, sellers and final consumers. It's clear how the energy paradigm is changing: energy transition and geo-political tensions are some of the causes of all the plans and responses that are putting into practice governments, companies and more in general the actors involved in the energy sector. Energy demand aggregation and flexibility recently have become a more and more reality in order to overcome all the problems that we may face in the following years, like reliable demand-response, which can handle big spikes in demand, mostly when energy is produced with renewable sources, and so it is more uncertain and difficult to handle. I had the opportunity to write this thesis at Bamboo Energy, a very young Spanish tech start-up that operates in this field. The purpose of this work is to illustrate how, after having understood the needs of the company in terms of data engineering, and end-to-end use case of data engineering can be built, covering four different areas of interest: data extraction transform, loading and aggregation pipelines, processes Orchestration and Alerting.

Acknowledgements

I would like to thank all Bamboo Energy team for this great opportunity: it has been a very formative experience, both personally and professionally. I've had the occasion to work side by side with amazing human beings with great knowledge in different fields. In particular a big thank to Cristina Corchero, Damien Tavan, Mattia Barbero, Josep Homs, Salvi Solà and Manel Sanmartì.
I would thank professor Paolo Garza, for his great availability, reliability and advice.
A special thank has to be done to my parents and my sister: without you all this would never have been possible, thank you for always having supported me and motivated.
I would also like to thank my relatives in general, especially my grandparents for having supported me in these years.
A big thank also has to be done my colleagues and my friends: thank you for all the moments shared together.
Finally, the greatest thank must be done to my girlfriend, Eleonora, who keeps inspiring me day by day.

Table of Contents

List of Tables

List of Figures

Listings

Acronyms

AI
artificial intelligence

DAG
directed acyclic graph

DB
database

ETL
Extract Transform Load

GCP
google cloud platform

ML
machine learning

VM
virtual machine

Chapter 1

Introduction

In recent years it has been very clear how great impact have had the consequences of a never stopping industrial, production and consumption growth. Since the industrial revolution huger and huger natural sources have been exploited to produce energy and to feed the mechanisms of a fast-growing world in terms of industrial production, technology, scientific growth and globalisation. It is undeniable that all this brought some incredible great advantages like medical progress, technological and knowledge access and general improvement of life conditions but clearly there is also the other side of the coin, which is more and more visible in the recent times. Despite some countries leaders continue to negate climate crisis facts there are fortunately a great scientific community and literature that agree [1] to the climate change crisis that is going on and continue to publish scientific articles and papers and to warn people that actions must be taken in order to reduce what seems a too big problem to resolve, especially in short times. Thanks to the scientific community, people, especially the young, have begun to externalize interest in this topic, like Fridays for future movement, and so have politicians and organizations which are now trying to plan politics and concrete actions.
The United Nations seriously started to reflect on this problem in 2015 with the adoption of the so-called Sustainable Development Goals (SDGs) whose purpose is "to end poverty, protect the planet and ensure that by 2030 all people enjoy peace and prosperity" [2]. Among the seventeen different SDGs, four of them are particularly significant concerning the scope of the working environment where this thesis work has been made, Bamboo Energy, a Spanish start-up involved in the sector of software products for energy assets management:

- Goal 7: Affordable and Clean Energy
- Goal 9: Industry, Innovation and Infrastructure
- Goal 12: Responsible Consumption and Production

- Goal 13: Climate Action

The Agenda 2030 and the SDGs are a good starting point for trying to resolve a global problem that can only be solved with great effort, unity, communion of interests of all the countries despite the great challenges and the differences among nations in terms of life quality, welfare level, industrial production, presence of resources and law system.
Focusing on European Union, climate change is a priority as its effects are particularly visible: it is already enough to say that 2020 was the warmest year recorded in Europe [3]. The majority of evidence indicates that this is due to the rise of greenhouse gas emissions (GHG) produced by human activity. In addition, the average temperature today is 0.95 to 1.20 °C higher than on the and of 19th. An increase of 2°C, with respect to pre-industrial revolution temperatures, is considered by international scientists as a threshold with dangerous and catastrophic consequences for climate and the environment: that's why the international community agrees that global warming must not exceed this threshold. The effects of climate change in Europe are multiple: lack of water, temperature increasing, forest fires, decreasing crop fields, biodiversity loss and people's health problems. Summing all this with the fact that the EU is the third biggest greenhouse gases emitter after China and the US, the EU is involved in international climate negotiations and, Under the Paris agreement, the EU committed in 2015 to cutting greenhouse gas emissions in the EU by at least 40 % below 1990 levels by 2030. In 2021, the target was changed to at least 55 % reduction by 2030 and climate neutrality by 2050. The Green Deal is the concrete answer, the EU roadmap to achieve the goal of climate neutrality, or zero net emissions, by 2050, and it's legally binding in the EU.
In this scenario, where there are very evident climate change problems, countries starting to take concrete steps and economic crisis with goods and energy prices keeping to increase, the wise use of technology can provide several and maybe small solutions that, if combined together, can really lead to problems solving or at least to mitigate the destructive consequences that already we are witnessing. And it is particularly in this scenario that Bamboo Energy, a brand new Spanish start-up set in Barcelona, is operating building software products that provide, with the use of Machine Learning, solutions to energy assets management with a data-driven approach. The idea behind this work is to focus on data engineering aspects of Bamboo Energy and propose improvements for a platform that is growing fast, resulting in a circular, from start to end, data cycle use case: ingesting data from IoT sources, ETL, computational pipelines and orchestration. The goal is to describe high-level processes of the start-up, make a snapshot of the data engineering situation when I joined the company, in November 2021, and to propose improvements according to product and business requirements.

1.1 Bamboo Energy Platform

Bamboo Energy is a brand new start-up born in the Spring 2021 as a side project of IREC, Fundación Instituto de Investigación de la Energía de Cataluña, founded by Mattia Barbero, Cristina Corchero and Manel Sanmartì. It is a tech start up which provides data-driven solutions to different actors involved in the energy sector. As climate change has been targeted as a priority problem in the EU and concrete actions have been taken, like for example energy transition acceleration, there is a lot of work to do and a lot of strategies to explore to provide solutions in order to improve how energy is managed. This is a tough aspect because the energy sector is very complex and a lot of different actors are involved: companies that produce energy, that transport it, energy retailers company and final consumers. Technology can lead to products that can help provide solutions to these problems and it is here that Bamboo Energy is investigating and proposing data-driven solutions: Bamboo Energy Platform.

1.1.1 High level process overview

The energy transition is a reality and the EU is investing a lot in this kind of project and is allocating funding. Climate change, concrete actions to respect EU directives, energy markets volatility, electric net instability, technology evolution and more consciousness are modifying the paradigm of energy systems. The current changes in Spain and in the rest of Europe are creating a electronic system more open and with the opportunity to offer new services. We can define four tendencies of the energy sector of the future:

- Climate change:
  - Energy transition
  - Great penetration of renewable energies
- Electric system decentralization :
  - Self-consumption
  - Local generation and conservation
- Active consume:
  - Energy communities and local markets
  - New legislation
- Digitization :

  - Smart nets
  - Internet of Things

Climate change makes the weather more unpredictable and consequently, with the great penetration of renewable energies for the energy transition which depends a lot on the weather, electric reds are more unstable and also the energy market become more volatile. The paradigm is also changing, going to be decentralized, because consumers are now beginning to self-consume and conserve energy: more and more buildings are adopting photovoltaic panels and batteries to store energy, making the consumer more conscious and more active in the energy market. New legislation and formations of energy communities and local energy markets are changing the relationship in this sector and making the electric consumers more active and more participated, from a passive perspective of a consumer that just consumes electricity and nothing else. Digitization and technology in general are creating smarter nets and smarter buildings whose assets connectivity and management are becoming easier due to the Internet of Things sector that is expanding. Combining the Sustainable Development Goals and the tendencies of the energetic future, Bamboo Energy has three main objectives:

- Accelerating energy transition
- Make demand aggregation a reality
- Positioning the consumer at the centre of energetic system

Bamboo Energy mainly provides, among other things, three core services: baseline forecast, flexibility forecast and bidding optimization. Let's imagine a modern building with different energy assets. Baseline forecast is, briefly, the service that makes forecast of the amount of energy a certain asset will consume at a certain time: the asset can be anything from heating, to ventilation to air conditioning, from battery to photovoltaic panels and industrial machines in general. Flexibility forecast is the service that makes forecast of how much a certain type of asset can increase or decrease its energy consumption without affecting the overall process like for example the capacity of an air conditioner to increase or decrease its electric consumption respecting temperature constraints and keeping the environment chill. Bidding optimization is the third step that combines all the previous forecasts and translates them, solving optimization problems, into optimized buying and selling energy with the participation in different energy markets.
Accelerating energy transition is one of the objectives of Bamboo in fact, due to the fact that renewable energies depend more on the weather and so it is more difficult to forecast making the electric net more unstable, flexibility forecast is a good answer to the issues coming from a net that is more unstable: the capacity of increase and decrease consumption, always respecting constraints and thresholds,

is a good weapon and a good tool in order to be flexible, to avoid energy waste, to optimize energy flows and to adapt in a more unstable world.
To this purpose, another goal of Bamboo is making demand aggregation a reality. Demand aggregation is the aggregation of different clients inside the electric system like consumers, suppliers, prosumers so that they act as a single entity when they participate in energy markets on when services to system operators are sold. In other words is the aggregation of different clients like consumers, retailers, distributors(DSO) and transporters(TSO) and suppliers so that they participate in energy markets in order to gain a common benefit. Flexibility is strictly connected to demand aggregation: with aggregated energetic needs of different actors involved it is easier to manage energy flows and energy buying/selling and the participants of this aggregation can vary in certain moments their consumes and productions for economic retribution. This is made easier and more flexible when assets like batteries and photovoltaic panels are installed. Demand aggregation, flexibility, self-energy generation and conservation are moving the passive figure of an energy consumer of the past, who just connected his assets to the net and consume energy, to an active actor that does not just consume energy, but can vary its electric consumption up and down and can inject to the net energy generated or saved. The consumer is now becoming a prosumer and is more conscious and active in the energy market. The new figure that is coming out from the new energy paradigm, despite the consumer who is more active, is the demand aggregator: this new figure can have a big role in the new energy paradigm and has a good potential in relation of the Sustainable Development Goals and the Green Deal. Bamboo Energy offers a platform that manages the aggregations of different actors in the energy market, and it's specialized in demand aggregation of energy retailer companies, managing the flexibility of their portfolios.

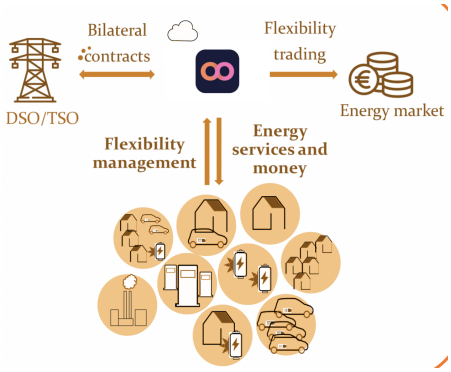

Figure 1.1: Bamboo positioning in energy sector

The following section focus more in details on the core services provided.

1.1.2 Baseline Forecast

Baseline forecast module makes forecast of baseline consumption of the different assets, which is the consumption that these installations have if there aren't any flexibility activation, utilizing Kernel Regression o KNN Regressor. Let's imagine a building with different devices installed: baseline module can be used to predict the baseline of the different installations like heaters, ventilators and air condition (HVAC), PV generations and battery baseline. This module can also make the meter forecast, which represents the expected energy injected or absorbed from the grid. This amount of energy in the one that the System Operator measures and how much the final costumers pay. The meter forecast is difficult to predict by himself, since it depend on different factors, such as the total consumption of the site, the on-site generation behind the meter (self-consumption strategy) and the battery usage. For this reason, Bamboo calculates in a second phase the expected meter consumption using the following logic:

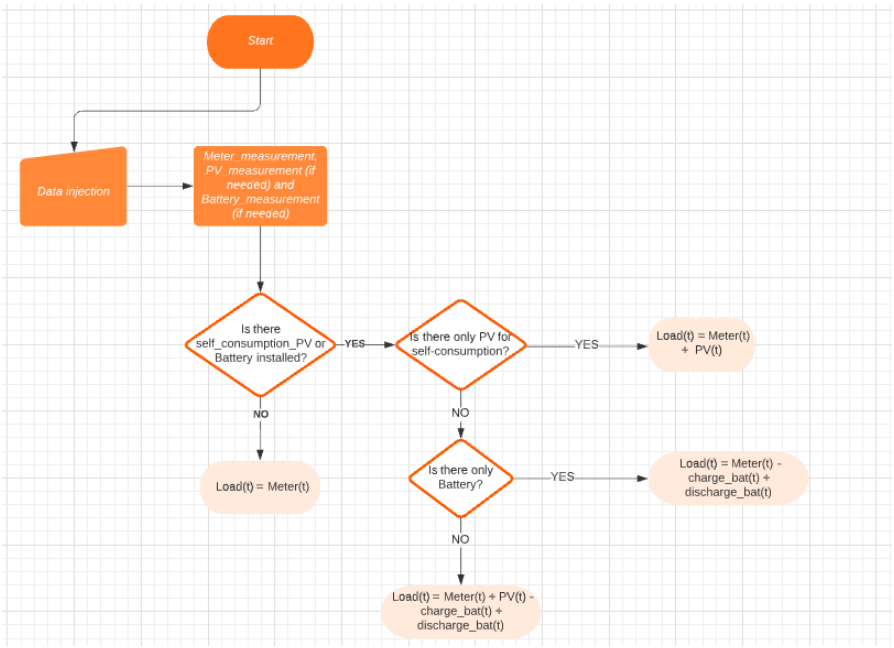

Figure 1.2: meter forecast

The baseline forecast module can be run or in day-ahead mode (da) or in intraday mode (id). The day-ahead mode makes a prediction in the day D for the 24 hours (or 96 quarter of hour) of the next day D+1. The intraday mode makes a prediction of the consumption in the day D for the next x hours of the same day D, where x is a customized parameter. Different source of data are used to feed these algorithms and the most important ones are:

- Historical consumption of the installation: power measures
- Weather forecast of the weather station close to the site

The target data to predict with Kernel regression or KNN regressor are of course power measures of the device of interest in the period of interest. This module is the foundation part of Bamboo process because it makes forecasts about how much energy a certain asset will consume not thinking about flexibility that will be use to make flexibility forecast.

1.1.3 Flexibility Forecast

The flexibility forecast module is used to estimate the flexibility available of the different installations, which means their capacity to change their baseline consumption if needed.The flexibility forecast module can be run or in Day ahead mode (da) or in Intraday mode (id). The day ahead mode makes a prediction in the day D for the 24 hours (or 96 quarter of hour) of the next day D+1. The intraday mode makes a prediction in the day D for the next hours of the same day D. Flexibility forecast module is the core part of demand aggregation making customers evolve from a passive condition to a more active and aware. Flexibility forecast are made solving optimization problems, making use of RC model which abstracts a physical building with its constraints. The module functionality is divided into training and forecast: in the training phase, whose source of data, among others are, building microgrid information, weather and assets measurements, physical parameters are estimated. After that, during the forecast phase, the model makes use of the before estimated physical parameters to compute the flexibility forecast of a particular device, which is often a device belonging to the HVAC category (Heating, Ventiltion, Air conditioning), using as input source data, among others, weather data and assets measures. Flexibility forecast are divided into two category:

- **Flexibility Ramp-up**: It is the amount of energy consumption to decrease over a time period
- **Flexibility Ramp-down**: Amount of energy consumption to increase over a time period.

1.1.4 Bidding optimization

Bidding Optimization module is in development phase and condenses baseline forecast, flexibility forecast modules and market prices to guide flexumers into concrete actions, in terms of decreasing or increasing particular assets consumption

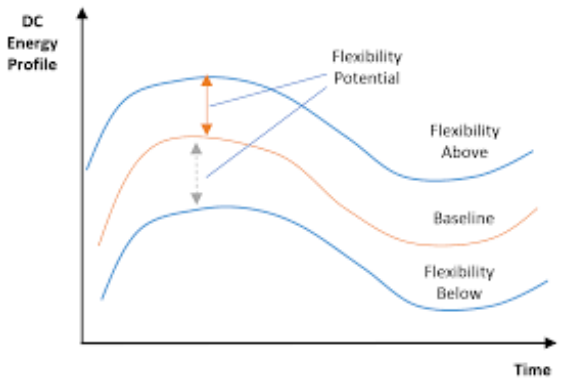

Figure 1.3: Flexibility up/down graph of a Data Center building

in a clever way in order to well participate in the different flexibility markets, especially tertiary markets, in order save energy, save money but without affecting industrial process constraints. .

1.2 Goals, motivation and methodology

This thesis work is intended as a practical case study in the field of Data Engineering related to young start-up operating in the Energy-Tech sector. The goal is to create an end-to-end data engineering cycle, from data extracting, transform, loading and data architecture planning to processes analysis, orchestration, monitoring and notifications, that fits Bamboo current and future needs. In order to have the instruments to carry on this thesis, the work can be subdivided into three phases:

- **Understanding**: first step is to deeply study and understand Bamboo data engineering environment, how it is structured, which technologies are used and for which purpose, which are the features implemented and the one planned or missing.
- **Exploration**: once information has been collected and needs, strengths and weaknesses identified, multiple paths are open to be explored in order to provide reliable ad hoc solutions
- **Implementation**: once technologies are compared and path chosen, solutions are practically implemented.

