Project Number: 774571
Start Date of Project: 2017/11/01
Duration: 48 months

Type of document 3.2 – V1.0

Document title

Dissemination level	PU
Submission Date	2020-04-30
Work Package	WP3
Task	T3:3
Type	Report
Version	1.0
Author	Emanuele Graziani, Silvia Samà, Renzo Fabrizo Carpio
Approved by	Andrea Gasparri + PMC

Executive Summary

This document aims at providing a description of the design and implementation of the environment for data management. In particular, the following aspects have been addressed in order to develop a data management system that would meet the requirements of the proposed PANTHEON SCADA architecture:

I. Software Architecture;
II. Technological Solutions;
III. Data Model.

Table of Content

Abbreviations and Acronyms

AMGA	Annotated Model Grant Agreement
BEN	Beneficiary
BSON	Binary JSON
CA	Consortium Agreement
CO	Coordinator
DCP	Data Collection and Pre-processing
DSP	Data Storage and Processing
DT	Data Transfer
DoA	Description of Action
EC	European Commission
GEOJSON	Geographic JSON
GEXF	Graph Exchange XML Format
GUI	Graphical User Interface
IOT	Internet of Things
JSON	JavaScript Object Notation
MEAN	MongoDB, Express.js, Angular, Node.js
NAS	Network Attached Storage
PR	Periodic Report
ROS	Robot Operating System
SyGMa	System for Grant Management
TS	Technical Staff
UAV	Unmanned Aerial Vehicle
UGV	Unmanned Ground Vehicle
WP	Work Package
XML	eXtensible Markup Language

1 Introduction

One of the main objectives of PANTHEON is to support the decisions of agronomists and farmers leveraging on the large quantity of data that is collected in an orchard by field-based sensors, weather stations, and both terrestrial and aerial robots.

This is a typical scenario of big data analysis, which requires to address the following, well known, main V's challenges:

- Volume: the size of collected data increases fast and can rapidly become very large, reaching a dimension that traditional database systems are not capable of managing and processing in an efficient way;
- Velocity: Data are generated at high speed and, especially for monitoring purposes, need to be processed in real-time, as soon as they arrive;
- Variety: data is produced by different systems, are heterogeneous by nature (e.g., records, images, laser scans), and rely on different formats, but they need to be reconciled and integrated to provide better insights to decision makers.

In order to consider all of these aspects, we need an environment for data analysis able to satisfy the following technical requirements:

- It must be able to operate both in real-time, for the monitoring of plantations, and in batch mode, for the processing of large collections of historical data oriented to predictive analysis and support of strategic decisions;
- It must guarantee low latency (response time of an analysis query), high throughput (number of operations performed over a period of time) and fault tolerance (reliability in case of software and hardware malfunctions);
- It should allow the applications to scale smoothly when the volume of data increases rapidly,

In the rest of this document we will describe in detail the whole architecture and the main features of a software system for data collection of analysis that we have designed and developed for PANTHEON, showing how it satisfies all the above requirements.

The main aspects of the systems are the following:

- Data is distributed and replicated across computer clusters to ensure application scalability and to increase fault tolerance and data availability;
- Data management and analysis is executed in a distributed processing environment relying on the above-mentioned computer cluster;
- The data is stored in JSON, an open standard file format that provides the needed flexibility for storing different types of data;
- MongoDB, a NoSQL document-oriented database program that natively store and manage data in JSON format, is used as database management system: it guarantees the needed efficiency and scalability in a distributed environment;
- Hardware and software resources are virtualized by adopting the cloud computing paradigm.

2 Software Architecture

The architecture of the data collection and processing system capable of meeting the above requirements is shown in schematic form in Figure 1 and is composed of three main components, which implement three operational levels [1]:

- The "Data Collection and Pre-processing" layer (DCP layer in the following): this component is replicated for each hazelnut field and is dedicated to the collection of data from the various sources located in the field: sensors, weather stations, ground robots (UGV) and drones (UAV).
- The "Data Transfer" layer (DT layer in the following): this is a middleware that deals with the transfer of data between the other two levels, in both directions, and between the overall system and the final users of the software;
- The "Data Storage and Processing" layer (DSP layer or center in the following): it consists of a centralized unit in which all the data coming from the various DCP components are stored and on which massive analyses are carried out, mainly for knowledge extraction and decision support.

In the following, these three components will be described in more detail.

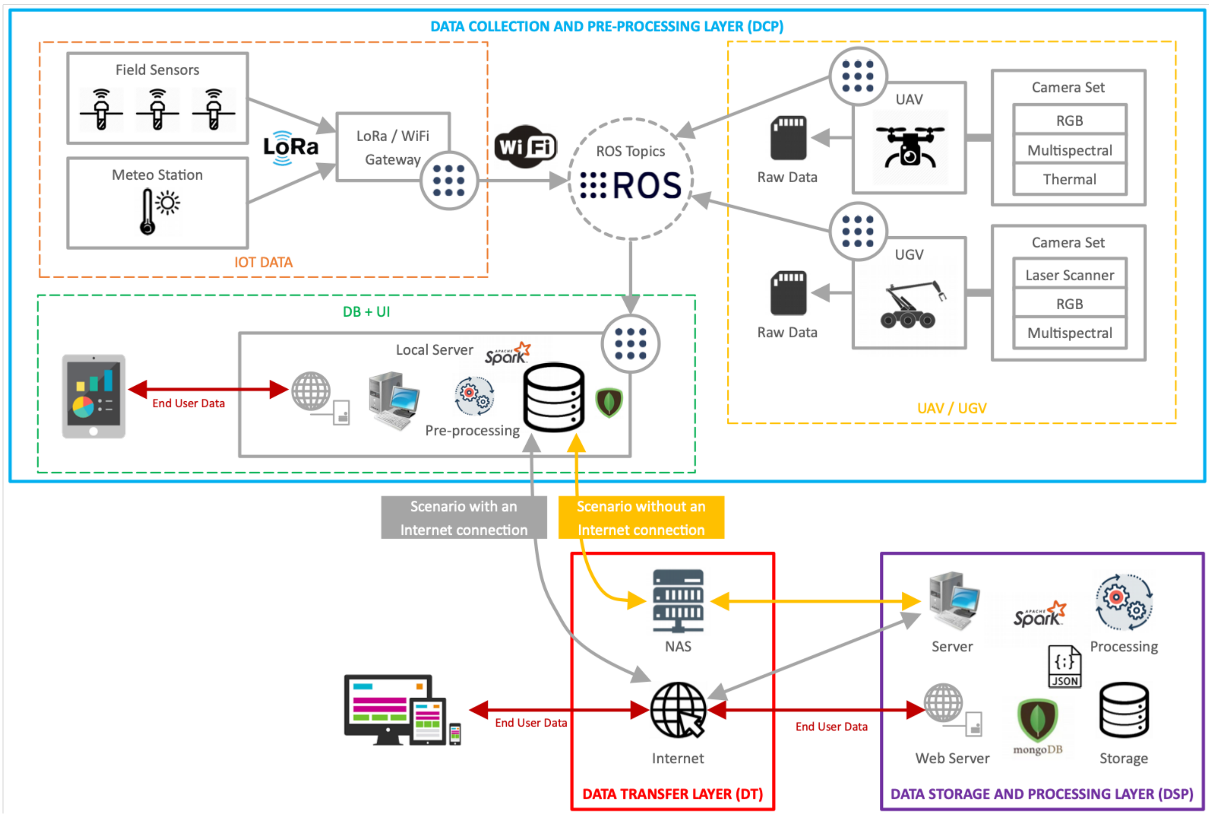


Figure 1 - The global architecture of the software system

2.1 Data Collection and Pre-processing Layer

Through a local communication network, the data coming from the collection nodes (sensors, weather stations, UGV and UAV) will be conveyed to the local server located in the warehouse near the hazelnut fields. The ROS protocol is used for data communication, as it is able to manage data transfers with all the collection nodes mentioned above (including the IoT nodes via a gateway with the LoRa network) and is based on the publish/subscribe mechanism, which allows the decoupling between data collection and data processing. However, data can also be stored on the internal mass storage of the various devices and then transferred manually to the local server. On the one hand, this guarantees the possibility of not losing acquired data even in the event of a malfunction of the communication network and, on the other hand, this guarantees the possibility of not occupying excessively the communication band, for example in the case of acquisitions of large spectral images by the UGVs.

The local server acts as a first point of collection and management of all the data coming from one hazelnut field. It is configured as a ROS node to communicate with the various collection nodes and will store data using MongoDB, a NoSQL database system. This choice was dictated by the amount of data to be managed, by their heterogeneity, and by the need to scale nicely as data volumes increase. MongoDB lends itself very well to IoT applications, especially those framed in the smart-farming area. All raw data acquired from the field will be stored on the database together with the result of data processing carried out locally or in the data storage level, as described below.

More specifically, on this system will be carried out some pre-processing activities aimed at:

- carrying out operations of data cleaning and transformation, oriented for example to eliminate grossly incorrect data and to standardize formats.
- carrying out pre-aggregations to reduce the amount of data to be transmitted to the DSP layer and to make them more suitable for the analyses to be carried out.
- performing, through a local software application, monitoring activities on the collected data and provide information to the farmers on the status of the field in real-time.

The local application will be Web-based, in order to be accessible using various types of devices and will be developed using big data technologies for processing large quantities of data at high speed. This application can be accessed directly by the operators in the field using the local server or using mobile devices, such as tablets and smartphones. An Internet connection is not required to access the application since it operates on the local database and so the network available in the field can be used for this purpose.

2.2 Data Transfer Layer

Data exchange between the database, stored in the local server, and the central database, located in the DSP layer, will occur using an Internet connection when available. If the area is not covered by an Internet connection, a portable device equipped with a large mass storage device, called NAS (Network-attached storage), will be used for data transfer. In this case, the NAS device will be physically transported from the hazelnut field to the central database. Figure 2 shows the two communication scenarios: with and without the presence of an Internet connection.

In both cases, only the data collected from the last data transfer (usually called Δ-data) is copied. In the first scenario, Δ-data is directly transferred from the local to the central database and added to the "Global Collected Data" (1). The results of data analysis carried out in the DSP center are stored in a special archive called "Global Processed Data" (2). The results obtained from Δ-data (called "New Processed Data" in Figure 2 are transferred back to the local server (3) so that they can be exploited by users operating on the field even when the DSP center is not directly accessible or the communication is low. In the second scenario, data transfer needs an intermediate step involving the storage and the transport of Δ-data in NAS devices.

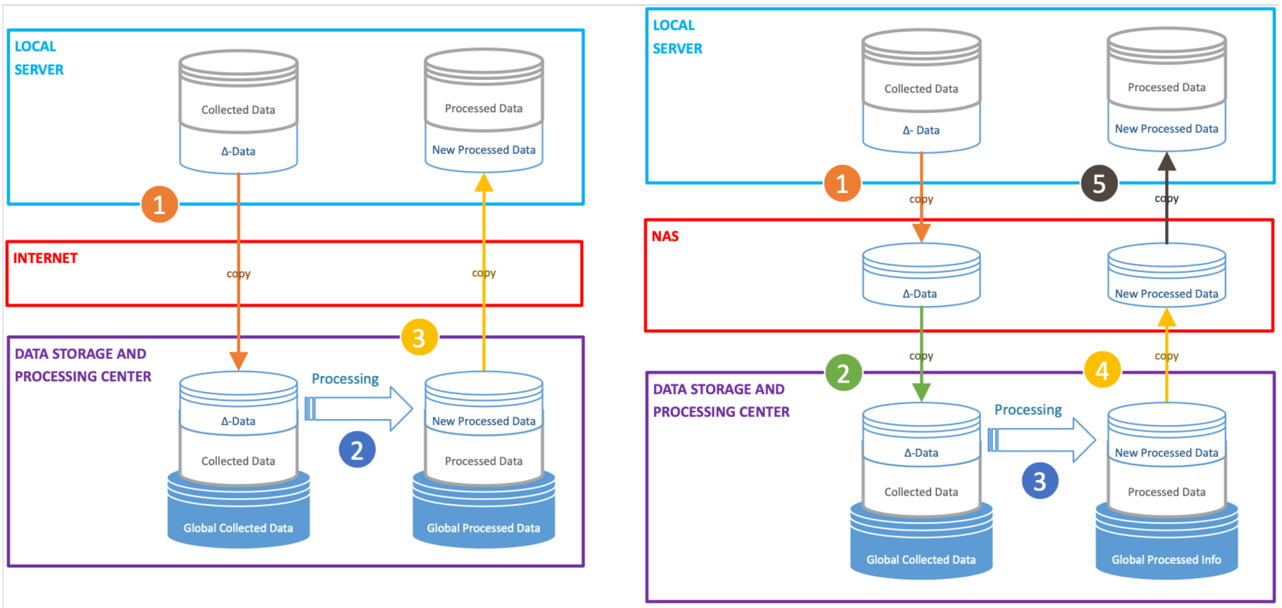


Figure 2 - Data exchange between the DCP and the DSP components.

2.3 Data Storage and Processing Layer

The DSP center is equipped with a computer infrastructure that is based on a cluster of computers whose nodes can be dynamically increased according to the requirements of storage and processing of the overall application. These requirements are driven by: the volume of data to be stored, the data replication policies, the physical distance between the DSP center and the hazelnut fields (e.g., located in different countries) that can be relieved by geographical clustering, and the need to support high workloads of data processing.

The computing nodes of the cluster will be equipped with CPUs supporting parallel computation and with a RAM and a mass storage of a size suitable for the overall needs of data storage and processing. All the collected data will be also stored in a MongoDB database, in order to be easily exchanged with the databases stored in local servers of the DCP layer. Data processing and analysis is activated at the DSP center when new raw data arrives from the DCP layer. The results of data processing are stored in the database itself.

All these choices follow the so called “data lake” approach, in which a large repository is used for storing any kind of data, coming from different sources and possibly heterogeneous, for later use, aimed usually at knowledge extraction.

2.4 Edge and Cloud Data

Pantheon architecture reflects a typical IoT (Internet of Things) architecture [2]. The system consists of a centralized component (DSP Center) that resides in the Cloud (private server in this scenario) part and, potentially, by many DCPs (one for each farm) representing the Edge nodes of the IoT system.

This structure allows to define two operating modes of the system, namely Farm Mode and Global Mode as shown in Figure 3.

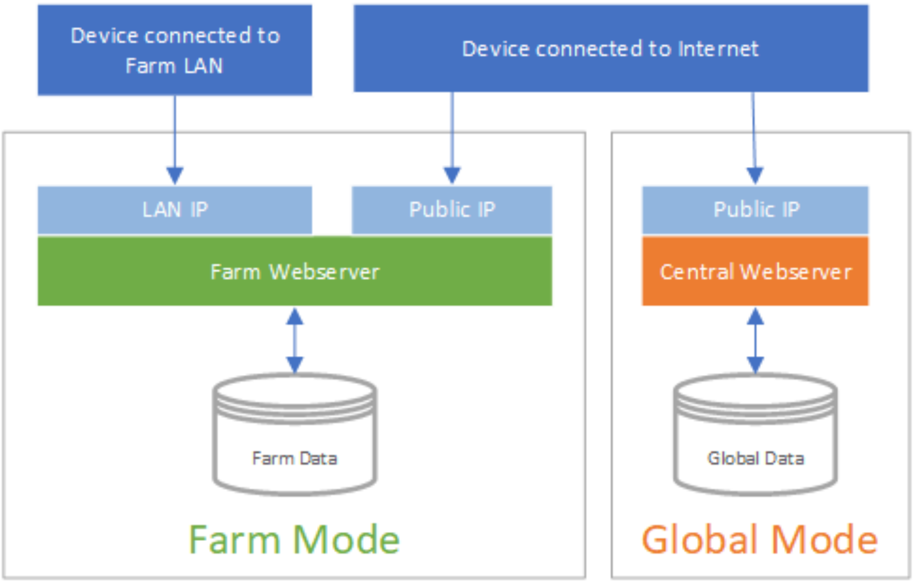


Figure 3 - Farm and Global mode schema

The operating mode is determined by the used database, specifically, if the user access to the farm database (Edge node) the system works in Farm Mode, while if the user access to the central database the system works in Global Mode.