The motivation behind the choice of this particular field and case study can be found in the point of conjunction of two topics I'm interested in: engineering and environment. More specifically I'm interested in real worldly spread problems and how possible solutions could be found. Nowadays there are a lot of technology

companies who put a lot of effort in scientific/technological research achieving incredible results providing surprising products with great AI/ML applications, but to many times these efforts are strictly business oriented in fields that compared to others, result a little silly and superficial. I think as human beings we are crossing a line for which, conscious of the incredible instruments available, it would be great to see more coordination of companies,governments,organizations efforts to provide reliable solutions to worldly spread problem that are becoming more and more relevant. Bamboo Energy is an attempt to provide reliable solutions especially for energy demand aggregation and also for the consequences and implications of this topic and energy management optimization in general. I've been lucky to have the opportunity to apply concepts learned during the Msc in Data Science and Engineering and to study, explore and implement interesting new technologies for a start-up operating in a field in which I find motivation.
Regarding the methodology used to carry on this thesis work a statement must be made: this thesis is not intended as a pure theoretical research work, but instead a very practical one, touching different topics, each once of them has its own characteristics, state of arts technologies and standards. Thus, regarding the three conceptual phases, understanding, exploration and implementation, way of thinking and solutions research, comparison and implementation, have been made taking into account firstly that Bamboo Energy is new start-up that is growing and in phase of construction, where a lot of things change very fast. This work is not a specific field research study where a personal method is compared with specific state of art where there are precise and well defined metrics to be compared but, instead, a multi-topic research whose purpose is tho build a working, scalable and reliable end-end data engineering architecture implemented taken into account business needs and constraints, compatibility with already built products, the ones planned or in phase of development, current, future clients, cloud environment and platform architecture, and cost of implementation and scalability.

Chapter 2

Bamboo Data Engineering snapshot

This section is an overview of the Data Engineering situation at the time of my arrival in the start-up (November 2021), basically focusing on Data Architecture, Platform Architecture, ETL pipelines, processes Orchestration and Cloud Environment. Lots of aspects have changed from that point: the following chapters describe some modifications and new paths explored to improve already implemented processes some processes and brand-new ad-hoc solutions to address specific product or business requirements.
This section is the starting point of all this thesis work: it is necessary to well understand the company processes, data infrastructure, data flows, ETL pipelines, Cloud environment, platform architecture, product requirements and difficulties in order to correctly propose improvements or new solutions.
Moving between high quality company documentation, meetings with the product team, studying of processes and focus sessions I have been able to get a deep overview of Bamboo Data Engineering situation and to build a solid base to make the following steps.

2.1 Data architecture

This section focuses on Bamboo Energy Platform data infrastructure, more specifically on the kind of databases chosen to address a specific question in relation of the different types of data that need to be stored.

2.1.1 Time series Database: InfluxDB

In terms of data types, time-series data are the most important for Bamboo Energy: they are the core of everything, the main sources for baseline, flexibility forecasts and bidding provision algorithms. Under this group we can identify three main time-series data: power measurements coming from flexumers assets, baseline or flexibility power forecasts, energy market prices data and weather forecast data. The database chosen for handle and storage these time-series data is of course a time series database (TSDB), which is a particular kind of database, optimized for storage and querying time series data in relation to time and values.
The database chosen for this purpose is **InfluxDB**, developed by InfluxData. It is a non relational, no SQL database, written in Go and it is well optimized for storing in a high-performing engine millions of data per seconds coming from the most different sources and for making high-availability query with different level of aggregation. Is has its own query language, called Flux, specifically built for time-series data, and its SQL-like alternative, FluxQL, is s SQL version of Flux. It is also possible to used these two languages to query SQL traditional databases.
In order to have an overview of the data-types managed in InfluxDB, the followings are snippets from the database and some concrete query examples, more specifically related to devices measurements which come from an IoT source, baseline/flexibility forecasts and market prices, which are among the most important times-series data for Bamboo.
The following is a typical sample of query result through Bamboo REST API, encoded as a Json file, of measurements of a particular device of a site, coming from an Iot source: we can identify the fields that identify the site and the device and in the "measurements" field it is actually included the measurements time series of power, with the field "quality" indicating the goodness of the sent data.

Listing 2.1: time-series data: measurements sample

```
[{
    "site_id": 10038,
    "device_type": "meters",
    "device_name": "meter",
    "measurements": [
      {
        "time": "2022-05-01T00:00:00+00:00",
        "quality": true,
        "power": 24
      },
      {
        "time": "2022-05-01T00:15:00+00:00",
        "quality": true,
        "power": 12
      },
```

```
16        {
17          "time": "2022-05-01T00:30:00+00:00",
18          "quality": true,
19          "power": 12
20        },
21        }]
```

A typical queried time-series of baseline forecast of a particular device of a site looks like the following snippet, the fields are just two and indicate the timestamp and the power forecast.

Listing 2.2: time-series data: baseline forecast sample

```
1  [
2    {
3      "time": "2022-05-29T14:30:00+00:00",
4      "power": 11.262
5    },
6    {
7      "time": "2022-05-29T14:45:00+00:00",
8      "power": 6.92
9    },
10   {
11     "time": "2022-05-29T15:00:00+00:00",
12     "power": 13.692
13   },
14     }]
```

In order to fuse baseline and flexibility forecasts to make good bidding offers and to make the flexumers be able to well participate in flexibility markets, the time-series related to market price are another important source of data. A typical query of market data time-series looks like the following snippet.

Listing 2.3: time-series data: market price sample

```
1  [
2    {
3      "time": "2021-05-01T00:00:00+00:00",
4      "energy_price": 0.84
5    },
6    {
7      "time": "2021-05-01T01:00:00+00:00",
8      "energy_price": 0.72
9    }
10 ]
```

One of the data sources of baseline forecast algorithms is weather forecast data. Instead of showing the result of the query through the API, the following image in a snapshot of the UI of InfluxDB showing a sample humidity time series of a weather station in Malaga:

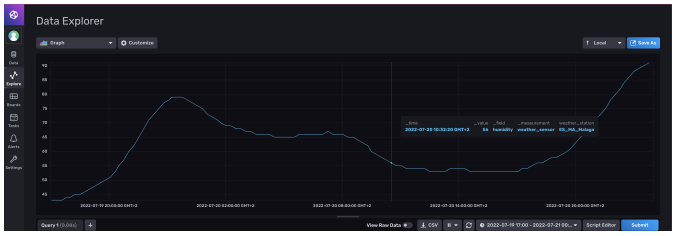

Figure 2.1: Bamboo positioning in energy sector

2.1.2 Relational Database: PostgreSQL

On the other side, where there is no need to use time-series specific databases, a traditional relational database is mounted to store data related to flexumers, their assets, the markets in which they participate and other relational information. Once the data model related to flexumers has been built the solution for this implementation currently used is PostgreSQL, which is an open-source object-relational ACID-compliant database. It is one of the most popular and widely used relational database and extends the SQL language combined with many features that safely store and scale the most complicated data workloads.

2.2 High level architecture overview

The following section is an overview of the Bamboo Energy Platform architecture, a snapshot of the modular structure at the time when I joined the company, briefly describing what functions and services the different modules address, where they are hosted and how they are connected.

2.2.1 Bamboo REST API

The Bamboo REST API is one of the core parts of Bamboo Energy Platform and it is used both by flexumers and Bamboo team for lots of different purposes. Its main function, as REST API, is to add an abstraction layer for writing and reading operations to and from the relational and time-series database: in this way all the modules, which are hosted in different Google Cloud Platform products, do not read or write directly from and to the databases but use Bamboo API as a standardization layer in order to be sure to make write and read operation in the same, well defined way. It has the central role of the architecture and it has been first part of the platform to be built. It has applications in every aspect of the overall processes of Bamboo Energy: through API requests, Bamboo provided services can be executed and general read/write operations can be performed, from

baseline and flexibility forecast, from energy market data to weather forecast, from bidding optimization to flexumers specific information and many more.

It provides access to flexibility assets managed by Bamboo Energy and to a lot of different services: for example through the API it is possible to create and list flexibility sites and assets, obtain activation for specific assets, post and get measurements for specific assets or make asset devices baseline and flexibility forecasts. In order to give users and flexumers a unified tool to interact with the API, a python client has been developed and it is available on pypi: it's widely used both internally for modules developing and externally by flexumers.

2.2.2 Cloud Environment

Nowadays more and more companies, especially start-ups, build their architecture or have in plan to migrate the ones already built to the cloud for a lot of different reasons and Bamboo Energy is not an exception: with the usage of cloud technologies it is faster to develop products and services and also it is cheaper and easier to maintain respect to physical servers. The Cloud infrastructure market is split between three major actors which provide a lot of different services related to Big Data, computation, data managements a lots more: Amazon AWS, Microsoft Azure and Google Cloud Platform. Bamboo Energy Platform is totally hosted on Google Cloud Platform (GCP) and satisfying specific constraints has access to credit to be spent on GCP products. A lot has changed in the cloud architecture from the time of my arrival in the company: the following subsections make a snapshot on the GCP products used at that time.

Cloud Run

Following the definition given by Google in its official documentation [4], Cloud Run is a managed compute platform that enables users to run containers that are invocable via requests or events. It takes a Docker container image and runs it in a stateless, autoscaling HTTP service allowing to run arbitrary applications serving multiple endpoints, not just small functions with a specific interface like the ones that could be developed through Cloud Function, a tool described in the following sections. Cloud Run allows users to write their script in the programming language they want and then to push it and package it as a container with Cloud Build. It's able to be be configured to support multiple concurrent requests on a single container instance.
Combining the great strength of Docker, which is to building software in packages called containers, separating application from infrastructure using OS-level virtualization, and the solidity of Cloud Run it is very useful and flexible to build applications through this tool. The resources needed to run a Cloud Run instance

are provided just in the moment when the request is made: after they shut down. The mounted Cloud Run instances are the following:

- **backend**: it's the container of Bamboo Flexibility API which provides access to flexibility assets managed by Bamboo Energy allows to make requests for a lot different purposes like for example create and list flexibility sites and assets, obtain activation for specific assets, post and get measurements for specific assets, list energy market prices and manage portfolios.
- **baseline**: it's the container of the Bamboo Flexibility API module related to baseline whose endpoints are used to train baseline assets models and to make baseline forecast for a specific asset.
- **flexibility**: it's the container of Bamboo Flexibility API module related to flexibility allowing to train flexibility thermal zones and to make flexibility forecast of thermal zones, batteries and shiftable load.

Cloud Function

Reporting the definition provided by Google in the official documentation [5], Cloud Functions is a scalable, pay-as-you-go functions as a service (Faas) product to help users build and connect event driven services with simple, single, purpose code. It's a server-less platform that supports individual services and can be called via HTTP request of can be triggered based on background events. The code requires to be packaged as a function and the cloud function component handles the packaging, deployment and execution of the code. It can only handle one request at time for each instance. The resources needed to run a Cloud Function are provided just in the moment when the function is called: after they shut down.
The mounted Cloud Function instances are the following:

- **mvp-market-prices**: it's the function used to make request to ESIOS API to download energy prices of the different energy market of interest. A more detailed explanation of this process is explained in the following sections.
- **mvp-weather-forecast**: it's the function used to download weather forecast data from tomorrow io API releted to the weather stations of interest

Cloud Storage

According to the definition provided in the official documentation [6], Cloud Storage is a service for storing objects in GCP. An object is an immutable piece of data consisting of a file of any format. Users store objects in containers called buckets. All buckets are associated with a project and projects can be grouped under an

organization. It is mainly used for storage ordinary data which are not main source of the algorithms, models build after training phase and generic tests result.

Compute Engine

According to the definition provided in the official documentation [7], Compute Engine is a computing and hosting service that lets users create and run virtual machines on Google infrastructure. Compute Engine offers scale, performance, and value that lets users easily launch large compute clusters on Google's infrastructure. Resources provided for the usage of a Compute Engine are always running, differently from Cloud Run and Cloud Functions.
The mounted Compute engine instances are the following:

- influxdb: this is an instance of Time Series database InfluxDB used to store the multi-type time series data describer at the beginning of the chapter.

Cloud SQL

According to the definition provided in the official documentation [8], Cloud SQL provides a cloud-based alternative to local MySQL, PostgresSQL, and SQL Server databases. It's a product built to allow users to spend lees time managing the the database and more using it. A lot of applications running on Compute Engine, App Engine and others services in Google Cloud use Cloud SQL for database storage.
The mounted Cloud SQL instances are the following:

- postgres: this instance of PostgreSQL is used as relational database for every data that is not a time series and can be modeled with the relational traditional model

Cloud Scheduler

According to the definition provided in the official documentation [9], Cloud Scheduler allows users do create cron jobs that are units of work whose execution can be scheduled at defined times or regular intervals. Each cron job created is sent to a target according to a specified schedule, where the work for the task is accomplished. The target must be one of the following types:

- Publicly available HTTP/S endpoints
- Pub/Sub topics
- App Engine HTTP/S applications

The mounted jobs are related to tests of baseline and flexibility algorithms on concrete sites or invented (for testing purposes), and to the automatic scheduling of the downloads weather and market data.

2.3 Dataflows and ETL

This sections focuses on the different data flows passing through the platform, the ETL pipelines already implemented and the automation of processes with orchestration tools.

2.3.1 Pilot projects: measurements data

As explained in the introduction the most important dataflows are the ones associated with the devices consumption measurements, which are supposed to flow every day with frequency that can be very high, depending on the specific device and the service of interest. The correct ingestion, processing and usage of these data, according to other external data sources and the choice of the algorithms of interest, could give Bamboo Energy a great competitive advantage. Data flows of asset devices measurements should be solid and reliable as they represent the core data source of Bamboo forecast algorithms. These data, which are time-series of energy or power measurements, are ingested in InfluxDB, the database chosen for time-series data storage, as described in the previous section. The following section focuses in a deeper way on how is handled and managed the current active asset devices measurements dataflows.
Bamboo start-up is very young, was born during Covid-19 pandemic in Barcelona and, at the time of writing, has not yet a huge production environment: product team is currently working day by day to construct and improve the platform. The minimum viable product currently built is active and used in synergy with some energy demand aggregators, energy sales companies, (*comercializadoras*) having great interests in energy demand aggregation, baseline forecast and flexibility, and some of their big clients who are basically industries to which *comercializadoras* sell energy and so they have too a strong interest in flexibility, both in terms of energy and money saving.
Currently, at the time of writing, there are two mounted pilot projects which are sending data of building devices. These pilot projects are made in synergy with European project whose purpose is to is establish different collaboration schemes between transmission system operators (TSOs), distribution system operators (DSOs) and consumers to contribute to the development of a smart, secure and more resilient energy system and, and an energy retailer company and one of its clients.

Extraction

The ingestion of the data flows consists of the following steps: when there is a new device measurement, a Bamboo API call is made. More specifically, a PUT

request stores new raw device measurement data in InfluxDB. Currently the only way to communicate between customer-side and Bamboo-side is through Bamboo API: this is the only available bridge to ingest data, and this approach brings some advantages but also some disadvantages, as explained later. Thinking about ETL pipeline concept, the two steps, which actually are just one API call, can be considered as single phase where Extract and Load steps are fused in a single API call: device measure is "extracted" and directly loaded to the database. In this sense, there is no actually a Transform phase between Extraction and Loading and moreover, there is no currently a Database, or a section of it, where clean data are actually stored, but data, when requested for some algorithms usage, are cleaned and processed "on demand" for the purpose of interest: clean data are not stored persistently anywhere, they are "thrown away" after having been loaded and processed for a specific usage (E.g: a ML algorithm launch) through other Bamboo API calls. Having explained how data is extracted and loaded in the Database, how is the process of extraction of these raw data in order to make them available for processing and future usage? When data is requested to InfluxDB, a GET request is made through Bamboo API asking the database to make a query in Flux language, enabling to specify start and end date of the data interval interested, with some other filters, selections and aggregation options available. After the Flux query is made, data is returned with the correct time interval but it is still raw, with no control or cleaning processing before persistent storage in the Database.

Transform and loading

How does Bamboo currently deal with cleaning and processing data to feed to his algorithms when they are requested ? As said, when a Bamboo API call is made, for example, for launching some ML algorithms, data requested from InfluxDB is returned as a result of a Flux query, computed within the InfluxDB database. In order to handle in a better and more manageable way and to perform cleaning processing on data, this returned Flux object is converted to a Pandas dataframe in order to have more flexibility and precision in data processing. The cleaning and processing phase consists of basically three steps: outlier detection, interpolation and up/down sampling This approach can be defined as an on demand batch processing whose final outcome is not stored anywhere after usage. The following paragraphs focus more deeply on the cleaning and processing techniques used.

Outlier detection Given a returned time series we are interested in understanding if any measurement could potentially be an outlier and how to deal with these found outliers. A value is flagged as an outlier if its Z-score has a value greater than a certain threshold; a common threshold value for data based on a normal distribution is 3. This approach evaluating Z-score as a metric for detecting outliers

can obviously be done only in batch processing. Values marked as outliers are replaced with Nan in order to be interpolated later, if it is the case. The purpose of this is to detect an outlier in batch and convert it to a Nan for future interpolation if it is needed.

Interpolation Once outliers are detected and replaced with Nan what is the approach used in order to deal with these gaps in data ? Interpolation is performed only for gaps with a timespan less than an interpolation threshold, while it is not performed for those gaps with a timespan bigger than a threshold. The interpolation technique used internally is the one provided by Pandas Dataframe class with method 'linear'. The purpose of this process is to fill little gaps with Nan value with plausible and close to reality data, interpolated with the close ones. This, related also to outlier question, is a tough question especially regarding time series data. When measures arrive as outliers or as NaNs a trade-off between interpolation, and so "invent" data and use the real available data must be made.

Up/down sampling Once a time series has been processed in order to have outliers detected, substituted, and with gaps interpolated, we are interested in knowing if the time series could be up or down sampled in order to handle and use these data with a lower or greater granularity. With this purpose some checks are performed in order to verify if the pre-processed Dataframe could be up or down sampled. This up/down sampling could be also performed in the Flux query, but doing it in Pandas gives more control and reliability to the final processed Dataframe because it is not so unusual that during the conversion from Flux object to Pandas Dataframe some additional NaNs are returned in the Dataframe and so interpolation should be re-performed again, following this way. The purpose of doing this up/down sampling it to give more freedom on the granularity requested when forecast algorithms are performed. For example, let suppose that we want to forecast the baseline consumption of the devices of a particular building, for the next day, every quarter of hour: the data loaded for training the ML models are requested with this frequency and so raw data queried are up or down sampled for this purpose.

2.3.2 Weather data flow

Weather data are a very important data source for Bamboo Energy Platform, used to make baseline and, consequently flexibility Ml algorithms. Weather forecast data of the stations of interest are downloaded from TomorrowIO Weather API, which provides reliable and accurate weather data. The stations of interest are the ones in proximity of the flexumers. Weather forecast of a particular weather station are downloaded through a Cloud Functions instance which takes as input parameters

data about the weather station, time frequency, start time and end time of the period of interest and stores in InfluxDB the following weather data forecast:

- temperature
- dewPoint
- humidity
- pressure
- irradiance
- windspeed

The usage of TomorrowIO API has some constraints, for example cannot be made requests of weather forecasts with a start date of more than six hours in the past, with certain frequencies the period of interest has a maximum length and there is a limited number of request that can be done in a certain time. This constraints are related of the free version of TomorrowIO API, whose use is experimental. As data from TomorrrowIO are accurate there is no preprocessing of data downloaded: they are stored directly in InfluxDB through the use of Bamboo REST API, with a Cloud Functions instance.

2.3.3 Market data flow

Market data are another important data source for Bamboo Energy Platform, whose usage is related to flexibility and bidding services. Prices of different energy or flexibility markets are downloaded from ESIOS API, which is API to use to download data related to the processes of the operations in the electric system who are responsibilities of the Red Eléctrica de España, the Spanish national electrical net. There is actually no preprocessing of the downloaded data: as weather data, they are stored in InfluxDB through Bamboo REST API, with a Cloud Functions instance that receives as input parameters data about the market, the particular price series, start and end time.
The market prices are related to different market:

- daily energy market
- intra-daily energy market
- intra-daily continous energy market
- secundary reserve flexibility market
- terciary reserve flexibility market

2.3.4 Orchestration

Orchestration is very important in Data Engineering. Making reliable data pipelines, that start and stop in an automatic and scheduled way and, furthermore, being able to collect good and precise logs data about the pipelines in order to rapidly and efficiently inspect their behaviors and functioning, making easier problems handling, is one of the keys of success of every data-driven company: having the complete control and freedom of scheduling what and when and at which conditions.
There are not scheduled and orchestrated data pipelines, in the sense that there are not scheduled ETL data jobs which regularly perform some actions on data sources, for the reasons explained above: the approach is on demand, there is nothing scheduled in time in this sense.
What is scheduled at precise times or specified intervals are Cloud Functions instances, like for weather forecast or market prices, or others Bamboo API modules like baseline or flexibility forecast with the usage of Google Scheduler.
To illustrate the Orchestration overview let's make an example of a very small baseline forecast use case. Let's suppose for a given building, that has three devices, that the baseline forecasts must be done with respect to every device, every day at a certain hour. The approach used is making scheduled baseline forecasts Bamboo API calls for every device at the given time and date with Google scheduler: every API call is scheduled in a different Google scheduler job, with its own trigger frequency. To summarize this approach, connected to what is said in the previous sections about ETL pipelines, this is what happens when for example the baseline forecast module is launched for the different devices of a given site. Let's assume that dummy site has four devices for which the baseline forecast module must be requested through Bamboo API:

- load (overall electric load of the building)
- hvac1 (thermal load)
- hvac2 (thermal load)
- battery

These devices have different dependencies:

- load, hvac1, hvac2 baseline forecasts can be launched in parallel, at the same time
- battery baseline forecast must be launched after load, hvac1, hvac2 forecasts

Four Google Scheduler jobs are created, one for each device:

- load, hvac1, hvac2 have the same trigger scheduling time in order to launch these forecast in parallel

- battery has ten minutes later trigger scheduling time in order to be launched after load, hvac1, hvac2 devices

Every time a baseline forecast module endpoint is triggered through Google Scheduler, and some data loading is requested for the algorithms, it is performed all the data processing described in the above sections: data is loaded and processed on the demand just for the usage and cleaned and processed data is not stored anywhere. The issue of storing or not processed data and how, in the DB or in other solutions, is a tough question in this particular sector in which Bamboo Energy operates. Obviously it is a good idea to store clean data, but everything depends on the kind of preprocessing, if the data sources are bacth or streaming, if it is the case of ingesting raw data and than doing preprocessing in a second moment, the amount of data, how many times these data are needed, how fast data become obsolete, the cost of the cost of the ingesting, processing and querying..
The pros of the usage of Google Scheduler is that it is integrated in Google Cloud Platform, making communication with other components of the platform very easy, like Cloud Run or Cloud Functions in this case. The principal con is that it does not provide the feature to make dependencies between jobs and trigger rules like "run this job even if that other job has failed" or "do not run this job if this other job has been successfully launched". Another is that there is not a clear and intuitive logs inspecting feature: logs about jobs can be inspected but it is not so well made for processes having multiple scheduled jobs which should have more or less strict dependencies and trigger rules between them.

Chapter 3

ETL, Data Aggregation pipelines, Orchestration and Alerting solutions exploration

After having deeply examined Bamboo Energy ways of handling extraction, transform, loading pipelines, orchestration management and alerting processes, this chapter focuses on the exploration of possible solutions that could improve already existing features or add new ones. The suggestion of possible paths to implement is outcome of the comparison of different technologies and programming models. Following the methodology criteria explained in the introduction, for every field of interest, after having identified features needs and improvements points, as the outcome of comparison, a specific technology and programming model has been identified in order to respect product constraints and requirements, exploring different approaches and taking suggestions from [10] and [11].

3.1 ETL

This section is intended to be the starting point to the research part related to data ingestion, transformation and loading especially for data sources coming from industry assets, basically power measurements.
In order to clarify, all the research work related to this topic covers all the data flows and processing from the point of ingress in Bamboo environment: the data flow of from the physical asset to Bamboo, which falls under the topic of IOT connectivity and often is carried on by the client itself or a thir-part company is

outer the scope of this thesis.
First, let resume the current approach used to carry on ETL processes: currently the only way way to communicate and send data to Bamboo is through Bamboo API, which was built, among other reasons, to have a standard to write and read data. With a POST HTTP Bamboo API call a raw asset measure is stored in InfluxDB: the specific module that needs data will again utilize Bamboo API to read data and to perform the specific processing requested for launching the algorithm. This approach has some strengths, that can be summarized as the follows:

- Standard way to write and read data through Bamboo API
- Real-time data ingesting
- As data are stored raw, each module can custom processing methods

This approach works well, especially in a limited production environment, but could have some limits and issues taking into consideration API load, scalability and compatibility. As before three main strengths have been identified, let's figure out what can be the weak perspectives of this approach:

- with a lot of assets sending data, Bamboo API can be overloaded and suffer a big production environment, carrying on ingestion and processing every time data are requested, even if the Cloud Run instance hosting Bamboo backend can obviously can be vertically scaled.
- Clients can only communicate through HTTP
- Processed clean data are not persistently stored anywhere

This analysis lays the foundation to the exploration phase: pilot projects data flows and processing gave the possibility to highlight the pros and cons of the current implementation. The main idea behind laying the foundations of possible new solutions is to keep the current strengths as the core part on top of which build something new and reliable, like for example the usage of Bamboo API as validation layer for writing data and for reading when some data are requested by some algorithm, and to try to mitigate the negative implications of the actual architecture, like the absence of clean data storage, the HTTP locking effect and the usage of Bamboo API for everything related to ingestion and ETL.
Before identifying the correct technology and programming model, able to satisfying requested special needs, the key features of the ETL pipeline of interest, related to assets measurements, must be correctly identified in order to not go out of scope and to have well defined constraints and requirements to follow.
After having taken into consideration the strengths and the weakness points of the

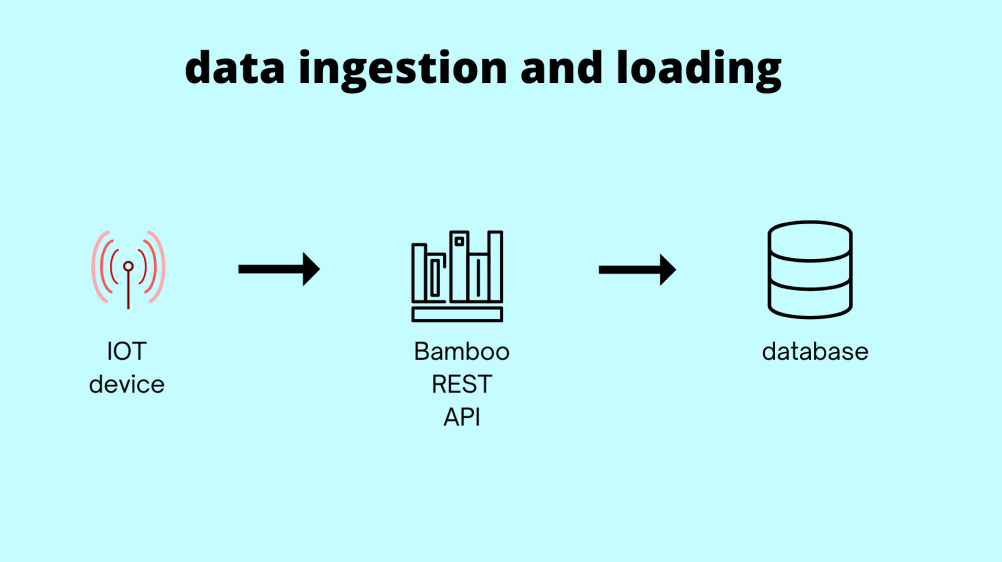

Figure 3.1: data ingestion and loading current implementation

actual implemented processes, and thought about possible future improvements, the following are the key-features taken into consideration for the exploration of different possible to paths to implement:

- Real-time data ingestion
- Store clean data
- Process data outside the API (but keeping API as a validation layer for writing data)
- Go beyond HTTP protocol to allow different communication protocols
- Good scalability
- Fault and delay tolerance

3.1.1 Technologies comparison

In order to propose reliable solutions, different technologies and programming models are compared. After the comparison the most promising one, with some suggestions from, the one that specific fits requirements is selected to be implemented. The first step has been searching in the vast offer of built-in products in Google Cloud Platform if there were some tools that could have been useful to implement ETL processes. After a first scan, two products were identify to make a comparison between them and their corresponding programming models:

- Google Dataflow and Google Dataproc
- Apache Beam and Apache Spark

Both Dataflow and Dataproc can be used to implement ETL solutions but they several differences. Let briefly make an overview of both tools, highlighting the main features.

Dataproc

As written in the Google official documentation [12], Dataproc "Dataproc is a fully managed and highly scalable service for running Apache Spark, Apache Flink, Presto, and 30+ open source tools and framework". It is a managed service that supports big data processing including ETL and machine learning.
With Dataproc on premise clusters can be moved to the cloud to maximize efficiency and enable scale. Dataproc automation helps users create clusters quickly, manage them easily, and save money by turning clusters off when are not needed. With less time and money spent on administration, users can focus on jobs and data. Once clusters are provided, for example Hadoop or Spark clusters, jobs can be submitted to Hadoop-ecosystem tools, like Apache Pig, Hive, and Spark for example. Clusters can be scaled up ow down even during the jobs execution, and prices are calculated in relation of used resources. Resuming, making use of four categories which are used for comparison with Dataflow in the next paragraphs, its key features can be identified as the followings:

- Velocity: on-premises clusters creation can be very time and resources consuming. Dataproc clusters are very quick to start, scale and shutdown.
- Integration: Dataproc has built-in integration with other Google Cloud Platform services, such as BigQuery, Cloud Storage, Cloud Bigtable, Cloud Logging, and Cloud Monitoring, facilitating jobs that make use of different GCP products
- Management: users can easily interact with clusters and Spark and Hadoop jobs through Dataproc REST API, Google Cloud Console or Cloud SDK without the assistance of a special cluster administrator o software. When a cluster turns to be problematic can simply be shut down and change with another one: data loss is avoided because of GCP built-in compatibility with products like Cloud Storage for example.
- Portability: even if there multiple tools where jobs can be submitted, the job code in bounded to the kind of runner and how it works, for example in the submitting of Spark job, that is bounded to the Spark runner and its logic

After a first analysis, Dataproc is an interesting tool that has features that can address Bamboo requirements like ETL but it is too bound to Spark and Hadoop systems where the starting point is the necessity to build a cluster, which probably goes out of scope of Bamboo needs.

Dataflow

As written in the Google official documentation [13], Dataflow is "Unified stream and batch data processing that's serverless, fast, and cost-effective." is a serverless data processing service that runs jobs written using the Apache Beam libraries. When a job runned on Cloud Dataflow, it automatically spins up a cluster of virtual machines, distributes the tasks in the job to the VMs, and dynamically scales the cluster based on how the job is performing: users do not need to address common aspects of running jobs on a cluster for example balancing work, or scaling the number of workers for a job; by default, this is automatically managed, and applies to both batch and streaming.
Dataflow also offers the ability to create jobs based on "templates," which can help simplify common tasks where the differences are parameter values. Dataflow core strength is linked to the usage of Apache Beam programming model which is intended to completely separate jobs native code, which is Beam with Python, Java or Go, from the runner: Beam jobs are portable across different runners like Dataflow, Flink or Spark.
Making use of the same categories used to highlight Dataproc characteristics, the following are identified as Dataflow key features:

- Velocity: Support environment is automatically and instantly built behind the scenes when a job is submitted to the Dataflow runner.
- Integration: Dataflow has built-in integration with other Google Cloud Platform services, such as BigQuery, Cloud Storage, Cloud Bigtable, Cloud Logging, and Cloud Monitoring, facilitating jobs that make use of different GCP products: multiple I/O connectors related to Google Cloud and other external products are available and many others are in development.
- Management: users do not need to spend time to environment management: it is guaranteed the automated provisioning and management of processing resources and the horizontal autoscaling of worker resources to maximize resource utilization.
- Portability: as said before Beam provides a clear separation between processing logic and the underlying execution engine. This helps with portability across different execution engines that support the Beam runtime: the same pipeline code can run seamlessly on either Dataflow, Spark or Flink.

After this first analysis, Dataflow is supposed to be a very interesting tool, especially looking at its underlying programming model, Beam. The first sensation is that Dataflow can be preferred instead of Dataproc to implement pipelines but let's make a more detailed comparison.

Dataproc-Dataflow comparison

Taken into consideration the key-features of both products, the final comparison can be made analyzing four factors: ETL logic, environment management, integration and compatibility. Taking into consideration data processing mechanics logic, we can say that both Dataproc and Dataflow can address the same kinds of data processing tasks: even if, of course, there are some differences between Beam and Spark, for example, like the fact that Baam is a more unified programming model in relation to batch/streaming, they can carry on basically the same ETL processes. Equally, thinking about integration, both have a great built-in compatibility with other Google Cloud Platform products or not Google ones, like fore example with InfluxDB through the Beam Influx I/O connector for example.
Taking into consideration instead environment management and portability we can notice grate differences. From one hand there a code/runner strict correlation in Dataproc, and on the other hand a complete separation in Dataflow: this makes the second one more flexible and not strictly bound to specific type of runner enabling portability and mobility to basically computing engine that supports Beam compatibility. In relation to environment management we can say that the first thing to say is understand if there is the necessity to use and maintain Hadoop/Spark clusters or to migrate on-premises Hadoop/Spark clusters to the cloud. Even if Dataproc provides many resources and automation the the setting up of clusters, it in needed a certain Hadoop-ecosystem familiarity and a certain DevOps approach, while with Dataflow users can completely forget about building, managing and maintaining the under-laying infrastructure and just focus on jobs. After this comparison, Dataflow is chosen to be the tool to study in deep in order suggest and implement alternative ETL approaches. After this choice, searching in the GCP documentation, a flowchart helping suggesting when it is the case to opt for a tool instead of the other, has been found mirroring this decision process:

3.1.2 Dataflow/Beam focus

After the decision of making use of the Dataflow/Beam paradigm to implement ETL pipelines, this section is intended to be a general overview of this GCP product, in particular to give more details about Apache Beam programming model. Previously it has been said that Dataflow is distributed processing backend: when a pipeline is runned with the Cloud Dataflow service, the runner uploads the executable code

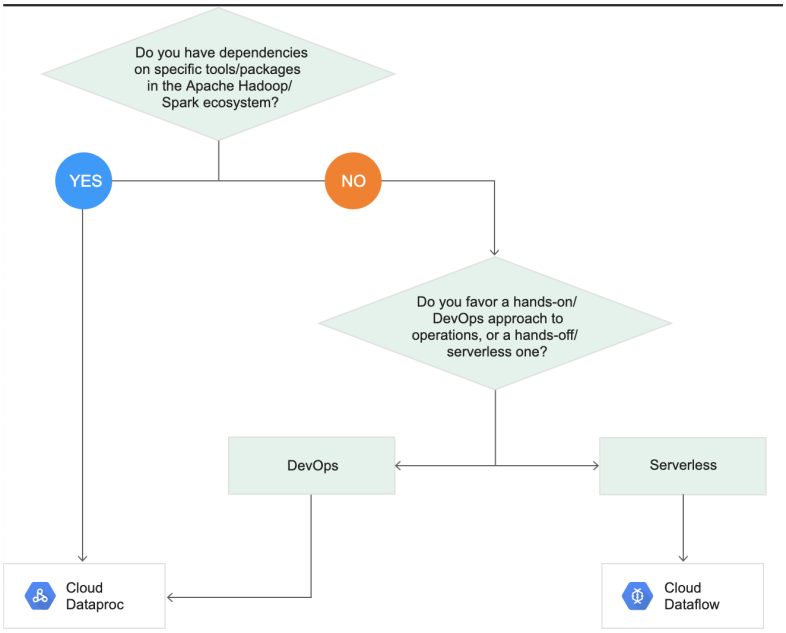

Figure 3.2: Dataproc/Dataflow choice flow-chart

and dependencies to a Google Cloud Storage bucket and creates a Cloud Dataflow job, which executes the pipeline on managed resources in Google Cloud Platform. This service handles very well large scale, continuous jobs providing provide a fully managed service, autoscaling of the number of workers throughout the lifetime of the job and dynamic work rebalancing. Once having understood Dataflow features, expecially the fact that is completely managed serverless service, we can just focus on pipelines developing with the Beam model.
Apache Beam is an open source unified programming model for both batch and streaming data-parallel processing pipelines [14]. A pipeline logic can be programmed making use of three different SDK:

- Apache Beam Java SDK
- Apache Beam Python SDK
- Apache Beam Go SDK

Jobs can be coded alsp through Scio, that is a Scala API for Apache Beam and Google Cloud Dataflow inspired by Apache Spark and Scalding. the coded job can be submitted to one of the compatibles distributed processing back-ends, which are:

- Direct Runner

- Apache Flink Ruuner
- Apache Nemo Runner
- Apache Spark Runner
- Google Cloud Dataflow Runne
- Hazelcast Jet Runner
- Twister2 Runner

In order to illustrate Beam programming, main concepts and abstraction definitions must be made.
The core concept is Pipeline: it contains all the data processing logic, from I/O to data transforms. It can be defined as a directed acyclic graph of all data processing tasks coded in the job. Foundations concepts of Pipelines are PCollection and PTransform.
To correctly define what is a PCollection, the official words can be useful: "A PCollection is an unordered bag of elements. Each PCollection is a potentially distributed, homogeneous data set or data stream, and is owned by the specific Pipeline object for which it is created. Multiple pipelines cannot share a PCollection. Beam pipelines process PCollections, and the runner is responsible for storing these elements". To make a parallelism, it is similar to the Resilient Distributed Database in the Spark model. PCollection can be bounded or unbounded. A bounded Pcollection is dataset with fixed o never growing in time size that can be processed in batch pipelines. An unbounded PCollection instead is a dataset that grows over time, and the elements are processed as they arrive. Unbounded data must be processed by streaming pipelines.
As it is defined in the documentation [14], "a Ptransform represents a data processing operation, or a step, in your pipeline. A transform is applied to zero or more PCollection objects, and produces zero or more PCollection objects." The processing transformation logic is provided in the job in the form of user-defined function and is applied to every element of one or more input PCollection. There are different kinds of transformations:

- Source transformations: they are used to read from data sources, like TextIO.Read and Create for example. A source transform conceptually has no input.
- Processing and conversion transformation: they are used to make transformations from input PCollections. Examples are ParDo, GroupByKey, CoGroupByKey, Combine, and Count. ParDo in a general user customizable logic that is processed in parallel

- Outputting transforms: they are used to write to data sinks, like TextIO.Write.
- User-defined transforms: they are used to code application-specific composite transforms.

There a lot of concepts more to be mentioned here in this section, like Windowing, which is the features of assign PCollection elements of time logic windows or Triggering, which is the feature of emitting aggregated results before the end of Window, but this section is intended to give a general overview focus on Apache Beam programming model.

3.1.3 Data ingestion

This section focuses on the explored solutions related to data ingestion. After having identified the key-features and the characteristics of the new ETL pipeline and usage of Dataflow, basically two different approaches have been taken into consideration.

Approach 1: original implementation + datalake + streaming ingestion

The first proposed solution is closer to the original one in order to try to improve the original implementation without too many modifications. The following figure resume the dataflow. Basically the original steps are maintained:

- Asset device datasource makes a POST Bamboo API call when there is a new measurement: API is used as validation layer to write data
- the API writes data without processing to a data sink, that in the original implentation is InfluxDB, the database used for production.

At this point begins the first proposed solution. The idea is not to write directly raw data to the time series Database without actual processing, but to write data to a datalake that for example could be InfluxDB itself or a Cloud Storage bucket. Then it is implemented a streaming pipeline coded in Beam and runned in Dataflow to make some processing and ingest data basically in real time in the database using Bamboo API as a last validation layer. This approach is not so different from the original one except from the writing of data in temporal location where a Dataflow pipeline can make some processing and load data to InfluxDB. The major advantages can be found thinking about the minimal changes to be made to implement this approach: changing the data sink in the API endopoint, that is called from the datasource when a new measurement is made, to load data in the datalake, means a null integration cost on the client side. Then it is implemented the streaming Dataflow pipeline that reads form the datalake, performs some

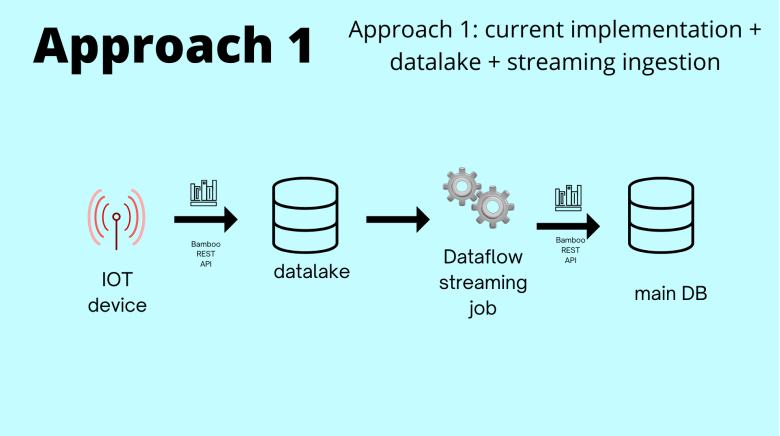

Figure 3.3: Approach 1: original implementation + datalake + streaming ingestion

processing and and loads data almost in real time to InfluxDB with Bamboo API. So basically the difference is the presence a of datalake where raw data are processed through Dataflow and loaded into the DB. This approach does not add anything new in terms of scalability and mostly delay and fault tolerance as the first step of the dataflow, from the asset device, is the same.

Approach 2: routing data sources to an asynchronous messaging service + ingetion through the API

This approach differs more from the original one, introducing the usage of an asynchronous messaging service as a middle-ware between datasource and processing and loading steps identified in Google Cloud Pub/Sub, a GCP built-in product with great integration compatibility with Dataflow. The reason behind this idea is the amount of documentation and use cases that associate a messaging service middle-ware with streaming ingestion pipelines: the usage in synergy of Pub/Sub and Dataflow guarantees data completeness and exactly one processing because of its low and high watermarks accuracy and efficient deduplication, handling very well late data management, data loss and scalability in general.
The figure illustrates the ingestion approach proposed. Basically the usage of Pub/Sub and Dataflow is the following: the asset device sends a data measurement to a Pub/Sub topic, which is the main Pub/Sub object used of asynchronous messaging where a specific subscription waits for messages to arrive. Connected to this subscription, in Dataflow it is active a streaming job that listens from the Pub/Sub subscription and process data and uploads them to InfluxDB utilizing Bamboo API as data writing validation layer. The idea is to split the data writing into two different location: store the raw data measurement in InfluxDB in a

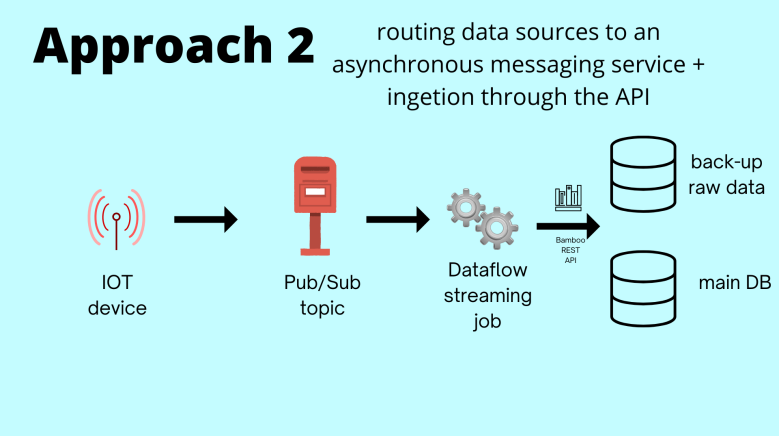

Figure 3.4: Approach 2: routing data sources to an asynchronous messaging service + ingestion through the API

section used for back-up data which has temporal memory policy and in parallel performing the data process and load the clean data in the main InfluxDB database. Resuming, this approach introduces something new but also makes use of original implementation:

- Pub/Sub: the introduction of this tool brings great advantages in terms of reliability as described before
- The way of writing data remains the same as the original approach making use of Bamboo API, but this time data is sent to to different locations.

Thinking into consideration the integration cost for customers-side, this approach is less soft that the first as the asset device has to publish to the Pub/Sub topic instead to make Bamboo API calls. A solution to that is to give the possibility to customers to publish directly to the topic if they are able to make this implementation, otherwise a new API endpoint is implemented to publish to the topic, without connecting directly the asset device and Pub/Sub.
Comparing these two approach, the second one seems the most interesting for the advantages brought by the introduction of Pub/Sub and the implementation cost that is not traumatic both for customers side and for Bamboo side. This kind one pipeline is selected to be implemented: more details are given in the next chapter.

3.1.4 Data cleaning and processing

after having understood where insert data ingestion in the pipeline, this section explain the proposed solutions related to data cleaning and processing. The most important question to answer, in relation to the processing steps described in the

previous chapters, it to decide what kind of processing is done in Dataflow, in the API and in which part of the pipeline. As baseline, flexibility and bidding module need have similarities and some differences in terms of types of datasources and kinds of preprocessing tasks needed it has been decided to leave preprocessing steps that are made in batch like outlier detection, up/down sampling and interpolation inside the API backend when some data are requested to be fed into some particular algorithms. The idea is to maintain the already implemented methods discussed in the previous chapter and to add some controls before ingesting the data trough the API in order to have a standard of data cleaning for every asset device typology before data loading into the database. In particular for every device different preprocessing policies are defined:

- Out-of bounds values: threshold values are queried and use to detect out-of-bounds measurements
- Fixed sign policy:sign policy is implemented: power consumption has positive real values, energy generation has negative real values
- Invalid measurements: Nan or None measurements are not stored in the clean database

With this approach, some cleaning rules are applied before ingesting the data and backend is not loaded with more computation, as this steps are performed in Dataflow: although these steps might seem very basic, and in fact they are, thinking about a grate production environment where there are a lot of sites with a lot of devices sending measurements with high frequency, the solution proposed so far in relation of data ingestion and cleaning is a good starting point to have a solid, reliable and scalable architecture. With this approach clean data are stored in the database, the different modules have a common clean database from which loading data and conducting particular additional preprocessing steps.
So the proposed solution that will be implemented, whose details are discussed in the next chapter is the following, to resume: before the API call to load data into the DB, common cleaning tasks are coded in Beam and performed in Dataflow Runner while special module specific tasks, that can also be in common between two different modules, are performed in the backend when data are requested for module usage. Resuming these are general preprocessing tasks computed in batch and streaming:

- common tasks performed in the streaming Beam pipeline runned in Dataflow:
  - Out-of bounds values
  - Fixed sign policy
  - Invalid measurement

- module specific tasks performed in batch in the backend:
  - Outlier detection
  - Up/Down sampling
  - Interpolation

3.2 Aggregation data pipelines

Analyzing Bamboo Energy processes the absence of data aggregation pipelines, especially related to baseline and flexibility forecast, has been detected and so, as a good starting point, aggregation pipelines solutions to compute KPIs related to baseline forecasts are explored.

3.2.1 KPIs

First, in order to inspect the pureness of baseline forecast module the correct KPIs must be identified and then the considerations about the architecture are made. A asset device baseline forecast is a regression task, among all the possible ones, the following have been selected to be the kpis computed for each device of the corresponding site taken into consideration, where y are predictions and x measurements:

- coefficient of variation of the Round-Mean-Square- (CV-RMSD) Deviation : $\sum_{i=1}^{D} \frac{(y_i - x_i)^2}{\hat{x}}$
- Mean absolute error: $\sum_{i=1}^{D} |y_i - x_i|$
- bias: $\sum_{i=1}^{D} y_i - x_i$
- max error among the bias

3.2.2 Technologies comparison

This section focuses on the technologies taken into account to implement KPIs aggregation pipelines, which can be divided into two categories:

- Architecture where batch aggregation pipeline takes place (Beam vs Cloud Function vs Cloud run)
- Database in which KPIs are stored (Big Query vs Bigtable)

Architecture

In order to select the correct tool to build the kpis computation architecture different solutions have been taken into consideration:

- Cloud Function
- Cloud Run
- Dataflow

Taking into consideration Cloud Run as host of kpis module the question is to decide if add backend endpoint to compute kpis for the devices of specific site as backend in hosted on Cloud Run, or to create a new instance. As it has decided to implement kpis computation outside the backend, for purpose to make a separable and more quick to modify module, the idea of mountain a new instance has been taken into consideration. Thinking about the not so great complexity in terms environment settings and dependencies and the not so heavy load in terms of computational complexity of the kpis module, this is idea is discarded as the strength in terms of scalability and resources provision of Cloud Run are too much respect to actual needs. The same reasoning can be applied to Dataflow: the parallel processing technologies are not need in this case, especially if the computation has time frequency not high. A good solid solution to build kpis module is represented by Cloud Function whose key features as described in the previous chapter.

Storage Database

InfluxDB is intended to be the main time series database. Even if in a certain sense kpis can be considerated time series, it has been decided to find another storage alternative, in ones offered in Google Cloud Platform. In particular, as the kpis implementation is finalized to kpis analysis, a good solution could be represented by a database with a low price writing cost and possibly with the possibility of easily and at low price making queries and analysis. In order to find a suitable and reliable solutions the following GCP built -in products are compared:

- Cloud Storage: its key features have been described in the previous chapter. Its a general Storage products without the possibility of making queries.
- Big Table: it is a NO-SQL columnar database running on HDSF, suitable for high throughput applications. It performs well for application with huge read/write operations or general very frequent data ingestion because it has low latency. It is highly scalable and has a great compatibility with BigTable. Three key factors must be taken into consideration in relation to pricing: Big Table instance and the total number of nodes, the amount of storage anf of network bandit [15] . To summarize, it is more used as a OLTP database.

- Big Query: it is a immutable database where new records can only be appended but its strength is that it is more like a Query Engine as it is optimized to run SQL queries on very large Dataset so it is intened to be a more OLAP database. It is very scalable and has great compatibility with other GCP built-in products like Dataflow. There are town componentes to be considerated in relation to pricing: storage and analytics, which is the more expensive [16].

Comparing these three different products, as Cloud Storage does not have analytics support, Big Table resources are too much in comparison to the ones needed by the kpis module, Big Query is selected to the the storage database, taking advantage of its query optimization, low storage cost and also low query cost as the Kpis dataset will not be huge as kpis data are results of aggregation pipelines, condensating a lot of data in very few numbers.

3.3 Orchestration

In the previous chapter it has been described what is and how and for which purpose Cloud Scheduler it used. Resuiming, it basically a managed Cron service that allows to schedule in time processes in a very easy and intuitive way: baseline and flexibility forecast, market and weather data download are examples of processes whose execution scheduling can be automated in time. The great ease of use of Cloud Scheduler hides some limitations that can be identified as the followings:

- limited offer of triggering possibilities: public available HTTP/S endpoints or built-in GCP resources.
- lack of processing dependencies
- poor processes retry policies
- scheduling options limited to Cron based jobs

As Bamboo Platform is the day by day growing in complexity, the Orchestration tool is intended to very very reliable and to give great flexibility and freedom in processes automation. Once having clearly identified actual implementation limitations, this section focuses on finding a better substitute of Google Scheduler to bring to Bamboo Platform to the Orchestration level needed: as the platforms grows in complexity, new modules are developed, more an more baseline, flexibility and bidding optimization algorithms are launched more and more frequently, the Orchestration tool must be the last layer that makes everything works in synergy, the last actor that is able to bring infrastructure to higher quality standard levels

3.3.1 Technologies comparison

Having discarded the option of making use of Cloud Scheduler, some reliable alternative solutions are compared. The first option has been to consider Apache Airflow, an open source project of Orchestration for its increasing popularity: its great offer of customizable scheduling and the multiple ways of usage made it the industry standard cite [17]. As seems very promising, and taking suggestions from [18], Airflow is chosen to be the Orchestration tool to be used to schedule processes, from data processing to modules launching.
Collecting information and studying Airflow complex architecture and programming model, whose details are given in next section, the thing to consider at this point is to decide to opt for a standard, manually built and maintained Airflow version or for managed one. This point is tough: where a tool as a complex infrastructure to build, maintain and monitor often there are companies that build on top a service to make the usage of the original tool easier and more user-friendly especially to novice users. It is the case for example of Google, who built its managed Airflow product, called Google Cloud Composer or the case of Astronomer, a company that basically did the same. Basically a trade-off must be made between the issues related to build and maintain an Airflow instance, the high cost for using managed services and the actual needs and requirements of the production environment: as Bamboo environment is small but it is growing day by day, it has been chosen to try both the approaches, as described in the following chapter.

3.3.2 Orchestration requirements definitions

The section focuses on the features that Bamboo products need to find in Airflow, the Orchestration tool selected. ETL Pipelines, forecasts algorithms, bidding optimization, data download are some examples of which kind of processes need a reliable and solid scheduling policies. The followings are identified as the features that need to be add to Bamboo process and that could be find in Airflow:

- temporary scheduling: go beyond the Cron based limitation, a process execution should be triggered on only by time policy but also by other process in relation of its outcome
- intra-jobs dependencies: two tasks, in the same processes, having logic dependencies have to be scheduled modelling this dependencies and just not using time based job
- retry policies: go beyond poor standard task retries policies, adding some action triggering in relation of the outcome of the task.
- log monitoring: have a solid monitoring of tasks logs in order to quick know the particular reasons behind failures

Having selected Airflow as Orchestration, for its popularity and great functionality appearance, and having identified the key requirements of the process scheduling that go beyond Scheduler limitations, Airflow architecture and programming model is deeply studied.