As in IoT architectures, where only local information is kept in the Edge nodes, the data of the farm is stored in the database of each farm. Instead, the central database collects data from all potential farms that are part of the Pantheon system.

Unlike IoT systems, the Pantheon system continues to operate independently, on the single farm, even without an Internet connection. The operating of the system, with this hybrid approach, is guaranteed by the Data Transfer Layer described in previous paragraphs.

2.5 Interface to GUI

PANTHEON system needs to make data available to the users and needs to store users input to the database. This role is played by the user's application, composed by front-end (GUI) and back-end elements.

Back-end receives requests from front-end. These requests get through web services functions that let both sides transfer data in a secure and simple way. Several versions of web service features are available on market nowadays. PANTHEON back-end will be compliant to REST protocols.

A web service is a web function that listens to new data arrival continuously and, when new requests have been sent, transmits data to the software body to let it elaborate and reply. Reply modes can be synchronous or asynchronous, meaning that, in the first option, service stays in pending status until the reply is ready to be broadcast and, after sending operation is over, comes back to listening level, while in the second, listening agent closes connection every time a new request is completed and returns ready to receive new requests. Replies will be sent when ready through a different service (sender should be polling on another service after request has been completed).

Front-end architecture requires that different services should be waiting for requests from front-end. These services might be synchronous or asynchronous depending on front-end needs.

In order to send replies, back-end should interact with the data storage layer. The use of MongoDB drivers will be necessary to complete that task. The back-end environment lets developers use a specific library that can manage all the communication features with that specific database. So, any bidirectional interaction with the database (storing or retrieving data) can be easily managed by resorting to this library. The back-end will use this feature to manage required data and interact with selected databases through the MongoDB universe.

The notification management of required information to the front-end is also in charge to the back-end. It uses just a subscriber role in the ROS environment in this task. This means that when a notification that is supposed to reach the front-end layer travels through the ROS communication environment, the back-end should be able to catch it and deliver to the front-end properly. This can be accomplished by the back-end ROS toolkit for the interaction with the ROS environment and using a web socket feature for sending data to the front-end.

2.6 Data flow

In the previous paragraphs, the system architecture was described focusing on data management. This paragraph illustrates the flow of the data, from its acquisition to its processing.

In the diagram in Figure 4, it can be seen the path of the data through the various levels: Data source, Local server, Analysis Server and Analysis (Processing).

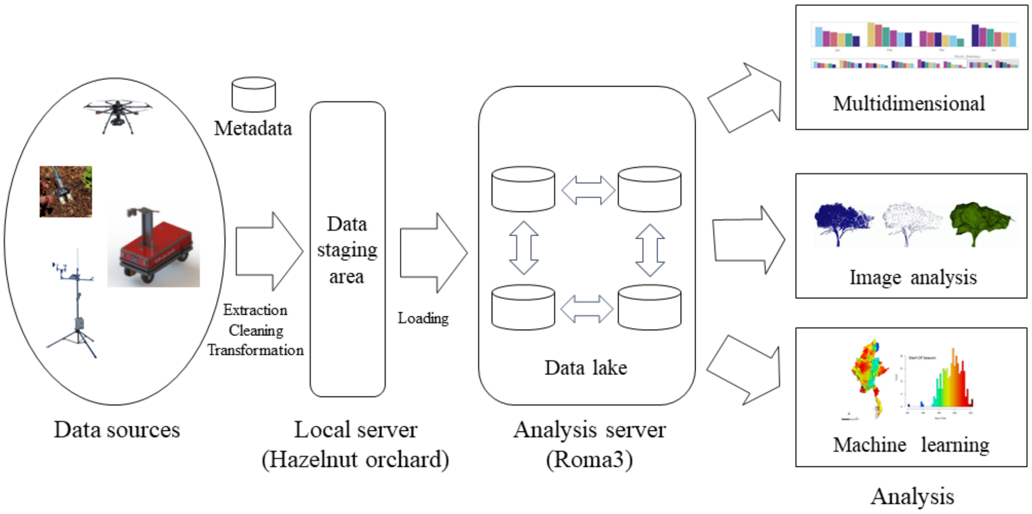


Figure 4 - Acquisition data flow schema

The data is collected by the sensors, installed on the UAV and UGV platforms, during the acquisition missions. In addition, weather station and soil sensors collect data continuously. All collected data are transferred to the farm server and then transmitted to the central server. As described in previous sections, the data can be transferred through the Internet connection, if any, or through a manual transport, using specific disks (NAS). The data sent to the central server populate the database to which the analysis algorithms will access. The elaborations are implemented by the processing chains that elaborate the raw data collected and generate final data, which can be accessed by users through the application.

The following diagram, Figure 5, specifically shows the data transfer that takes place from the acquisition on the field to the database, using ROS middleware.

Figure 5 - Collected data transferred with ROS middleware

2.7 Data Acquisition procedures

2.7.1 UGV

The data acquisition missions performed with UGV are composed by the following steps:

- Definition of purpose of mission, target trees, and path of the robot
- Deployment of robot on the field
- Initialization of the local DB and all scanning ROS nodes on the robot
- Scanning session
- Withdrawal of the robot from the field
- Exportation of capture files acquired during the scanning session
- Exportation of JSON files from the local DB with all capture metadata
- Loading of all data exported to the central server

The purpose of mission determinates target trees and sensors required during the data acquisition, and it depends on the current season and expected task to reach. According to these requirements, it was scheduled to meet the following timetable:

Date range	**Scan**	**Sensors**	**Field**
15 Nov -15 Jan	• Pre-pruning Tree Geometry reconstruction (no leaves)	• Faro Focus-S70	Field 16
15 Feb - 15 Mar	• Pre-pruning Tree Geometry reconstruction (no leaves)	• Faro Focus-S70	Field 16
20 Apr -30 Apr	• Sucker detection	• Faro Focus-S70 • Sony a5100 • MicaSense RedEdge-M	Field 18
1 May -15 May	• Tree Geometry reconstruction (with leaves)	• Faro Focus-S70	Field 16
20 May - 30 May	• Sucker detection • Pest and Disease detection	• Faro Focus-S70 • Sony a5100 • MicaSense RedEdge-M	Field 18
10 Jun - 20 Jun	• Pest and Disease detection • Water stress detection	• Faro Focus-S70 • Sony a5100 • MicaSense RedEdge-M	Field 16, Field 18
20 Jun -30 Jun	• Sucker detection • Pest and Disease detection • Water stress detection • Fruit detection	• Faro Focus-S70 • Sony a5100 • MicaSense RedEdge-M	Field 16, Field 18
10 Jun – 20 Jul	• Pest and Disease detection • Water stress detection	• Faro Focus-S70 • Sony a5100 • MicaSense RedEdge-M	Field 16, Field 18
20 Jul - 30 Jul	• Sucker detection • Pest and Disease detection • Water stress detection • Fruit detection	• Faro Focus-S70 • Sony a5100 • MicaSense RedEdge-M	Field 16, Field 18

10 Aug -20 Aug	• Pest and Disease detection • Water stress detection	• Faro Focus-S70 • Sony a5100 • MicaSense RedEdge-M	Field 16, Field 18
20 Aug -30 Aug	• Sucker detection • Pest and Disease detection • Water stress detection • Fruit detection	• Faro Focus-S70 • Sony a5100 • MicaSense RedEdge-M	Field 16, Field 18

After the deployment of robot, the global planner [3] generates the path of the robot composed by a set of waypoints to reach. The latter requires as input the target trees to elaborate the path for the current campaign, after this step the set of waypoints is saved in the collection “Waypoints” on the local DB onboard the robot.