3.4 Airflow focus

Airflow is an open source platform to programmatically author, schedule and monitor workflows. These workflows are defined as Directed Acyclic Graph (DAG) and contain individual pieces of work called Tasks, arranged with dependencies. In a Dag can be specified the characteristics that make Airflow such a powerful tool: inter-tasks dependencies, the order in which to execute them, the number of retries if some tasks fail and many more. In the Tasks it is defined what to do: fetching data, making a connection, calling an endpoint, triggering a service and so on.
Airflow is an Orchestrator allowing you to execute tasks and more generally data pipelines at the right time, in the right order, in the right way ensuring great reliability and monitoring. These are some of the key benefits:

- dynamicity: basically everything that can be done in python can be done in airflow data pipelines
- scalability: automatic distributed task processing depending on the workload and the set available resources brings great efficiency
- user interface: Airflow brings its user interface making DAGs visualization, triggering, testing, monitoring very user friendly and clear
- extensibility: plug-ins can be added to provide custom ad hoc solutions

Architecture

Airflow has its particular architecture, which is different among the single node and the multi-nodes architecture versions, but these are the key components of the general Airflow architecture.

- Web Server: it's a flask server run using gunicorn and it's responsible for serving the UI Dashboard with http. It's useful to get an overview of the overall health of different Dags and also to help visualize different components and states of each DAG. The Web Server also provides the ability to trigger and test DAGs and to manage users, roles, and different configurations for the Airflow setup like connections and variables

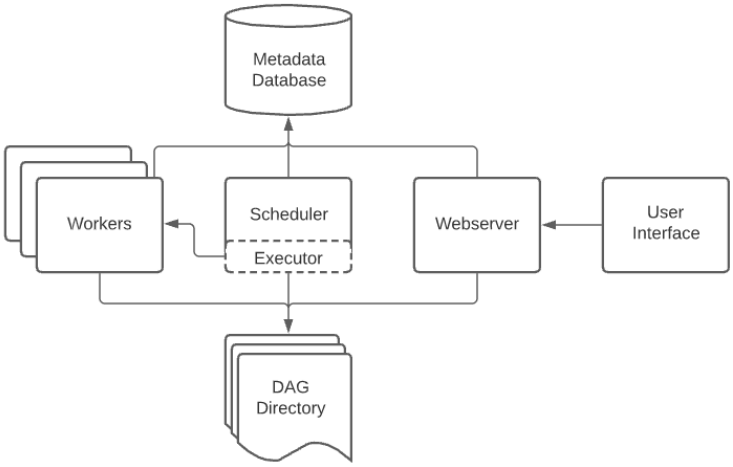

Figure 3.5: Airflow architecture diagram [18]

- Scheduler: it's the most important part of Airflow, which orchestrates DAGs and their tasks, taking care of their interdependencies. It's responsible for turning the Python files contained in the DAGs folder into DAG objects that contain tasks to be scheduled. It's a daemon, built with python-daemon library, in charge of scheduling workflows, tasks, and pipelines. A workflow script consists of several components which the scheduler interprets and stores in the metastore.

- Metastore: a database where Web Server, Scheduler and Executor store metadata states. The metadata database stores configurations, such as variables and connections. It also stores user information, roles, and policies. The Scheduler parses all the DAGs and stores relevant metadata such as schedule intervals, statistics from each run, and their tasks. Everything that happens in Airflow is registered in the metastore. All database operations are performed with SQLAlchemy, a Python ORM framework.

- Executor: it's the actual entity handling running tasks. In the default installation this runs everything inside the scheduler but most production-suitable executors actually push tasks execution out to workers the executor has. There can be different types of executors, divided in Local and Remote. Airflow can have only one executor configured at a time.

- Worker: process/subprocess executing tasks that listen to, and process, queues containing workflow tasks.

Airflow programming model concepts

Airflow is written in Python, and workflows are created through Python scripts. Beside Python logic, the following paragraphs are intended to be an overview of the core concepts of Airflow programming model

DAG A Dag is the core entity of Airflow and is made of tasks organized together with dependencies and relationships to say how they should run. A DAG is stored in the dags folder of the Airflow environment and the Scheduler parses the DAGs code, which is a Python file, in order to make them visible and usable in the Web Server.

Task A task is the basic unit of execution in Airflow. Tasks are arranged into DAGs with their interdependencies set in order to express the order they should run in. There are three basic kinds of Tasks:

- Operators: predefined task templates that can be chained together to build most core parts of DAGs
- Sensors: a special subclass of Operators which are entirely about waiting for an external event to happen
- TaskFlow-decorated @task: a custom Python function packaged up as a Task

They are all subclasses of Airflow's BaseOperator and Tasks and Operator concepts are interchangeable but it's useful to think of Operators and Sensors as templates, which become Tasks when they are called and built in a Dag. The strength of Tasks is declaring their inter-dependencies, once Tasks are declared.

Operator An Operator is conceptually a template for a Task that are defined inside a Dag. Some examples of Operators are:

- BashOperator: executes a bash command
- PythonOperator. calls an arbitrary Python function
- EmailOperator: sends an email
- SimpleHttpOperator: calls an HTTP endpoint

Users can also override BaseOperators class in order to write customized Operator in relation to special needs.

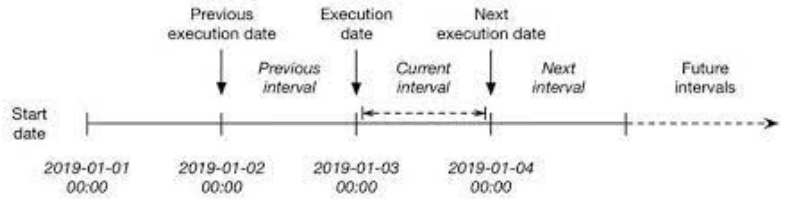

Figure 3.6: Airflow schedule interval[17]

Schedule Interval Users can define when Airflow runs a DAG with three parameters:

- start_date: date from which Airflow starts to run the dag
- schedule_interval: Trigger DAG frequency
- end_date (optional): last date of DAG triggering

In this interval-based representation of time, a DAG is executed for a given interval as soon as the time-slot of that interval has passed. In Airflow, the execution date of a DAG is defined as the start time of the corresponding schedule interval rather than the time at which the DAG is executed (which is typically the end of the interval). As such, the value of execution_date points to the start of the current interval, while the previous_execution_date and next_execution_date parameters point to the start of the previous and next schedule intervals, respectively. The current interval can be derived from a combination of the execution_date and the next_execution_date, which signifies the start of the next interval and thus the end of the current one.

XCom Data transfer between different tasks is not direct and intuitive as it might seem. Different tasks do not directly communicate because of the Airflow architecture, as by definition tasks are entirely isolated and may be running on entirely different workers. To address this issue, XCom, which stands for "cross-communications", makes possible data transfers between different tasks of the same DAG run. Xcoms are sort of variable, but they are only related in the specific DAG run and are not globally available to other DAGS as Airflow Variable are. XComs are suited for small amount of data that can be serializable. Through the usage of *xcom_push* and *xcom_pull* on the Task instasnces, XComs are explicitly send and returned from their storage. There are some Operators that by default push their outcome to XCom, like the BashOperator.
A broader focus on architecture and programming model is given to Airflow as it has a central role in the developing of this thesis: it's the actor that orchestrates all

the processes examinated so far and makes possible also the Alerting ones described in the following sections.

3.5 Alerting

In a data-drive company it is very important being aware in very quick times when some process or job is failing in order to behave accordingly and be prepared. Nevertheless, when cloud architecture are modular and complex it is always a good practice to have a well defined Alerting policy, in order to monitor processes in tidy and precise way, centralizing in the same channel the notifications of the same type. This section focuses on Alerting, on the features requested and the technologies compared to plan an alerting policy which is composed of internal and external alerts policy. Before working on these topics, it is crucial to correctly identify the kind of events that are must be detected in order to trigger internal and external alert policies

3.5.1 Technologies comparison

The purpose to implement internal or external Alert policies is to notify to Bamboo team or to customers some kind of event through some medium. Basically, both of internal and external purposes, the comparison is made between the following technologies:

- Google Error Reporting
- Slack Notifications
- Email service provider

Google Error Reporting

One of the great benefits of having a cloud-based architecture, especially in GCP, is the great integration and compatibility between different products. Google Error Reporting is a key products on which a lot of other GCP built-in product can send logs of operations [19]. During the building of application running in Google Cloud App Engine, Cloud Function, Cloud Engine it is possible to enable the automatic send of processes logs to Error Reporting, otherwise utilizing Error Reporting API. This feature is already implemented in Bamboo platform modules: whenever a service is called, the response is logged directly into Error Reporting, making it a good location in which store error logs and ding monitoring. Despite this fact, which makes Error Reporting a good place to store error logs it cat not alone be used for alerting purposes as there must be a person continuously monitoring Error

Reporting user interface in order to detect certain events and to accordingly set notifications, both in case of internal or external ones.

Slack Notification

Slack is the tool used for Bamboo internal communication. Apart from direct communication between developers and other employees, there is the possibility to create channels in which more users can participate and where Slack messages can be sent making use of Slack api [19]. This is a interesting tool to route internal notifications for example, in the form of Slack messages.

Email provider

Another tool that can be used when a certain event is detected to send the corresponding notification is a generic Email provider. An account can be provided and services can be created to send emails of the same kind from the same account. users can subscribe to these services and receive emails notifying the occurrence of a particular event. This is an interesting medium in particular for external alerting purposes.

Taken into consideration both the internal and external alerting policies it is decided not to centralize all notifications in the same medium or location but to use an hybrid approach with all the three combined for internal notifications and just emails for external alerting: this is due to the nature of events that are presented in the next section.

3.5.2 Alert policies

This section talks about the features requested for building an alert policy, both for internal and external purposes, the events that are detected, how and with which technologies are detected. It is decided, as design choice, that every external notification has a internal correspondent.
The first thing taken into consideration has been to decide to maintain the current implementation of making use of Error Reporting when services, hosted in GCP products, are requested. This serves as a central repository where services responses can be inspected.
Secondly, as Slack is used for internal communication and constantly during working hours it is decided to open a dedicated notification channel where are routed logs of failed services and other events.
Regarding customers, when certain kinds of events are detected a mail is sent to customers. This is in developing phase and the infrastructure building goes out of scope of this thesis work. A corresponding mail in sent internally and a correspondent Slack notification is sent to the dedicated channel. Now it is the

time to describe the particular kind of events that are detected and if they are intended for internal or both internal and external purposes:

- a device has stopped or re-stared sending data (internal and external notifications)
- a device has sent a measurements that is above or below a certain threshold, both for power or temperature measurements (internal and external notifications)
- market data download failed (internal notifications)
- weather data forecast failed (internal notifications)
- baseline and flexibility forecast training (internal notifications)
- baseline and flexibility forecast fail (internal notifications)
- kpis computation failed (internal notifications)

These kinds of events are selected taking into consideration the level of production environment, the state of development of some modules and some customers need and are intended to be a starting point in order to build solid and reliable events detection and alerting policies environment.
In relation of the technology used to detect the events it has been decided to maintain Error Reporting for modules calls failures, to implement detection of events related to devices measurements with Apache Beam and Dataflow as represent perfect use cases for streaming pipelines. In addiction, it has been decided to use a great Airflow feature, called callback function, to implement internal Slack notification in case of modules call failures as it is described in the next chapter.

Chapter 4

ETL, Aggregation Data pipelines, Orchestration and Alerting implementations

Chapter 2 served as overview of Bamboo processes and understanding of the already built implementations related to this different areas of Data Engineering in order to correctly identify strengths and weaknesses to lay solid foundations to next step. In the last chapter instead, multiple alternative solutions, related to every Data Engineering field inspected, both in terms of implementations logic and architecture design, have been proposed and some comparison have been made in order to select the most reliable and promising ones to be actually implemented.
In this chapter this Data Engineering journey, after understanding and exploration phases, goes to an end. Intended to be the conjunction point between chapter 2 and 3, the following sections focus on the practical implementations of the proposed and selected possible solutions to address some specific tasks related to the fields inspected.

4.1 ETL and data aggregation pipelines

It has been decided to include in this section both ETL and data aggregation pipelines: in particular, the new approach to address data cleaning and ingestion, making use of Pub Sub and Dataflow, and the baseline kpis module implementation are described.

4.1.1 Data pipelines definitions

ETL: Data cleaning and ingestion pipeline

In order to improve the currently implemented ETL processes, as an outcome of the previous chapters, the solution chosen to be implemented to address data cleaning and ingestion is a streaming pipeline built with Apache Beam (Python SDK) and executed in Google Dataflow. To conduct tests on this particular use case, some considerations and assumptions must be made in relation to the preliminary steps needed to put in in practice this pipeline before putting it into production.
In order not to make changes on the customers side to route measurements data to a Pub Sub topic, resulting in possible production issues, the processing of ingesting data sent from IoT sources has been simulated. The followings are the assumptions of this particular use case:

- Assets random measures generation: a Cloud Function has been built in order to simulate random measures of particular assets.
- PubSub topic: a specific topic for this use case has been created.

The Cloud function, coded in Python and making use of Numpy and Google Cloud PubSub libraries, takes as input the following information data about a specific asset:

- site name
- asset name
- threshold values

and generates a random asset measurement giving a certain probability associated to a particular event in order to simulate all possible real cases scenarios:

- correct value range measurement
- correct value range measurement but with sign inverted
- Nan measurement
- out-of-bounds measurement

Following general requirements explained in section 3.1 and making use of the Cloud Function to simulate asset measurements and a PubSub topic to ingest them, a Beam streaming pipeline has been implemented: its logic details are explained in the next section.

Data aggregation pipeline: baseline kpis computation

In the previous chapter different solutions of possible baseline kpis computation have been explored: the goal is to build a light-weight and manageable tool to periodically compute kpis and and kpis analysis. The following, as explained are the kpis to be implemented:

- coefficient of variation of the Round-Mean-Square- (CV-RMSD) Deviation : $\sum_{i=1}^{D} \frac{(y_i - x_i)^2}{\hat{x}}$
- Mean absolute error: $\sum_{i=1}^{D} |y_i - x_i|$ (MAE)
- bias: $\sum_{i=1}^{D} y_i - x_i$
- max error among the bias

In particular, among pretty well known and common metrics, CV-RMSD, being normalized, is an interesting ad useful metric as can be used as unified criterion to inspect baseline forecast pureness among different devices, which have very different orders of magnitude.
As in this specific use case we do not need the strength of cloud parallel computing and great scalability offered by the Dataflow Engine, it has been decided to mount a light-weight Cloud function to have a very handy tool to compute batch kpis pipeline.

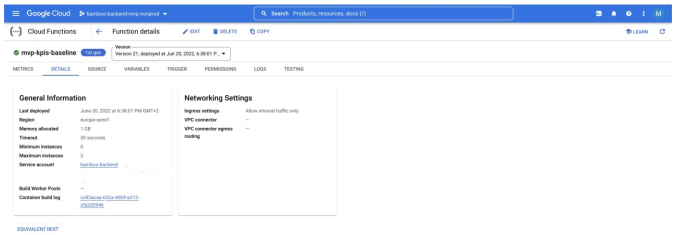

Figure 4.1: Baseline kpis cloud function general information

To easily inspect and analyze devices kpis results it has been decided to use BigQuery as database as it fits our requirements and has a built-in SQl Engine to rapidly make queries in an optimized way. A dedicated database has been created hosting the baseline kpis table, with a defined schema.

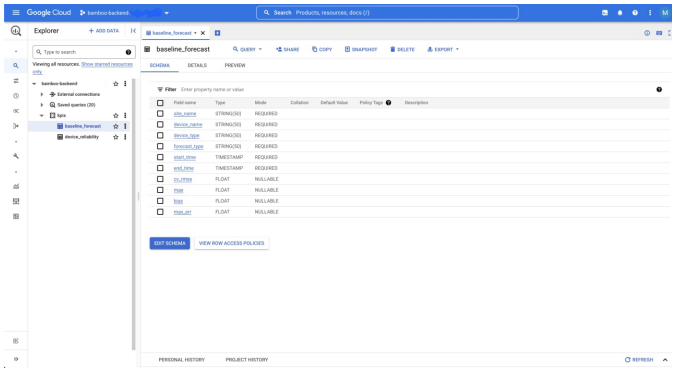

Figure 4.2: Baseline kpis table schema

4.1.2 Data pipelines implementations

This section is intended to be an overview of the streaming pipeline used to carry one ETL processes, made with Apache Beam and executed in Dataflow, and the batch processing used to compute kpis, built with a Cloud Function: for every pipeline, architectural and programming logic details are explained.

ETL: Data cleaning and ingestion pipeline

To execute this pipeline some steps must be made in order to build this use-case. As said in the previous chapter three components have been built to lay the foundation of the experiment.

- PubSub topic: a specific topic has been created in order to route simulated measurements. The topic has been created with a Pull subscription mode.
- Dummy fake measurement generation: a Cloud Function has been built. Its entry-point receives information about the site, the specific device to simulate, like device name and threshold values for example, and with the usage of Numpy and random packages generates a random measurement, among the four possible scenarios describe in the previous section. After the measure is generated, it is ingested to the PubSub topic through the usage of the pubsub client of the Google Cloud package.
- Simulated measurements periodic generation: in order to automatically and periodically generate a random measure an Airflow DAG has been created to trigger the Cloud Function with a custom schedule interval.