The scanning system establishes communication with the local DB to coordinate the current waypoint to reach, and to save each time the current waypoint and all metadata generated during the scanning session. At the beginning, the scanning system generates only one time a campaign element in the collection “Campaigns”.

During the scanning session, every time the robot gets stopped on the current waypoint, the scanning system saves a position element on the collection “Positions”, then based on the field parameter takes a set of 3-6 scans with Sony a5100 and MicaSense RedEdge-M and only one scan with the Faro Focus-S70.

On each waypoint the scans can be organized in 2 different conceptual levels:

- Scan level
- Sensor level

At scan level, the scanning system saves 2*N capture elements on the collection “Captures” for the Sony a5100 and MicaSense RedEdge-M, where N depends on the size of tree (e.g. 3 for young trees and 6 for the adult ones). In addiction at this level the scanning system saves only one element on the collection “Captures” for Faro Focus-S70.

At sensor level, the scanning system saves one file element for each capture of Sony a5100, 5 file elements for each capture of Micasense RedEdge-M, and one file element for Faro Focus-S70 on the collection “Files”.

Following there is a table about the estimation size of dataset acquired by the sensors:

# scans	# trees*	Size of Sony a5100 dataset	Size of MicaSense RedEdge-M dataset	Size of Faro Focus-S70 dataset
1	0	156MB (1*6*1*26MB)	75MB (5*6*1*2.5MB)	54MB (1*1*1*54MB)
4	1	624MB (1*6*4*26MB)	300MB (5*6*4*2.5MB)	216MB (1*1*4*54MB)
6	2	936MB (1*6*6*26MB)	450MB (5*6*6*2.5MB)	324MB (1*1*6*54MB)
8	3	1248MB (1*6*8*26MB)	600MB (5*6*8*2.5MB)	432MB (1*1*8*54MB)
10	4	1560MB (1*6*10*26MB)	750MB (5*6*10*2.5MB)	540MB (1*1*10*54MB)
22	10	3432MB (1*6*22*26MB)	1650MB (5*6*22*2.5MB)	1188MB (1*1*22*54MB)

*assuming that trees are contiguous and on the same line in the adult field.

The estimation of the dataset size is based on the following equation:

- (#files)*(#captures)*(#scans)*size = dataset size

After finishing the scanning session, all sensor data is archived on a specific folder in the file system, all meta data required can be exported as JSON files from the local DB. The data transfer to the server currently is done manually.

2.7.2 UAV

The UAV platform, with 3 different sensors, provides data for the tasks related to the water stress and the pest and disease detection. In this regard, the data collection plan for the UAV starts at the beginning of May and lasts until October, when the last mission is performed.

During the campaign period, both tasks require at least one day of sensing activities per month. The number of flights performed per mission depends on the purpose of the mission. A water stress mission implies the performance of several flights during the day, starting after the sunrise and finishing one hour before the sunset. Namely, an ideal water stress mission involves 5 flights during the day:

1. an hour after sunrise
2. 09:00
3. 12:00
4. 15:00 GMT
5. an hour before sunset

where the area covered is denoted as *area 2* in Figure 6.

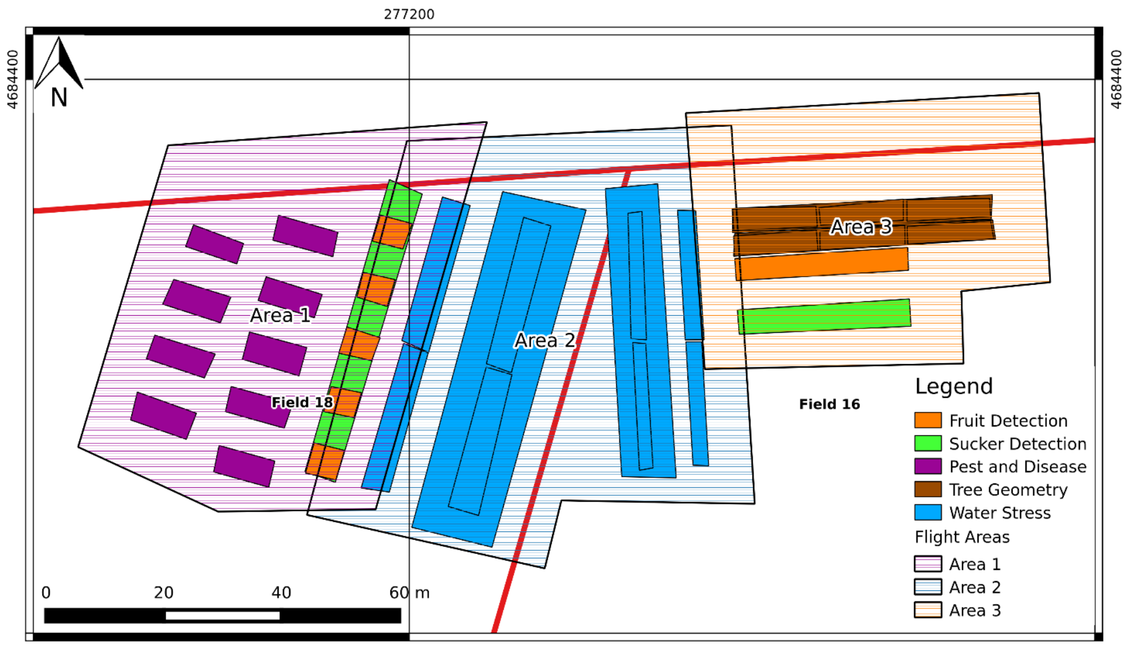


Figure 6 - Area division for the UAV remote sensing activities.

Differently, a pest and disease mission only requires a single flight covering *area 1* of Figure 6. The ideal time for this activity is at 12:00 GMT.

Remark: Modifications to this ideal plan may be carried out to comply with the legal and logistic constraints, for instance reducing the test area or the number of flights.

The 3 sensors installed on the UAV platform are:

- *Tetracam MCAW 6*, a Multispectral camera
- *Sony a5100*, an RGB camera

- *Teax ThermalCapture 2.0*, a thermal camera.

After each remote sensing flight, the data generated comes from the raw data collected by the 3 sensors and the data associated to each capture (position, velocity, orientation, etc.), required for the postprocessing activities.

The associated data is generated based on the information sent by the ROS node of the flight controller to the onboard computer. This information is locally stored in a lightweight file, called *acquisition.txt*. This file includes the information provided by the navigation sensors of the UAV regarding the pose of the UAV at each capture ('Lat','Lon','Alt','Roll','Pitch','Yaw'), the information of the relative position of the gimbal ('GimbalRoll','GimbalPitch','GimbalYaw') and the sensors triggered ('Command'). Once the mission is finished, these raw measurements are extracted and processed as metadata.

The installed cameras provide the following type of data:

- Tetracam MCAW 6: .TIFF files with a size of 15 mb/picture.
- Sony a5100: .TIFF files with a size of 12-13 mb/picture.
- Teax ThermalCapture 2.0: .TMZ files with a size of 3-4 mb/picture.

Depending on the parameters selected for a given flight, the number of images obtained may ostensibly vary. Parameters such as altitude or overlap significantly affect to the data collected by a single flight.

In this regard, subject to the data processing requirements, standard parameters consider an altitude of 30-40 meters and an image overlap superior to the 80%. This configuration provides the following amount of data per flight:

- Number of images: 100-110 images.
- Generated data:
  - I. Tetracam MCAW 6: 1.5-1.65 GB
  - II. Sony a6100: 1.2-1.3 GB
  - III. Theax ThermaCapture 2.0: 0.3-0.4 GB
- Total per flight: 3-3.35 GB.

This implies a generation 15-16.75 GB per mission in the case of water stress and 3-3.35 GB in the case of pest and disease detection, considering that both kind of flights cover areas of similar size. Note that the additional data associated to each image is not included in the computation given the great difference in size.

As a result of the large number of images obtained during each flight and the high frequency between triggers, the data captured by each sensor is locally stored on each camera during the flight. This procedure avoids possible interruption or delays on the transmission of data during the activity. On the other hand, as mentioned, the data related to the position and behavior of the UAV is stored on the onboard computer of the UAV.

Once the mission is finished, the data of each sensor is collected manually and transferred to the local server. This methodology has been proven to be the most efficient procedure given the large amount of data generated and the field conditions.

2.7.3 IoT Network

The IoT network based on the LoRa communication protocol provides data about the weather and atmospheric changes over time. It consists on the following modules:

- Meteorological station
- 9 LoRa nodes
- LoRa gateway of the network
- WebSite

The meteorological station acquires cyclically every 5 minutes several environmental variables: precipitation, wind direction and wind speed, air temperature, relative humidity, air pressure and solar radiation.
LoRa nodes represent peripheral units installed in the field for the acquisition of high-resolution soil moisture and temperature data, which collect at two depths, using capacitive SDI12 sensors. Nodes are based on a Teensy microcontroller that uses a 72 MHz Cortex-M4 processor and an RF transceiver module RFM95W that features an LoRaTM long range modem at 868 MHz.
The LoRa gateway is based on Raspberry Pi 3b+ with Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz, 1GB RAM, an Ethernet Port, a WiFi module, a LoRa RFM95W module and a high gain antenna for receiving data sent from the nodes.
The LoRa gateway collects data from the sensors and send them to the Gateway, which is responsible for loading on the WebSite and converting data into the ROS standard for storage in the primary DB server.
The data transmission of the system is based on 2 binaries: DATA PRODUCER (gw) and DATA CONSUMER (gw_wifi). The former collects all data from the LoRa network and save it on a specific folder. After this step, it also updates the queue in the file “queue.txt”. Every 20 minutes, the latter looks for new data in the queue, and if it finds new data, it consumes it. The consuming action consists in sending all new data collected to the server online and cleaning the queue. A Locking Advisory Mechanism has been implemented to manage the access to the queue.

Data type and format of string send by the nodes:

- ID,TIMESTAMP,LAT,LON,ALT,N_SENSORS,SOIL_0_WM,SOIL_0_TEMP,SOIL_0_WM_N,SOIL_0_TEMP_N,SOIL_1_WM,SOIL_1_TEMP,SOIL_1_WM_N,SOIL_1_TEMP_N,BATT_5V,BATT_12V,COUNTER,ACQUISITION_FREQ,N_SEND,DELAY,GPS_FIX,RETRY,RESET

where

- ID – ID of the node
- TIMESTAMP – timestamp
- LAT – latitude
- LON – longitude
- ALT – altitude
- N_SENSORS – number of sensors
- SOIL_0_WM – wm sensor 0 (mean)
- SOIL_0_TEMP – temperature sensor 0 (mean)
- SOIL_0_WM_N – number of samples for the mean of wm sensor 0
- SOIL_0_TEMP_N – number of samples for the mean of temperature sensor 0
- SOIL_1_WM – wm sensor 1 (mean)
- SOIL_1_TEMP – temperature sensor 1 (mean)
- SOIL_1_WM_N – number of samples for the mean of wm sensor 1

- SOIL_1_TEMP_N – number of samples for the mean of temperature sensor 1
- BATT_5V – remaining battery voltage if powered by 5V
- BATT_12V – remaining battery voltage if powered by 12V
- COUNTER – counter used to make unique the current string
- ACQUISITION_FREQ – rate of data acquisition of the sensor in minutes
- N_SEND – number of cycles before sending the string
- DELAY – delay of sending based on the ID of the node (e.g. delay of TN_02 is 1, etc.)
- GPS_FIX – checks if the node has the GPS fix
- RETRY – number of transmission attempts before receiving the ack from the LoRa gateway (max 9)
- RESET – checks if the node has been restarted

Example of a string from the node TN_01:

- TN_01,20200428000005,42.2799,12.2985,262.6,2,0.228,16.700,1,1,0.241,15.300,1,1,3.35,0.01, 19731,5,1,0,1,1,0

3 Technological Solution

In this section we illustrate the software tools that we have chosen to implement the architecture described in the previous section.

3.1 Database Management System

MongoDB is a general purpose, document-based, distributed database built for modern applications in distributed environments.

The main features of MongoDB are the following:

- It stores data in flexible, JSON-like documents, where fields can vary (from textual data to images) and data structure can be changed over time;
- It adopts a document model that maps to the objects in the application code, making data easy to work with;
- It supports ad hoc queries, indexing, and real time aggregation, thus providing powerful ways to access and analyze data;
- It is a distributed database at its core, so high availability, horizontal scaling, and geographic distribution are built-in and easy to use.

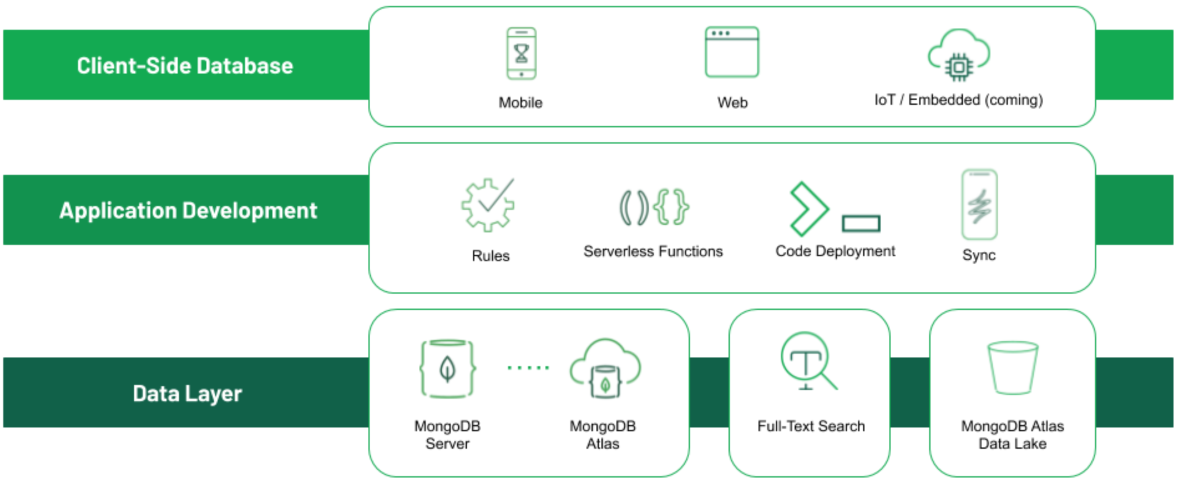


Figure 7 - The main components of MongoDB

As described in Figure 7, MongoDB relies on three main components:

- The Data Layer, a general purpose OLTP database storage system designed to serve operational and real-time analytics workloads;
- The Application Development, which helps the developers to build full-stack applications faster by providing easily configurable rules for accessing data directly from the application front-end, along with serverless functions to execute application logic.
- The Client-Side Database, which provides a support for complex queries, safe threading, responsive user interfaces, encryption, and cross-platform adoption.

In contrast to the tabular data model used by relational databases, MongoDB uses the document data model. Documents are a much more natural way to represent data: they present a single structure, with related data embedded as sub-documents and arrays, collapsing tables linked by foreign keys in a relational database.

Beyond ease-of-use, documents have many other key properties that improve developer productivity:

- Schemas can be modified at any time, allowing us to continuously integrate new application functionality, without wrestling with complex schema migrations. With Schema Validation, we have the option to enforce a schema against the data, ensuring the presence of mandatory fields, permissible values, and appropriate data types.
- Documents in a collection (analogous to a table in a relational database) can have different structures compared to other documents in the same collection.
- Data can be modeled in any way the application demands it – from rich, hierarchical documents through to flat, table-like structures, simple key-value pairs, text, geospatial data, and the nodes and edges used in graph processing.