Once built these three components, we have these simulated assets that periodically send measurements to our PubSub topic. Let's now inspect the streaming pipeline that cleans and stores data, listening from the topic. The pipeline basically listens

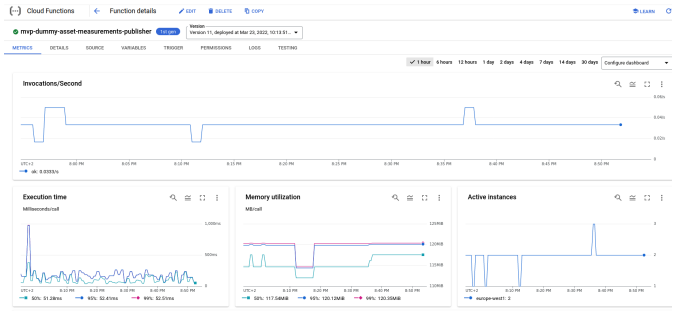

Figure 4.3: Cloud functions generating random measurements metrics

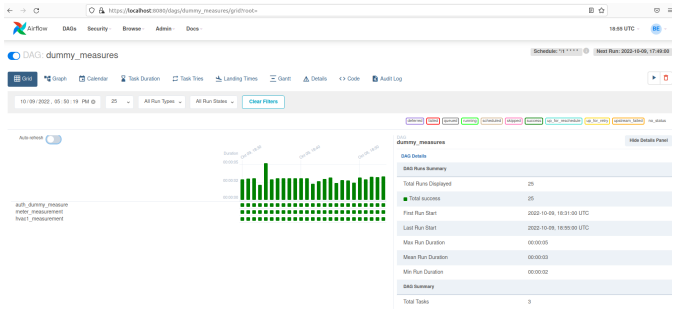

Figure 4.4: Airflow triggering random measurements Cloud Function

to the flow of the topic and when a message is published in the topic by the cloud function processes it and stores it in InfluxDB making use of the BambooAPI Client. The streaming pipeline makes use of two different input, defined as follows:

- main input: measurements PubSub messages read from the topic through the Beam IO PubSub connector
- side input: specific devices thresholds queried from the backend through Bamboo API client in order to cache these values to detect out-of-bounds values.

In order to give more details about the pipeline, in relation of both side and main inputs every step performed is described. Starting from side input, the first step is to create a custom PCollection from in-memory device categories data making use of Create transform. The available device categories are the following, passed as input parameters to the Create transform:

- meters
- shiftable loads
- batteries

- pv systems
- ev chargers
- thermal loads
- thermal comfort devices

At this point, we have a first PCollection to be processed. The second step is to create a custom DoFn executed inside the ParDo transform in order to be processed in parallel. The logic of this custom DoFn is the following: for each element of the input PCollection, and so for each available device category, through Bamboo API client a call to the backend is made in order get specific devices thresholds, outputting the specific device and its thresholds. In other words, it is a sort of Flat Map because for each device category one or more devices can be outputted, with the corresponding thresholds. The output results in a PCollection where each element is a tuple with the specific device name and its threshold.
Having queried and cached devices name and thresholds, the focus is to correct read and process PubSub messages. This pipeline consists in four steps:

- Read from Pub/Sub topic: through the usage of Beam io library, more precisely of PTransorm ReadFromPubSUB which allows to read utf-8 string payloads from Cloud Pub/Sub, the pipeline is able ingest PubSub messages from a PubSub source, which in this case is a topic as the specific topic to carry on the simulation is specified in the input parameter of the PCollection.
- Covert bytes to dictionary: making use of the classical Map PTransform, each incoming record is decoded to a dictionary from utf-8 encoding, through the usage of a simple lambda function.
- Map for Bamboo API requirements: each decoded element is mapped again in order to be processed and to be ingested respecting API requirements. From each decoded message some information are extracted such as site name, device name and measurement and a tuple is emitted as output.
- Out-of-bounds detection and Loading: a custom DoFn is created to be applied to each incoming measure and takes as input the side input with devices names and thresholds. For each measurement, the specif device thresholds are found in the side input in order to decide what to do with the incoming message:
  - out-of-bounds value: it is not stored and an internal log is emitted
  - Nan value: it is not stored and an internal log is emitted
  - in-bounds value: it is stored through the Bamboo API client and an internal log is emitted

– in-bounds value with sign inverted: the sign is changed, stored through the Bamboo API client and an internal log is emitted

At this points, having explained the general pipeline loigc, some others key features are inspected.This streaming pipeline makes use of default Global Window policy: when a new message coming from the topic is read the PTransform are immediately applied and the record is processed. So, it created a PCollection of one element every time a new message is read and it is processed. This design choice is because we are interested in loading data to database as they are are ingested in the topic and so Fixed Windows policies have been discarded. During the construction of the pipeline object the Streaming flag is set in order to use specific streaming resources, like the PubSub source for example, and also the "save_main_session" in order to use globally imported modules in the different DoFn as they might be executed in different workers in the Dataflow Engine.

Two main improvements are identified for this pipeline:

- specify an existing subscription and not a topic in the input parameters of the ReadFromPubSub PTransform: in this way it is guaranteed no data loss when the pipeline is stopped as the subscription remains active ingesting messages

- cyclically update the side input thresholds query from backend with the usage of Periodic Impulse PTransform in order to correctly reflect in the pipeline when new devices are added to a site microgrid or when thresholds change.

In the next figure some results and details of execution in the Dataflow Runner and exposed.

Figure 4.5: Cleaning and loading pipeline throughput

Data aggregation pipeline: baseline kpis computation

Having illustrated baseline kpis definitions in the previous section, this subsection gives more details on the actual implementation. As explained before, the chosen architecture to build this module is a Cloud Function. On the other hand the storage solution is BigQuery. A corresponding dataset, entitled "kpis" is created and a child table to store baseline kpis is created, with a specified schema. The core libraries imported are Scikit-Learn, Pandas and Bamboo APi client. The main entrypoint of the Cloud Function receives as input parameters the site id, the device name, start and end time of the corresponding time period of interest, and the type of baseline forecast, Day-ahead or Intra-day. The main steps of the computation of baseline kpis are the following:

- Data loading: through the usage of Bamboo API client, measurements and forecasts of the corresponding device are queried from InfluxDB, in relation to specified start and end times
- Preprocessing: the measurementes and forecast pandas dataframe loaded are concatenad and some check to find Nans in made
- Kpis computing: the four, before cited, kpis are computed making use of metrics module of sklearn.
- Kpis loading: through the usage of BigQuery client, kpis computed are loaded to the corresponding database, respecting the specific BigQuery table schema.

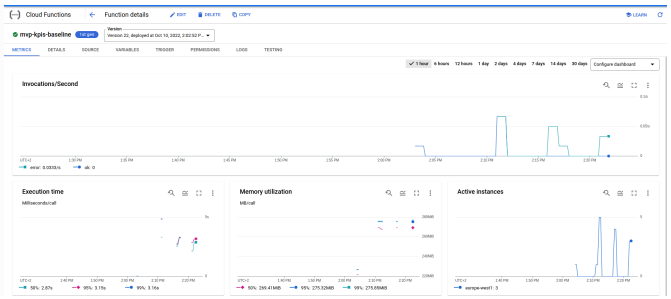

Figure 4.6: Baseline kpis cloud funtion general information

One of the purposes of the kpis pipeline is to be is to periodically scheduled so an Airflow DAG, whose details are in the following sections, has been built to automate the execution. So once the cloud function has been locally tested, a DAG periodically every week, launched the kpis batch computation on a specific site. The pipeline correctly computes and store results in BigQuey. In order to make some data analysis some basic queries have been made to inspect aggregated

results. The queries have been made in the SQL query engine built-in in BigQuery, a very handleable and fast way to make queries and inspect results. Some results of these queries are illustrated in the two following tables.

site_name	device_name	forecast_type	start_time	end_time	cv_rmse	mae	bias	max_err
site_name_1	load	day-ahead	2022-05-01 00:00:00.000000 UTC	2022-05-28 00:00:00.000000 UTC	0.277	0.545	-0.419	2.671
site_name_1	load	day-ahead	2022-05-13 15:06:27.000000 UTC	2022-06-10 15:06:27.000000 UTC	0.305	0.567	-0.489	2.671
site_name_1	load	day-ahead	2022-05-14 22:05:31.000000 UTC	2022-06-11 22:05:31.000000 UTC	0.309	0.621	-0.542	2.671
site_name_1	load	day-ahead	2022-05-21 22:15:47.000000 UTC	2022-06-18 22:15:47.000000 UTC	0.531	1.317	-1.247	5.768
site_name_1	load	day-ahead	2022-05-28 22:05:54.000000 UTC	2022-06-25 22:05:54.000000 UTC	0.468	1.104	-0.937	5.768
site_name_1	load	day-ahead	2022-06-04 22:01:05.000000 UTC	2022-07-02 22:01:05.000000 UTC	0.439	1.018	-0.786	5.768
site_name_1	load	day-ahead	2022-06-11 22:05:43.000000 UTC	2022-07-09 22:05:43.000000 UTC	0.427	0.966	-0.647	5.768
site_name_1	load	day-ahead	2022-06-18 22:05:40.000000 UTC	2022-07-16 22:05:40.000000 UTC	0.343	0.854	-0.495	4.907
site_name_1	load	day-ahead	2022-06-21 13:50:44.000000 UTC	2022-07-19 13:50:44.000000 UTC	0.315	0.783	-0.415	4.203
site_name_1	load	day-ahead	2022-06-21 14:23:00.000000 UTC	2022-07-19 14:23:00.000000 UTC	0.315	0.782	-0.415	4.203
site_name_1	load	day-ahead	2022-06-26 00:06:03.000000 UTC	2022-07-24 00:06:03.000000 UTC	0.279	0.76	-0.417	3.863
site_name_1	load	day-ahead	2022-07-03 00:00:25.000000 UTC	2022-07-31 00:00:25.000000 UTC	0.233	0.679	-0.364	3.842
site_name_1	load	day-ahead	2022-07-10 00:00:32.000000 UTC	2022-08-07 00:00:32.000000 UTC	0.22	0.695	-0.386	3.842
site_name_1	load	day-ahead	2022-07-17 00:02:05.000000 UTC	2022-08-14 00:02:05.000000 UTC	0.24	0.748	-0.133	3.764
site_name_1	load	day-ahead	2022-07-24 00:06:20.000000 UTC	2022-08-21 00:06:20.000000 UTC	0.4	0.91	0.3	6.179
site_name_1	load	day-ahead	2022-07-31 00:07:22.000000 UTC	2022-08-28 00:07:22.000000 UTC	0.45	0.982	0.201	6.179
site_name_1	load	day-ahead	2022-08-07 00:00:50.000000 UTC	2022-09-04 00:00:50.000000 UTC	0.479	0.964	0.21	6.179
site_name_1	load	day-ahead	2022-08-14 00:02:24.000000 UTC	2022-09-11 00:02:24.000000 UTC	0.556	0.951	0.134	6.179
site_name_1	load	day-ahead	2022-08-21 00:02:03.000000 UTC	2022-09-18 00:02:03.000000 UTC	0.345	0.724	-0.494	5.499
site_name_1	load	day-ahead	2022-08-28 00:01:01.000000 UTC	2022-09-25 00:01:01.000000 UTC	0.159	0.431	0.019	1.449
site_name_1	chiller1	day-ahead	2022-05-13 15:06:06.000000 UTC	2022-06-10 15:06:06.000000 UTC	0.507	0.186	-0.164	0.798
site_name_1	chiller1	day-ahead	2022-05-14 22:00:22.000000 UTC	2022-06-11 22:00:22.000000 UTC	0.512	0.218	-0.199	0.798
site_name_1	chiller1	day-ahead	2022-05-21 22:10:34.000000 UTC	2022-06-18 22:10:34.000000 UTC	0.938	1.042	-1.009	4.674
site_name_1	chiller1	day-ahead	2022-05-28 22:11:06.000000 UTC	2022-06-25 22:11:06.000000 UTC	0.878	0.833	-0.74	4.674
site_name_1	chiller1	day-ahead	2022-06-04 22:05:26.000000 UTC	2022-07-02 22:05:26.000000 UTC	0.865	0.789	-0.654	4.674
site_name_1	chiller1	day-ahead	2022-06-11 22:05:40.000000 UTC	2022-07-09 22:05:40.000000 UTC	0.862	0.758	-0.547	4.674
site_name_1	chiller1	day-ahead	2022-06-18 22:05:38.000000 UTC	2022-07-16 22:05:38.000000 UTC	0.723	0.744	-0.497	3.218
site_name_1	chiller1	day-ahead	2022-06-21 13:50:45.000000 UTC	2022-07-19 13:50:45.000000 UTC	0.684	0.716	-0.466	3.218
site_name_1	chiller1	day-ahead	2022-06-21 14:23:00.000000 UTC	2022-07-19 14:23:00.000000 UTC	0.683	0.718	-0.468	3.218
site_name_1	chiller1	day-ahead	2022-06-26 00:06:04.000000 UTC	2022-07-24 00:06:04.000000 UTC	0.607	0.742	-0.498	3.218
site_name_1	chiller1	day-ahead	2022-07-03 00:05:58.000000 UTC	2022-07-31 00:05:58.000000 UTC	0.518	0.696	-0.451	3.218
site_name_1	chiller1	day-ahead	2022-07-10 00:06:04.000000 UTC	2022-08-07 00:06:04.000000 UTC	0.487	0.728	-0.458	3.201
site_name_1	chiller1	day-ahead	2022-07-17 00:11:42.000000 UTC	2022-08-14 00:11:42.000000 UTC	0.513	0.74	-0.204	3.446
site_name_1	chiller1	day-ahead	2022-07-24 00:06:43.000000 UTC	2022-08-21 00:06:43.000000 UTC	0.646	0.772	0.109	3.446
site_name_1	chiller1	day-ahead	2022-07-31 00:02:03.000000 UTC	2022-08-28 00:02:03.000000 UTC	0.658	0.738	0.151	3.446
site_name_1	chiller1	day-ahead	2022-08-07 00:00:51.000000 UTC	2022-09-04 00:00:51.000000 UTC	0.68	0.688	0.152	3.446
site_name_1	chiller1	day-ahead	2022-08-14 00:02:34.000000 UTC	2022-09-11 00:02:34.000000 UTC	0.637	0.54	0.135	3.38
site_name_1	chiller1	day-ahead	2022-08-21 00:02:24.000000 UTC	2022-09-18 00:02:24.000000 UTC	0.371	0.394	-0.134	2.428
site_name_1	chiller1	day-ahead	2022-08-28 00:00:45.000000 UTC	2022-09-25 00:00:45.000000 UTC	0.339	0.32	0.007	1.995
site_name_2	load	day-ahead	2022-05-01 00:00:00.000000 UTC	2022-05-28 00:00:00.000000 UTC	0.311	2716.61	938.892	18308.885
site_name_2	load	day-ahead	2022-05-13 15:11:19.000000 UTC	2022-06-10 15:11:19.000000 UTC	0.349	565.82	-158.447	5713.279
site_name_2	load	day-ahead	2022-05-14 22:05:36.000000 UTC	2022-06-11 22:05:36.000000 UTC	0.55	301.883	-91.839	5482.746
site_name_2	load	day-ahead	2022-05-21 22:10:50.000000 UTC	2022-06-18 22:10:50.000000 UTC	0.707	8449.044	-7273.821	50564.253
site_name_2	load	day-ahead	2022-05-28 22:05:41.000000 UTC	2022-06-25 22:05:41.000000 UTC	0.684	8154.865	-6878.915	50564.253
site_name_2	load	day-ahead	2022-06-04 22:01:08.000000 UTC	2022-07-02 22:01:08.000000 UTC	0.641	7398.687	-5742.884	50564.253
site_name_2	load	day-ahead	2022-06-11 22:00:44.000000 UTC	2022-07-09 22:00:44.000000 UTC	0.627	6811.457	-5240.647	50564.253
site_name_2	load	day-ahead	2022-06-18 22:05:40.000000 UTC	2022-07-16 22:05:40.000000 UTC	0.532	5769.332	-4016.01	37378.661
site_name_2	load	day-ahead	2022-06-21 14:22:11.000000 UTC	2022-07-19 14:22:11.000000 UTC	0.475	5084.019	-3303.914	36826.235
site_name_2	load	day-ahead	2022-06-26 00:03:17.000000 UTC	2022-07-24 00:03:17.000000 UTC	0.431	4823.2	-2896.853	36826.235
site_name_2	load	day-ahead	2022-07-03 00:05:57.000000 UTC	2022-07-31 00:05:57.000000 UTC	0.384	4647.466	-2399.683	36826.235
site_name_2	load	day-ahead	2022-07-10 00:07:02.000000 UTC	2022-08-07 00:07:02.000000 UTC	0.41	4779.497	-953.797	28735.185
site_name_2	load	day-ahead	2022-07-17 00:01:31.000000 UTC	2022-08-14 00:01:31.000000 UTC	0.434	4341.915	62.945	28735.185
site_name_2	load	day-ahead	2022-07-24 00:07:52.000000 UTC	2022-08-21 00:07:52.000000 UTC	0.476	3950.83	1607.243	28735.185
site_name_2	load	day-ahead	2022-07-31 00:02:52.000000 UTC	2022-08-28 00:02:52.000000 UTC	0.476	3188.969	1921.496	28735.185
site_name_2	load	day-ahead	2022-08-07 00:07:16.000000 UTC	2022-09-04 00:07:16.000000 UTC	0.363	2554.521	931.169	26625.338
site_name_2	load	day-ahead	2022-08-14 00:07:11.000000 UTC	2022-09-11 00:07:11.000000 UTC	0.391	2658.885	701.368	26625.338
site_name_2	load	day-ahead	2022-08-21 00:00:54.000000 UTC	2022-09-18 00:00:54.000000 UTC	0.275	2222.475	-571.287	15850.478
site_name_2	load	day-ahead	2022-08-28 00:02:49.000000 UTC	2022-09-25 00:02:49.000000 UTC	0.256	1822.179	-1267.963	15850.478
site_name_2	hvacpbd	day-ahead	2022-05-13 15:11:19.000000 UTC	2022-06-10 15:11:19.000000 UTC	1.625	143.62	-106.654	3114.44