Finally, MongoDB provides an expressive query language, secondary indexes, and aggregation pipeline that allows us to query data in different ways: from simple lookups and range queries to sophisticated processing pipelines for data analytics and transformations, through JOINs, geospatial processing, on-demand materialized views, and graph traversals.

3.2 Data Format

3.2.1 JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

As JavaScript became the default language of client-side web development, JSON began to take on a life of its own. By virtue of being both human- and machine-readable, and comparatively simple to implement support for in other languages, JSON quickly moved beyond the web page, and into software everywhere.

JSON shows up in many different cases:

- APIs
- Configuration files
- Log messages
- Database storage

JSON quickly overtook XML, is more difficult for a human to read, significantly more verbose, and less ideally suited to representing object structures used in modern programming languages.

JSON and MongoDB

MongoDB was designed from its inception to be the ultimate data platform for modern application development. JSON ubiquity made it the obvious choice for representing data structures in MongoDB innovative document data model [4].

However, there are several issues that make JSON less than ideal for usage inside of a database.

1. JSON is a text-based format, and text parsing is very slow
2. JSON readable format is far from space-efficient, another database concern
3. JSON only supports a limited number of basic data types

In order to make MongoDB JSON-first, but still high-performance and general-purpose, BSON was invented to bridge the gap: a binary representation to store data in JSON format, optimized for speed, space, and flexibility. It is not dissimilar from other interchange formats like protocol buffers, or thrift, in terms of approach.

MongoDB stores data in BSON format both internally, and over the network, but that does not mean MongoDB cannot be thought as a JSON database. Anything that can be represented in JSON can be natively stored in MongoDB and retrieved just as easily in JSON.

In PANTHEON, JSON is the main format used to model and manage data from acquisition task to the end-user GUI.

3.2.2 GEOJSON

GeoJSON is a JSON based format designed to represent the geographical features with their non-spatial attributes. This format [5] defines different JSON (JavaScript Object Notation) objects and their joining fashion. JSON format represents a collective information about the Geographical features, their spatial extents, and properties. An object of this file may indicate a geometry (Point, LineString, Polygon), a feature or collection of features. The features reflect addresses and places as point's streets, main roads and borders as line strings and countries, provinces, and land regions as polygons. Using the GeoJSON, different mobile routing and navigation applications can indicate the coverage of their services.

Following the GeoJSON specification.

Coordinate

Coordinate is the basic element of any geographic data. This is a single dimension (Longitude, latitude) representing a single number (decimal format) and sometimes record a coordinate for elevation too. Time is a dimension too, but its complexity makes it difficult to record it as coordinate. Coordinates in both JSON GeoJSON are formatted like numbers.

Position

An ordered array of coordinates represents the position. This is the smallest unit that can indicate a point on earth.

[Longitude, latitude, elevation]

Before the release of the current specification, GeoJSON allowed to record three coordinates per position but is not allowed by the new specification.

Geometry

Geometries are simple shapes (points, curves, and surfaces) in GeoJSON which consist of a type and a collection of coordinates. Point is the simplest geometry that represents a single position

{"type": "Point", "coordinates": [0, 0]}

LineStrings

At least two connected places are used to represent a line**.**

{"type": "LineString", "coordinates": [[10, 30], [10, 10]]*}*

Point and line strings are the two simplest categories of geometry. Both types of geometry don't bother many geometric rules. A point can be represented in a place anywhere, and a line can have more than one points, even if the points are self-crossing.

Polygons

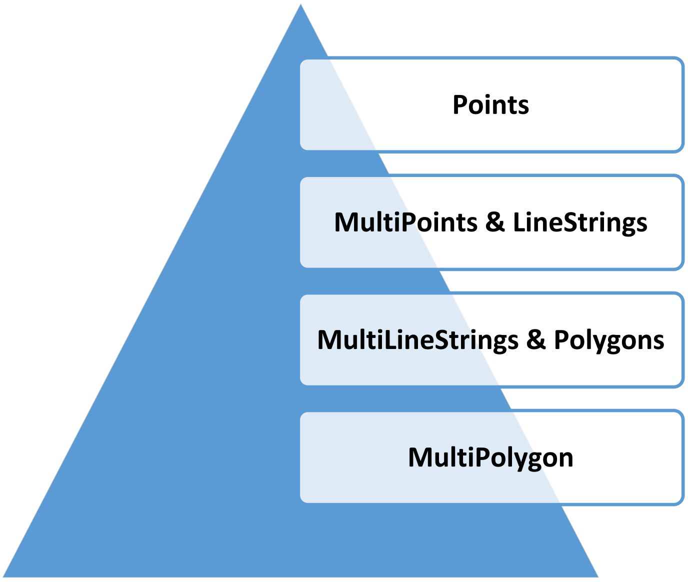
GeoJSON geometries seem significantly more complex in Polygons. Polygons have insides & outsides areas and can possess holes in that inside.

```
{
  "type": "Polygon",
  "Coordinates": [
    [
      [30, 10], [10, 10], [10, 0], [20, 40]
    ]
  ]
}
```

As compare to LineStrings, in polygons, the list of coordinates is one more level nested and can have cut-outs like donuts.

Coordinate Level

In GeoJSON format, for the coordinate property, there are four 'levels of depth'.



Features

Geometries are the central part of GeoJSON, therefore, the real-world data is more than theses simple shapes having identity and attributes. Features records the geometry as well as their properties.

```
{
  "type": "Feature",
  "geometry": {
    "type": "Point",
    "coordinates": [20, 10]
  },
  "properties": {
    "name": "fortune island"
  }
}
```

A feature property can be a type of JSON object contain single-depth key-value mappings.

FeatureCollection

At the top level of GeoJSON files, FeatureCollection is the most common thing that looks like:

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [20, 10]
      },
      "properties": {
        "name": "null island"
      }
```

```
    }
  ]
}
```


A lot of mapping and GIS software packages support GeoJSON including GeoDjango, OpenLayers, and Geoforge software. It is also compatible with PostGIS and Mapnik. The API services of Google, yahoo and Bing maps also support GeoJSON.

In PANTHEON, GeoJSON is used as data representation format for geographical attribute of all the elements deployed in the field, like trees, terrain areas, sensors, UAV and UGV.

3.2.3 GEXF

Graph file written in the GEXF (Graph Exchange XML Format) language, a language used for describing network structures; specifies the nodes and edges of the graph as well as user-defined attributes such as node weights or edge directions; can be used as an interchange format between graphing applications.

Basic topology: a GEXF file aims to represent one and only one graph.

This is a minimal file for a static graph containing 2 nodes and 1 edge between them:

```
<?xml version="1.0" encoding="UTF-8"?>
<gexf xmlns="http://www.gexf.net/1.2draft" version="1.2">
    <graph mode="static" defaultedgetype="directed">
        <nodes>
            <node id="0" label="Hello" />
            <node id="1" label="Word" />
        </nodes>
        <edges>
            <edge id="0" source="0" target="1" />
        </edges>
    </graph>
</gexf>
```

Associated data: GEXF provides a way to add data and meta-data to topology elements.

A bunch of data can be stored within attributes. The concept is the same as table data or SQL. An attribute has a title/name and a value. Attribute's name/title must be declared for the whole graph. It could be for instance "degree", "valid" or "url". Besides the name of the attribute a column also contains the type. Some meta-data can be set to the graph, like the creator's name, the date of creation, or a description.

Dynamics: GEXF provides a way to add a lifetime to nodes, edges and data.

Time in GEXF is encoded in two ways. Continuous by default, it is encoded as a double, but may also be an international standard date (yyyy-mm-dd). Discrete, it is an integer. Both network topology and data have a lifetime. The whole graph, each node, each edge and their respective data values may have time limits, beginning with an XML-attribute start and ending with end. Attributes declared as dynamic can exist during a time scope.

Hierarchy: clustering can be stored inside a hierarchy of nodes.

There are 2 ways to write a hierarchy in GEXF, depending on how data is processed:

- Sequential-safe Reading: nodes can simply host other nodes and so on.
- Random Writing: each node refers to a parent node id with the XML-attribute pid.

The first style is preferred when the structure written is previously ordered. Sequential reading of this kind of GEXF is safe because no node reference is used. But in the case the used program cannot provide it, the second way allows writing (and then reading) nodes randomly, but linear reading can be less straightforward.

Visualization: this module is an extension using a different namespace. It provides attributes for coloring in RGB, positioning inside a 3D space, setting size, color, position and shape of nodes and edges.

In PANTHEON, the GEXF file format is used to model the tree geometry and visualize the 3D representation on the end-user application.

3.3 Data Elaboration

3.3.1 ROS Communication

Here we will explore some of the main components of ROS [6]. One of the primary purposes of ROS is to facilitate communication between the ROS nodes. These nodes represent the executable code. The code can reside entirely on one computer, or nodes can be distributed between computers or between computers and robots. The advantage of this distributed structure is that each node can control one aspect of a system.

For example, one node can capture the images from a camera and send the images to another node for processing. After processing the image, the second node can send a control signal to a third node for controlling a robotic manipulator in response to the camera view.

The main mechanism used by ROS nodes to communicate is by sending and receiving messages. The messages are organized into specific categories called topics. Nodes may publish messages on a topic or subscribe to a topic to receive information.

ROS Nodes

Basically, nodes are processes that perform some computation or task. The nodes themselves are really software processes but with the capability to register with the ROS Master node and communicate with other nodes in the system. The ROS design idea is that each node is independent and interacts with other nodes using the ROS communication capability.

One of the strengths of ROS is that a task, such as controlling a wheeled mobile robot, can be separated into a series of simpler tasks. The tasks can include the perception of the environment using a camera or laser scanner, map making, planning a route, monitoring the battery level of the robot's battery, and controlling the motors driving the wheels of the robot. Each of these actions might consist of a ROS node or a series of nodes to accomplish the specific tasks.

A node can independently execute code to perform its task but can also communicate with other nodes by sending or receiving messages. The messages can consist of data, commands, or other information necessary for the application.

ROS Topics

Some nodes provide information for other nodes, as a camera feed would do, for example. Such a node is said to publish information that can be received by other nodes. The information in ROS is called a topic. A topic defines the types of messages that will be sent concerning that topic.

The nodes that transmit data publish the topic name and the type of message to be sent. The actual data is published by the node. A node can subscribe to a topic and transmitted messages on that topic are received by the node subscribing.

Continuing with the camera example, the camera node can publish the image on the “camera/image_raw” topic. Image data from the “camera/image_raw” topic can be used by a node that shows the image on the computer screen. The node that receives the information is said to subscribe to the topic being published, in this case “camera/image_raw”.

In some cases, a node can both publish and subscribe to one or more topics.

ROS Messages

ROS messages are defined by the type of message and the data format. The ROS package named “std_msgs”, for example, has messages of type “String” which consist of a string of characters. Other message packages for ROS have messages used for robot navigation or robotic sensors.

In PANTHEON, the ROS communication features are used to exchange data between the data acquisition sensors and the Farm server.

3.3.2 Acquisition Data Import

During the project, many data acquisition sessions are performed on the field. This activity is carried out for testing purposes and for collecting real data of the project.

In addition, this allows to have a real basic dataset on which develop all the components involved in data management. In particular, the data acquired with the UAV and UGV platforms are stored inside a SD Cards installed on board. Then, the data is transferred to the local server's file system (or alternatively to the central server file system).

Regarding file system management, a specific folder structure has been defined to catalogue all the collected data, grouped by dates on which the acquisition missions were performed, by platform type, by sensor type, etc. In Figure 8 there is a sample screenshot of the structure used.

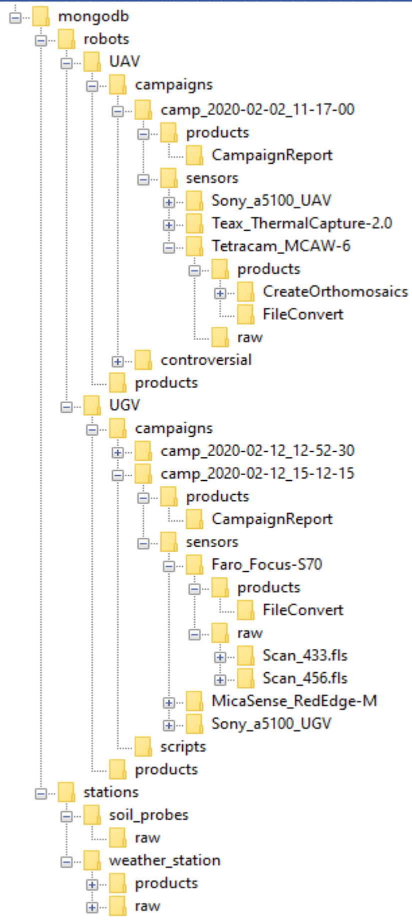


Figure 8 - Screenshot of the FTP folder structure

A dedicated Python script has been developed for the last step of data transferring to the system database (MongoDB). The script allows, automatically, to parse the metadata and the acquisitions from the file system and import the data into the database collections. The import algorithm can be executed whenever new acquired data is inserted into the file system. The flow is represented in following Figure 9.