Table 4.1: Day-ahead baseline kpis query: load and chiller1 devices of Alamos site

site_name	device_name	forecast_type	AVG_cv_rmase	MIN_cv_rmse	MAX_cv_rmse	AVG_mae	MIN_mae	MAX_mae	AVG_bias	MIN_bias	MAX_bias	AVG_max_err	MIN_max_err	MAX_max_err
site_name	load	day-ahead	0.35	0.16	0.56	0.82	0.43	1.32	-0.37	-1.25	0.3	4.57	1.45	6.18
site_name	chiller1	day-ahead	0.64	0.34	0.94	0.65	0.19	1.04	-0.31	-1.01	0.15	3.22	0.8	4.67
site_name	load	day-ahead	0.46	0.26	0.71	4223.24	301.88	8449.04	-1822.79	-7273.82	1921.5	30500.61	5482.75	50564.25
site_name	hvacpbd	day-ahead	1.64	0.81	4.14	1011.07	143.62	1819.04	-279.6	-1798.57	1003.99	5974.06	3114.44	8271.31
site_name	hvacpbi	day-ahead	1.56	0.78	3.76	959.74	164.21	1882.68	-384.68	-1728.39	597.04	8655.49	1963.59	12100.63
site_name	hvacp12d	day-ahead	7.35	4.89	12.95	0.53	0.09	0.85	-0.47	-0.85	-0.04	39.08	8.0	68.42
site_name	hvacp12i	day-ahead	1.38	1.1	1.86	431.05	30.95	557.43	-318.97	-540.36	-30.38	3761.96	1276.0	4398.6
site_name	hvacp34d	day-ahead	0.93	0.41	1.28	347.6	184.61	558.88	-237.94	-549.66	-93.75	2628.53	804.35	3929.09
site_name	hvacp34i	day-ahead	2.25	1.84	2.9	359.49	212.47	608.17	-135.72	-521.86	121.75	6284.66	1717.08	9039.68
site_name	hvacp12cd	day-ahead	6.67	4.33	8.61	1.23	0.36	2.01	-0.54	-1.28	0.23	68.56	9.0	107.52
site_name	hvacp12ci	day-ahead	4.9	3.36	7.1	1.76	0.5	3.18	-1.43	-3.18	-0.07	66.63	15.59	90.92
site_name	hvacp34cd	day-ahead	3.09	0.04	8.91	1113.64	4.39	2021.62	-417.56	-2010.77	845.83	6463.33	11.74	9285.7
site_name	hvacp34ci	day-ahead	2.25	1.25	5.13	461.68	47.96	808.81	-84.27	-808.42	381.85	3554.05	805.58	6284.0

Table 4.2: Day-ahead baseline kpis aggregation query: kpis average, maximum and minimum, grouped by device and site

4.2 Orchestration

This is a core section as are exposed all the implementations related to Orchestration, the point where every module is scheduled and automated in order precisely carry on reliable production tasks. As a result of the previous chapter, Airflow has been chosen to be the Orchestration tool. Before focusing on DAGs implementations, multiple ways or Airflow installation are illustrated as they all have been tried, for different reasons. The history the the reasons behind different kind of installations are provided:

- Google Could Composer (Managed Airflow)
- Airflow installed in a virtual machine hosted on Google Compute Engine
- Airflow Dockerized container

As a powerful and production Airflow installation can be hard, not so much to build but instead to maintain due to its complex architecture, especially in case of failures, and this can not happen when there are production DAGs which schedule several important services. The first choice made about of Airflow mounting has been to making use of Google Cloud Composer, the managed Airflow built-in product of GCP, in which users can create an Airflow instance specifying environment resources, the kind of Scheduler, of Metastore and basically everything that is configurable. The main pro of this installation is that once the Airflow environment is created, users can forget everything about maintenance and architecture as everything is carried on by Google and, for this experience, we can say that it has worked very well. The main disadvantage is that is not cheap: a relatively small airflow instance might cost 350/400 $/month and comparing actual orchestration needs, even if free GCP credit has been used, after a while this option was discarded and Composer has been shut down.
After having discarded Composer, Airflow has been manually installed in a virtual machine hosted on Google Compute Engine. The main, which are available in the code snippets section, include virtual environment creation, dependencies

installation, Executor changed from Sequential to Local, PostGres setting as backend Metastore, and custom configurable options such as number of task retries. In order to make the webserver and the scheduler run when the terminal of the VMs, used to launch these services, are closed, webserver and scheduler are executed as daemons. This installation is very cheap but does not give many stability guarantees: in one month the environment stopped working, and so the instance has been installed again from scratch.
The third installation evolution has been made through Docker. First, a Dockerfile of an Airflow academic environment has been created, the image built and the container activated. Then, in coordination with the DevOps engineer, a more sophisticated and production-oriented Dockerized version has been mount, and it is the one the is currently in usage, hosted in Google Compute Engine. The main advantages of this installation approach are the ones related to all the benefits derived from Docker containers, such portability and fast deployment, the fact that is cheap, that for testing or developing purposes the Airflow container can be launched in local making experiments without affecting the production environment.

4.2.1 Orchestration processes definitions

Having identified the correct tool to address Orchestration, the Airflow features to be applied and to be exploited such as temporary scheduling, intra-jobs dependencies, retry policies and log monitoring, this section focuses on the different Airflow DAGs definitions related to different data and processes. The services that need to be scheduled and automated are the following:

- Baseline and Flexibility Forecast
- KPIs
- Weather data
- Market data

Regarding forecasts temporary scheduling, for every site of interest, for every corresponding device, baseline forecasts have to be scheduled according to the baseline forecast mode: in Day-ahead, once a day, otherwise,in Intra-day mode, every hour. In relation of intra-jobs dependencies, some forecasts can be done in parallel at the same time and other that have to be done in a second phase, such as batteries or meters. Regarding retry policies we are interested in retry a forecast task three times more in case of retry state of the task, in the case it definitely fails to send an internal notification, es explained in the next section. Regarding flexibility forecasts temporary scheduling, the follow the same scheduling of baseline forecasts in relation to Day-ahead on Intra-day mode but have logic dependencies

from the baseline forecast device from which they depend and so they have to made after: for example, the thermal zone 1 and 2 depend on the thermal load 0, and so their flexibility forecast must be done after and in case of success of the baseline forecast of the thermal load 0.
Baseline forecast kpis, for design choices, are scheduled once a week and compute the kpis on the past 28 days, both for Day-ahead and Intra-day mode. In this case there are not particular logic dependencies and the task retries policy is set in the same way as baseline and flexibility forecasts DAGs.
Weather forecast service is scheduled four times a day and downloads data for the following week: in this way every six hours weather forecast data are updated in order to have more reliable data to be used for baseline and consequently flexibility forecasts. There are no particular logic dependencies and the task retires policy is set in the same way as before.
Market data DAGs follow a scheduling in relation of the time of publication: they are triggered before the maximum delay that can happen when a market prices time series is publish by Esios, the web where a lot of data regarding different actors involved in energy paradigm are published. There are not particular logic dependencies dependencies and the task retires policy is set in the same way as before.
Once that the general services DAGs definitions are set, before moving to the actual implementation it is important to define Backfill and Callback policies.
Backfill is the Airflow feature that allows to re-launch DAGs runs in the past. It is available through the Airflow CLI and in the Web Interface in the re-run options. For our particular requirements, the input time parameters of the scheduled services, such as baseline, flexibility forecasts, kpis, market and weather data must be coherent to the specific DAG run date and not to the date and time when they are actually triggered. For this reason, every input time parameter that is fed for the scheduled services is computed from the specif DAG run logic data in order to make a run in the past, in case of failure, with the correct input parameters without make modifications in the DAG code.
Callback is an important feature that allows to call a specific function in relation of the particular state of the task, that can be success, retry, failure, or service level agreement missing. For every Bamboo service triggered, a callback function is called in the case of failure in order to send an internal notification as explained in the next section.

4.2.2 Orchestration processes implementations

This section is intended to give an overview of the actual implementations of the Airflow DAGS and their features. For every kind of services a particular concrete example among the production DAGS is illustrated. For every Dag, every kind of

Operator used and its purposes are explained, ass well logic dependencies and data transfer between different tasks.

Baseline and Flexibility Forecast DAG

Among baseline and flexibility forecast DAGs, a particular site has been chosen (whose name is not specified for privacy reasons), which has not not many devices and so fits well for graphical representation, in which dependencies are clearly visible. There are two different DAGs addressing this particular service, one for Day-ahead mode and one for Intra-day mode, as they have different schedule intervals, respectively once a day and every hour. In this particular DAG we can find three different main jobs that are needed to carry on forecasts:

- Authentication: as Cloud Run instances of baseline and flexibility forecast backends are private the user of the module must authenticated. There are multiple way to address this issue in Airflow without hard-coding tokens and passwords. As a design choice, it has been to make an on-demand authentication through the usage of the GCP CLI
- Time input parameters extraction: in order to correctly being able to exploit backfill Airflow feature, time input parameters for the modules triggering must be computed from the logic the of the specific DAG run
- Baseline and Flexibility modules triggering: to make a forecast for a specific device, an HTTP endpoint call to the corresponding API must be made.

In order to implement these tasks, different kind of Operators are used:

- BashOperator (Authentication): as to utilize the token to authenticate to the corresponding module a bash command with the usage of the Google Cloud CLI, the operator use is the Bashoperator as it serves for launching bash commands. The value of the output of the *gcloud auth* command, which is the token, is automatically pushed through XCom
- PythonOperator (Time input parameters extraction): as we interested to correctly compute time input parameters from the logic date of the DAG run, and so to compute some Python preprocessing, the PythonOperator addresses this kind of task. The parametes *include_context* is set to True in order to have available some variables related to the specif DAG run, such the logic date, in the task instance.
- SimpleHttpOperator (Baseline and Flexibility modules triggering): this Operator is made to make HTTP endpoints call and this is exactly what is needed to perform the forecast. To use this operator in the *http_conn_id* parameter

must be specified the id of the connection where it is stored the URL or the module. In the *endpoint* parameter it is specified the particular endpoint to call with all its path parameters. Some data input parameters regarding particular forecast features are passed to the request in the *data*. Some of them are hard-coded, others, such as start-time and end-time of the forecast are pulled from XCom. The same stands of token pueel from the BashOperator instance and passed to the *headers* parameters. All this pull XCom values are passed to parameters through the usage of JinJa templating as all these Operator parameters are templated fields. In the *on_failure_callback* parameter it is specified the function to be called in case of task failure and it is how we send an internal notification as it is described in the next section

Before getting the token for authentication purposes and start making the forecast, an API call is made to be sure that the baseline forecast module is active, through the usage of a SimpleHttpOperator. This is the starting point of these Dags. Once all Operator task instances are defined, logic dependencies are defined:

- Baseline API availability –> Baseline Authentication
- Extract time input parameters –> Baseline Forecast, Flexibility Forecast
- Baseline Authentication –> Baseline Forecast
- Flexibility Authentication –> Flexibility Forecast
- Baseline –> Flexibility logic dependencies.

The graph view is inserted to give a graphical representation.

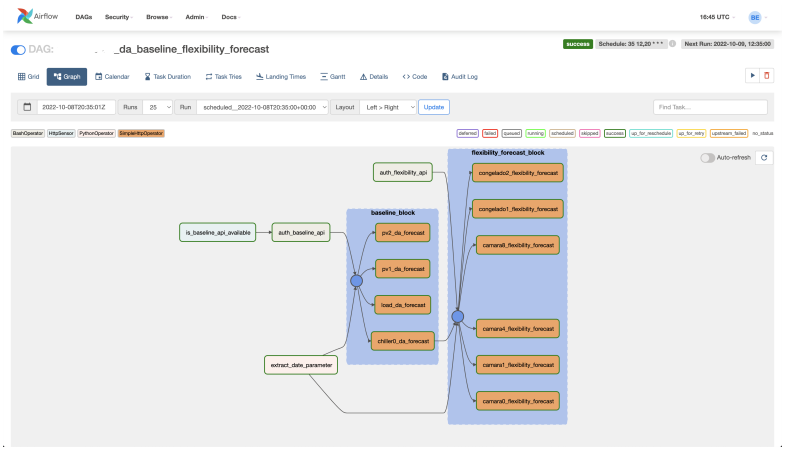

Figure 4.7: specific site Day-Ahead baseline and flexibility forecast DAG

Kpis DAG

Among baseline kpis DAGs, a particular one is chosen to be the illustrated one. Both kpis DAGs of Day-ahead baseline and Intra-day mode are scheduled once a week. Two main jobs can be identified to execute this DAG:

- Authentication: as Cloud Function instances of baseline kpis are private the user of the module must authenticated. As a design choice, it has been to make an on-demand authentication through the usage of the GCP CLI
- Time input parameters extraction: in order to correctly being able to exploit backfill Airflow feature, time input parameters for the moduls triggering must be computed from the logic the of the specific DAG run
- Baseline kpis: to make a forecast for a specific device, an HTTP endpoint call to the corresponding Cloud Function must be made.

In order to implement these tasks, different kind of Operators are used:

- BashOperator (Authentication): as to utilize the token to authenticate to the corresponding module a bash command with the usage of the Google Cloud CLI, the operator use is the Bashoperator as it serves for launching bash commands. The value of the output of the *gcloud auth* command, which is the token, is automatically pushed through XCom
- PythonOperator (Time input parameters extraction): as we interested to correctly compute time input parameters from the logic date of the DAG run, and so to compute some Python preprocessing, the PythonOperator addresses this kind of task. The parametes *include_context* is set to True in order to have available some variables related to the specif DAG run, such the logic date, in the task instance.
- SimpleHttpOperator (Baseline kpis): this Operator is made to make HTTP endpoints call and this is exactly what is needed to perform the forecast. To use this operator in the *http_conn_id* parameter must be specified the id of the connection where it is stored the URL or the module. Some data input parameters regarding particular kpis module requirements such as site id and device name are passed to the request in the *data*. Some of them are hard-coded, others, such as start-time and end-time of the kpis computation are pulled from XCom. The same stands of token pulled from the BashOperator instance and passed to the *headers* parameters. All this pulled XCom values are passed to parameters through the usage of JinJa templating as all these Operator parameters are templated fields. In the *on_failure_callback* parameter it is specified the function to be called in case of task failure and it is how we send an internal notification as it is described in the next section

. Once all Operator task instances are defined, logic dependencies are defined:

- Authentication -> Extract time input parameters
- Extract time input parameters -> Kpis computation

Market prices DAG

Among all several market prices DAGs, which have different schedule intervals due to different publication times, the Spain Spot market is chosen as a sample. This series is published once a day and consists of the following day energy prices for the retailers companies in the Spain market. Apart the different schedule intervals, the programming logic is very similar to the kpis DAG as market prices module is a Cloud Function and the authentication process follow the same logic. In order to not be too repetitive, is is omitted the description of the jobs and the operators used as they are the same of the previously exposed DAGs: BashOperator. PythonOperator and SimpleHTTPOperator. The graph view is inserted to give a graphical representation.

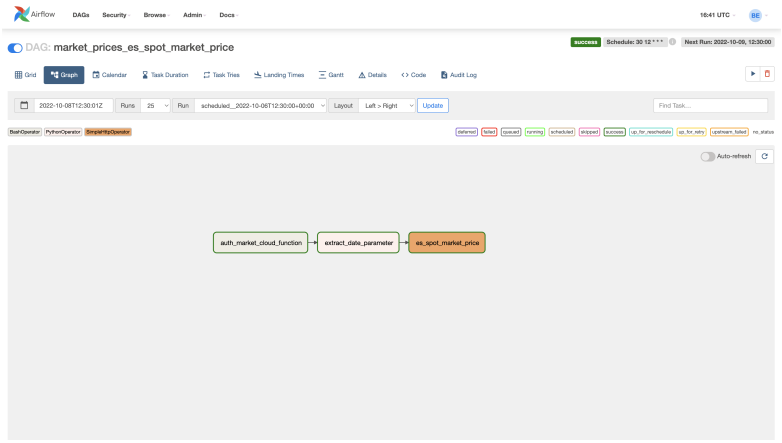

Figure 4.8: market prices DAG

Weather forecast DAG

There is only one weather forecast DAG, which is scheduled every six hour. Apart the schedule intervals, the programming logic is very similar to the kpis DAG and market prices DAG as weather forecast module is a Cloud Function and the authentication process follow the same logic. In order to not be too repetitive, is is omitted the description of the jobs and the operators used as they are the same of the previously exposed DAGs: BashOperator. PythonOperator and SimpleHTTPOperator. The graph view is inserted to give a graphical representation.

4.3 Alerting

As a result of of the previous chapter, in this section the Alerting policies definitions and implementations are described. From one hand, some events related to devices are detected and alerts sent internally or externally, through the implementations of Beam pipelines, on the other hand, modules executions, which are managed by Airflow, are monitored making use of callback functions, sending internal notifications.

4.3.1 Alerting policies definitions

This section focuses in the technologies used for internal notifications, which is a mixed approach with Google Error Reporting and Slack notification API. The external notifications are planned with the usage of an email provider but currently are not yet implemented: the simulation is carried only by internal logging detection.

Devices measurements and execution events events

Two main events have been identified to implement policies, for both internal and external notifications:

- A devices sends a out-of-bounds measurements
- A devices has stopped or restarted sending data

As we have planned to utilize PubSub and Beam/Dataflow to manage data cleaning and loading, it has been decided to include in this architecture these two events detection as we are interested in scalable environment, almost real-time notification in streaming mode as we approach data ingestion in streaming. The processes execution monitored are all the services whose execution is managed by Airflow:

- market data download failed (internal notifications)
- weather data forecast failed (internal notifications)
- baseline and flexibility forecast training (internal notifications)
- baseline and flexibility forecast fail (internal notifications)
- kpis computation failed (internal notifications)

After the retries attempts of Task, when its state is set to failure by the Scheduler, a callback function is triggered sending and internal notification as explained in the next section.

4.3.2 Alerting policies implementations

In this section are described the actual implementations of internal notifications, making use of Google Error reporting and Slack notifications API in callback functions of Airflow tasks, and external notification, whose goal is to implement an Email service to which flexumers subscribe: this feature is in development phase and for these experiments the correspondent events are detected and internally logged to implement a solid logic for event detection and lay the foundations to future developments and improvements.

Devices measurements events detection

In this subsection are exposed the approaches to detect measurements events in the Beam/Dataflow paradigm. For what concerns out-of-bounds values detection there is no need to implement a new pipeline as this kind of event is already detected in the cleaning and loading pipeline: as we are reading incoming PubSub messages with a Global Window policy, and so processing is triggered as soon as a new measurement is ingested, in the case of an out-of-bounds value the data is not stored in the the DB but the event is internally log, to test this implementation, and, when the the Email notification service is implemented, the corresponding API endpoint to send external and internal notification will be called.

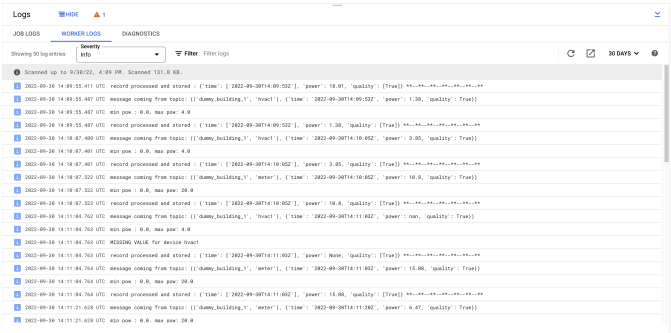

Figure 4.9: Cleaning, loading and out-of-bounds event detection logs

To detect when a particular device has stopped or restarted sending data a different pipeline has been created as it makes use of different logic, like Sliding Windows policy for example. It consists of a streaming pipeline that is listening for incoming PubSub messages: in other words is a parallel streaming pipeline listing for the same data as the cleaning and loading pipeline. Telemetry start and telemetry stop are the names given to these events and are detected in the same pipeline.
In order to correctly implement this Telemetry pipeline, the sending data frequency of the devices of a site is fundamental to reliably implement Sliding Window policies. In the parsed input parameters, apart for general information needed

for the pipeline itself to be executed in the Dataflow Engine, are also included parameters to build the sliding windows:

- window size: the length in seconds of the window
- window period: the step of the slide of the window
- window split: the amount of seconds to split the windows in sub-windows. Usually it is the half of window size, as in this experiment. However it is has been left to be a custom parameter, as different sub-windows logic could be implemented.

The telemetry pipeline consists of the following steps:

- Read from Pub/Sub topic: through the usage of Beam io library, more precisely of PTransorm ReadFromPubSUB which allows to read utf-8 string payloads from Cloud Pub/Sub, the pipeline is able ingest PubSub messages from a PubSub source, which in this case is a topic as the specific topic to carry on the simulation is specified in the input parameter of the PCollection
- Covert bytes to dictionary: making use of the classical Map PTransform, each incoming record is decoded to a dictionary from utf-8 encoding, through the usage of a simple lambda function.
- Map for events detection requirements: for each incoming massage, some useful information are extracted and it is emitted a key-value pair whose key is a tuple ,containing the start and end timestamp of the corresponding window, site and device name, and value is a tuple with the timestamp associated to the PubSub ingestion and the threshold timestamp of the specific window in in order to correctly establish whether the measurement belongs to the first-half sub-window or to the second one.
- Sliding window application: on the streaming flow is applied the Sliding Window policy with the related parsed arguments as input parameters
- GroupbyKey: this classic transformation is applied to group all the elements with the same key, and so all the same devices measurements arrived in a specific window.
- Telemetry detection: a custom DoFn is created to be applied on the grouped elements. Comparing values of the grouped elements, and so ingestion timestamp and threshold timestamp of the windows, are counted the elements arrived respectively in the first or in second sub-half windows:

– if the number of counted elements of the first sub-half window is positive and there are not elements in the second half, the event Telemetry Stop is detected and internally logged, and in the future the endpoints to send the corresponding notification mail will be called
– if the number of counted elements of the second sub-half window is positive and there are not elements in the first half, the event Telemetry Start is detected and internally logged, and in the future the endpoints to send the corresponding notification mail will be called

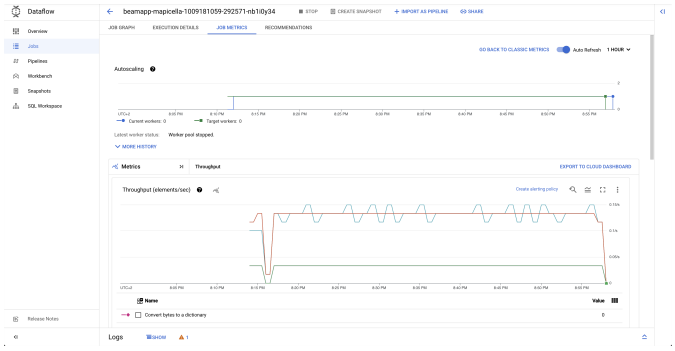

Figure 4.10: Telemetry pipeline throughput

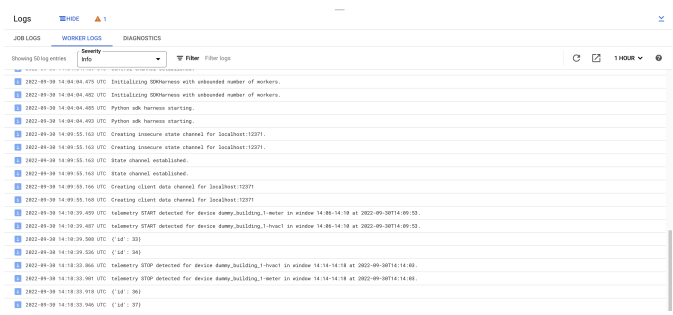

Figure 4.11: Telemetry events logs

Modules execution failures detection

These kinds of events are detected in Airflow, as it is the Orchestrator of all the services. A python script is coded and uploaded in the DAGs folder in order to be globally imported by all the DAGs and used as callback function in case of failure of the SimpleHttpOperator that calls a Bamboo module on GCP, which is Cloud Function or a Cloud Run instance. This function makes use of two different libraries to send internal notifications:

- Google Error Reporting: this library is used to send the log the corresponding exception of the failed task. As the corresponding module called, which is hosted on GCP, has already the corresponding log of the failed module call. we basically already have this failed event logged in Error Reporting, but it has been decided to send alnso the corresponding information of the specific task id, DAG id and DAG run to correctly isolate the event
- Slack library: this API is used to send the corresponding log of the specific Airflow task failed in the dedicated Slack notification channel in order to immediately receive a sound and visual alert everytime a module called through Airflow task, after the retries option, it is definitively marked as failed, to be able to be immediately be aware and understand the causes and if, necessary, begin to debug and improve product features.

Chapter 5

Conclusion

The idea behind this thesis work is to create an end-to-end Data Engineering architecture for this particular use case for Bamboo Energy, a tech start-up operating in the energy sector. The development of this work has been made possible first doing an internship and then being included in the product team. The overall research and implementation process can be conceptually divided in four different phases. The first step consisted in the study of the high level processes of Bamboo Energy, its main products and its features, its placement in the industry sector and the figure involved in the energy paradigm. After, the focus has been moved in topics related to Data Engineering, covering ETL and aggregation pipelines, Orchestration and Alerting processes. The goal was to make a snapshot of Bamboo Energy situation related to these fields, in order to identify strengths and weaknesses, to have a base from which begin a research and exploration work. The third phase can be identified in the pure exploration of possible new paths, comparing and studying different technologies. Finally, among different solutions, some have been chosen to be implemented, utilizing as methodology product and budget constraints, platform and cloud environment compatibility. The outcome is a very practical end-to-end use case covering all aspects from data sources ingestion to transformation and aggregation pipelines, from processes Orchestration to Alerting policies.

Appendix A

Code Snippets

```
    import json
import logging
from datetime import datetime, timedelta

from airflow.models import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator
from airflow.providers.http.operators.http import SimpleHttpOperator
from airflow.providers.http.sensors.http import HttpSensor
from airflow.utils.task_group import TaskGroup

from dags.report_error import report_error

import dateutil.tz as tz

default_args = {'start_date': datetime(2020, 1, 1)}


def extract_date_parameter(**kwargs):
    ts = kwargs['ts']
    ti = kwargs['ti']
    run_id = kwargs['run_id']
    logging.info(f'run id : {run_id}')
    if run_id.startswith('scheduled'):
        logical_date = datetime.strptime(ts, '%Y-%m-%dT%H:%M:%S%z')
    elif run_id.startswith('manual'):
        logical_date = datetime.strptime(ts, '%Y-%m-%dT%H:%M:%S.%f%z'
    )
        logical_date = logical_date.replace(microsecond=0)
    else:
        raise NotImplementedError(
```

```
            'only scheduled and manual DAG runs are implemented'
        )
    logical_date_local_time = logical_date.astimezone(
        tz=tz.gettz('Europe/Madrid')
    )
    # start_date_local_time = logical_date_local_time
    if run_id.startswith('scheduled'):
        if 0 <= logical_date_local_time.hour <= 14:
            # this corresponds to run scheduled at 20:35 UTC
            # this is a situation where logical data and start date are
            #   in the same day as the schedule interval is twice a day
            start_date_local_time = logical_date_local_time.replace(
                hour=22, minute=0, second=0, microsecond=0
            )
        elif 15 <= logical_date_local_time.hour <= 22:
            # this corresponds to run scheduled at 14:35 UTC
            # this is a situation where logical data and start date have
            # different (consecutive) days as the schedule interval is twice
            # a day
            start_date_local_time = logical_date_local_time + timedelta(days=1)
            start_date_local_time = start_date_local_time.replace(
                hour=14, minute=0, second=0, microsecond=0
            )
    else:
        start_date_local_time = logical_date_local_time

    start_date_utc = start_date_local_time.astimezone(
        tz=tz.gettz('utc')
    ).strftime('%Y-%m-%dT%H:%M:%SZ')
    logging.info(
        f'start_date parameter of baseline forecast (local time): '
        f'{start_date_local_time} '
    )
    logging.info(
        f'start_date parameter of thermalzone flexibility forecast'
        f'(local time): {start_date_local_time} '
    )

    ti.xcom_push(key='start_date', value=start_date_utc)

    logging.info(
        f'start_date parameter of baseline forecast (UTC): {start_date_utc}'
    )
```

```
74      logging.info(
75          f'start_date parameter of thermalzone flexibility forecast'
76          f'(UTC): {start_date_utc} '
77      )
78
79
80  with DAG(
81      'site_example_da_baseline_flexibility_forecast',
82      default_args=default_args,
83      schedule_interval='35 12,20 * * *',
84      catchup=False,
85      tags=[
86          'baseline_forecast',
87          'day-ahead',
88          'flexibility_forecast',
89      ],
90  ) as dag:
91      baseline_training_sites = {
92          'site_name': ('1', ['load', 'chiller0', 'pv1', 'pv2']),
93      }
94
95      auth_baseline_api = BashOperator(
96          task_id='auth_baseline_api',
97          bash_command=(
98              'gcloud auth print-identity-token '
99              '"--audiences=XXXX"'
100         ),
101     )
102
103     token_baseline = (
104         "{{ task_instance.xcom_pull(task_ids='auth_baseline_api') }}"
105     )
106
107     is_baseline_api_available = HttpSensor(
108         task_id='is_baseline_api_available',
109         http_conn_id=XXXXXX,
110         endpoint='/ping',
111     )
112
113     extract_date_parameter = PythonOperator(
114         task_id='extract_date_parameter',
115         python_callable=extract_date_parameter,
116         provide_context=True,
117     )
118     auth_flexibility_api = BashOperator(
119         task_id='auth_flexibility_api',
120         bash_command=(
121             'gcloud auth print-identity-token '
122             '"--audiences=XXXXXX'
```

```
123             ),
124         )
125
126         token_flexibility = (
127             "{{ task_instance.xcom_pull(task_ids='auth_flexibility_api')
        }}"
128         )
129
130         flexibility_thermal_zones = [
131             'camara1',
132             'camara0',
133             'camara4',
134             'camara8',
135             'congelado1',
136             'congelado2',
137         ]
138         start_date = (
139             "{{ task_instance.xcom_pull(task_ids='extract_date_parameter
        ', "
140             "key='start_date') }}"
141         )
142         with TaskGroup('flexibility_forecast_block') as
        flexibility_forecast_block:
143             for thermal_zone in flexibility_thermal_zones:
144                 data = {
145                     'forecast_type': 'day-ahead',
146                     'frequency': 'quarter',
147                     'ini_prediction': start_date,
148                 }
149                 current_task = SimpleHttpOperator(
150                     task_id=f'{thermal_zone}_flexibility_forecast',
151                     http_conn_id='XXXXXX',
152                     endpoint='XXXXXXXXX',
153                     method='POST',
154                     data=json.dumps(data),
155                     headers={'Authorization': 'Bearer ' +
        token_flexibility},
156                     log_response=True,
157                     on_failure_callback=report_error,
158                 )
159         with TaskGroup('baseline_block') as baseline_block:
160             for device in baseline_training_sites['site_name'][1]:
161                 if device in ('load', 'chiller0'):
162                     data = {
163                         'frequency': 'quarter',
164                         'interpolation_threshold': 60,
165                         'da_forecast_horizon': 1,
166                     }
167                     current_task = SimpleHttpOperator(
```

```
168                     task_id=f'{device}_da_forecast',
169                     http_conn_id='XXXXXX',
170                     endpoint='XXXXXXXX',
171                     method='POST',
172                     data=json.dumps(data),
173                     headers={'Authorization': 'Bearer ' +
    token_baseline},
174                     log_response=True,
175                     execution_timeout=timedelta(minutes=20),
176                     on_failure_callback=report_error,
177                 )
178                 if device == 'chiller0':
179                     current_task >> flexibility_forecast_block
180             elif device in ('pv1', 'pv2'):
181                 data = {'forecast_frequency': 'quarter'}
182                 current_task = SimpleHttpOperator(
183                     task_id=f'{device}_da_forecast',
184                     http_conn_id='XXXXXXXX',
185                     endpoint='XXXXXXXX',
186                     method='POST',
187                     data=json.dumps(data),
188                     headers={'Authorization': 'Bearer ' +
    token_baseline},
189                     log_response=True,
190                     execution_timeout=timedelta(minutes=20),
191                     on_failure_callback=report_error,
192                 )
193
194     is_baseline_api_available >> auth_baseline_api >> baseline_block
195     auth_flexibility_api >> flexibility_forecast_block
196     extract_date_parameter >> [baseline_block,
    flexibility_forecast_block]
```

Bibliography

[1] John Cook et al. «Consensus on consensus: a synthesis of consensus estimates on human-caused global warming». In: *Environ. Res. Lett.* 11.5 (1995), pp. 1244–1245 (cit. on p. 1).

[2] United Nations. *What are the Sustainable Development Goals?* URL: https://www.undp.org/sustainable-development-goals (cit. on p. 1).

[3] Copernicus. *WCopernicus: 2020 warmest year on record for Europe; globally, 2020 ties with 2016 for warmest year recorded.* URL: https://climate.copernicus.eu/copernicus-2020-warmest-year-record-europe-globally-2020-ties-2016-warmest-year-recorded (cit. on p. 2).

[4] Google. *Google Cloud Run.* URL: https://cloud.google.com/run/docs/overview/what-is-cloud-run (cit. on p. 14).

[5] Google. *Google Cloud Function.* URL: https://cloud.google.com/functions/docs/concepts/overview (cit. on p. 15).

[6] Google. *Google Cloud Storage.* URL: https://cloud.google.com/storage/docs/introduction (cit. on p. 15).

[7] Google. *Google Cloud Compute Egine.* URL: https://cloud.google.com/compute/docs (cit. on p. 16).

[8] Google. *Google Cloud SQL.* URL: https://cloud.google.com/sql/docs/introduction (cit. on p. 16).

[9] Google. *Google Cloud Scheduler.* URL: https://cloud.google.com/scheduler/docs/overviewn (cit. on p. 16).

[10] James Densmore. *Data Pipelines Pocket reference. Moving and Processing Data for Analytics.* Sebastopol, CA: O'REALLY, 2021 (cit. on p. 23).

[11] Ben G. Weber. *Data Science in Production: building Scalable Model Pipeline with Pyhton.* leanpub platform: leanpub platform, 2021 (cit. on p. 23).

[12] Google. *Google Cloud Dataproc.* URL: https://cloud.google.com/dataproc/docs/concepts/overview (cit. on p. 26).

[13] Google. *Google Cloud Dataflow*. URL: `https://cloud.google.com/dataflow/docs` (cit. on p. 27).

[14] Apache Foundation. *Beam programming guide*. URL: `https://beam.apache.org/documentation/programming-guide/` (cit. on pp. 29, 30).

[15] Google. *Google Cloud BigTable*. URL: `https://cloud.google.com/bigtable/docs` (cit. on p. 36).

[16] Google. *Google Cloud BigQuery*. URL: `https://cloud.google.com/bigquery/docs` (cit. on p. 37).

[17] Ben Hanrenslak Julian de Ruiter. *Data Pipelines with Apache Airlfow*. Reading, MA: Manning, 2021 (cit. on pp. 38, 42).

[18] Apache Foundation. *Airflow programming guide*. URL: `https://airflow.apache.org/docs/apache-airflow/stable/index.html/` (cit. on pp. 38, 40).

[19] Google. *Google Cloud Error Reporting*. URL: `https://cloud.google.com/error-reporting/docs` (cit. on pp. 43, 44).