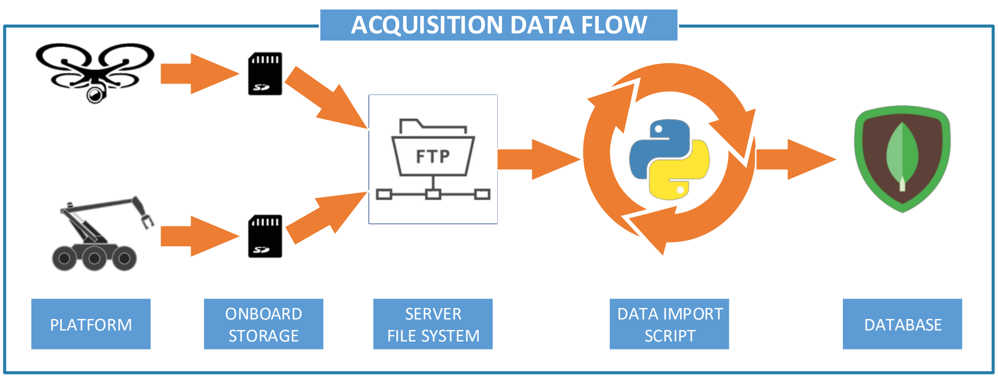


Figure 9 - Acquisition data flow schema

3.3.3 MEAN

MEAN (MongoDB, Express.js, AngularJS (or Angular), and Node.js) is a free and open-source JavaScript software stack for building dynamic web sites and web applications [7].

Because all components of the MEAN stack support programs that are written in JavaScript, MEAN applications can be written in one language for both server-side and client-side execution environments.

Though often compared directly to other popular web development stacks such as the LAMP stack, the components of the MEAN stack are higher-level including a web application presentation layer and not including an operating system layer.

Main components of the stack (composing acronyms) are the following:

- MongoDB: is a NoSQL database program that uses JSON-like BSON (binary JSON) documents with schema. The role of the database in the MEAN stack is very commonly filled by MongoDB because its use of JSON-like documents for interacting with data as opposed to the row/column model allows it to integrate well with the other (JavaScript-based) components of the stack.
- Express.js: (also referred to as Express) is a modular web application framework package for Node.js. Whilst Express can act as an internet-facing web server, even supporting SSL/TLS out of the box, it is often used in conjunction with a reverse proxy such as NGINX or Apache for performance reasons.
- Angular and alternatives: typically data is fetched using Ajax techniques and rendered in the browser on the client-side by a client-side application framework, however as the stack is commonly entirely JavaScript-based, in some implementations of the stack, server-side rendering where the rendering of the initial page can be offloaded to a server is used so that the initial data can be prefetched before it is loaded in the user's browser. Angular (MEAN), React (MERN) and Vue.js (MEVN) are the most popular amongst other web application frameworks used in the stack and a number of variations on the traditional MEAN stack are available by replacing the web application framework with similar frameworks, or even by removing this component of the stack altogether (MEN).
- Node.js: is the application runtime that the MEAN stack runs on. The use of Node.js which is said to represent a "JavaScript Everywhere" paradigm is integral to the MEAN stack which relies on that concept.

The concept of the MEAN stack technology is to allow developers in developing more responsive apps with a single language at all the platforms.

MongoDB database imparts a splendid similarity to different databases; however, it is composition less which makes additions and deletions very simple. This factor and element of the MEAN stack development tool completely avoid complications and terminations while working with a big data. It is truly a complex task to deal with data isolated into tables and columns in SQL databases. This capacity similarly makes MEAN based development synchronized with cloud and cloud-based applications. Therefore, the cloud-based apps can be easily developed and presented to the cloud network.

In PANTHEON, MEAN stack is used in the implementation of the end-user application. That application works directly with the system database (MongoDB). In this scenario, the back-end component exchange data with MongoDB in JSON format and expose the data management functionalities to the front-end component through REST APIs.

3.3.4 Data Processing

The data associated with remote sensing tasks are typically processed via Python [8]. For each task, one or many processing chains in form of python scripts are created. In general, such a script connects to MongoDB, queries the data it needs, processes or analyses the data and writes back the results to the database. The core Python libraries used to process the data are OpenCV [8] [9] [10] [11] and scikit-learn [12]. A processing chain might also pass tasks to command line scripts, e.g. to convert files, if required.

- OpenCV: OpenCV is used as a tool to perform image processing in Python. In particular, the images of the multispectral UGV cameras are aligned with the 3D (three-dimensional) laser scans. To enable this, OpenCV functions are used to calibrate the intrinsic and extrinsic sensor orientation.
- Metashape: Agisoft Metashape is used to generate orthomosaics using multispectral UAV images photogrammetricly processing of digital images and generates 3D spatial data. Its Python interface [TODO citation AgisoftLLC_2020b] allows for a smooth integration into PANTHEONs architecture.
- PyMongo: PyMongo represents a Python driver for MongoDB. It is used to query and write data for tasks associated with data processing.
- Pyoints: Pyoints is used as a tool to deal with various representations of 2D and 3D geodata in Python. In particular, the laser scan alignment and various functions to deal with the UGV data is implemented using Pyoints.
- scikit-learn: The Python module scikit-learn is used in particular for machine-learning based tasks. In particular, classification, clustering and regression is performed using this module.

To make a processing chain available to other applications, each script is linked to a “Chain” object stored in the “chains” collection of MongoDB. This allows for a live triggering of the chain by other applications. In particular, the configuration of the chain is passed to the script. This concept enables dynamical adding of processing chains to PANTHEONs architecture, without having to modify e.g. the front-end.

4 Data Model

4.1 Introduction

The Pantheon project database includes various groups of tables used to store large amounts of data for homogeneous purposes.

It supports both the data acquisition processes, performed by the various platforms (UAV, UGV, weather station, human and soil sensor), and the data processing processes that determine the data on the state of the trees and the agronomic activities to be performed.

They also support the functionality of the web application. Following, in Figure 10, there is the synthetic version of the scheme that represents only the collections and their references, for the complete schema details, with all the attributes, see the individual sections or appendix section 5.1.

The colour of the collection represents the group the collection is part of.

In the next sections, grouped by homogeneous function, the individual collections will be described in detail.

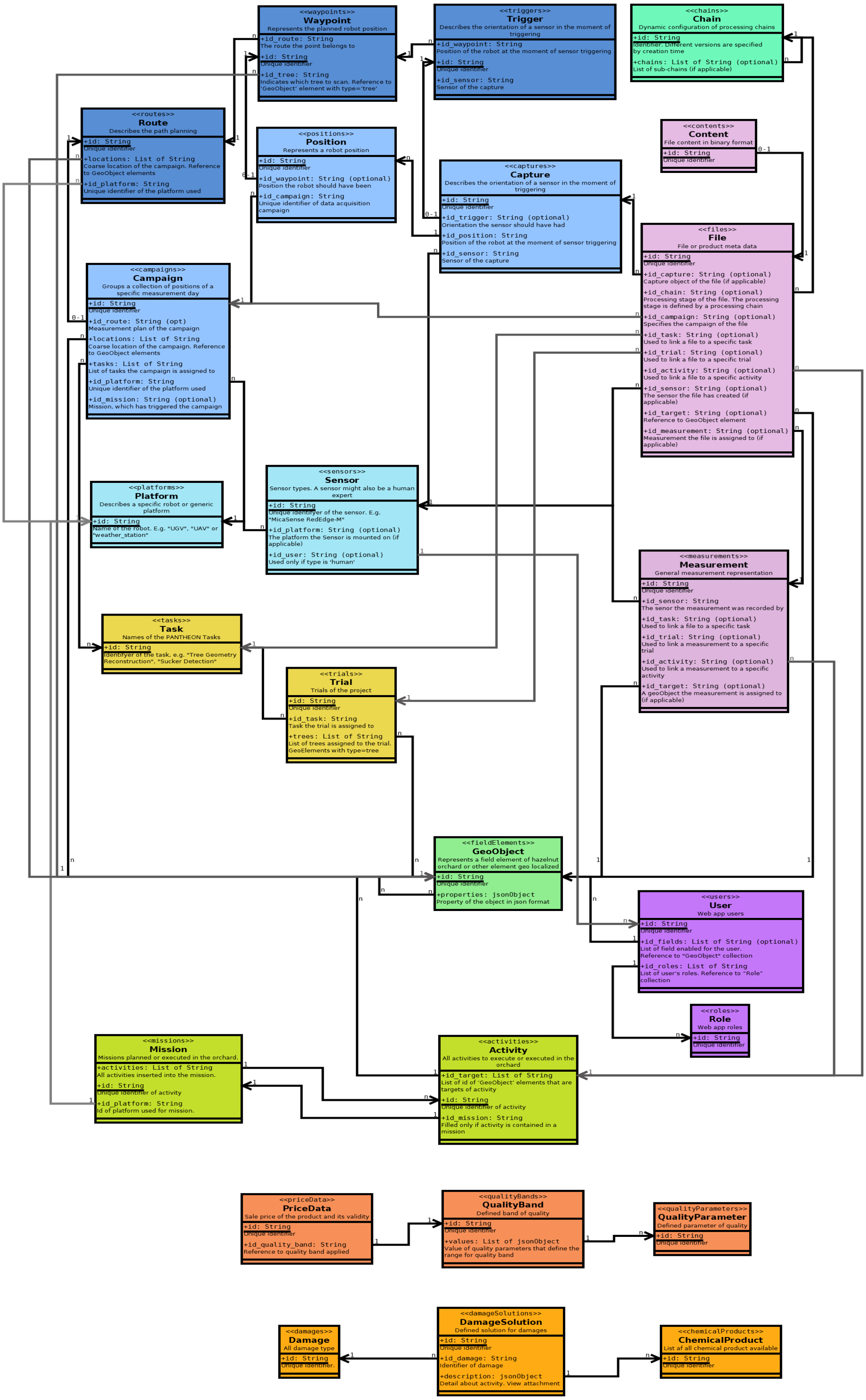


Figure 10 - Synthetic full data model schema

4.2 Configuration

These collections, shown in Figure 11, need to store the structure of the field and the geo-located elements that can be used as targets of operations and activities.

In addition, there are platforms and sensors used in the system to perform data acquisition.

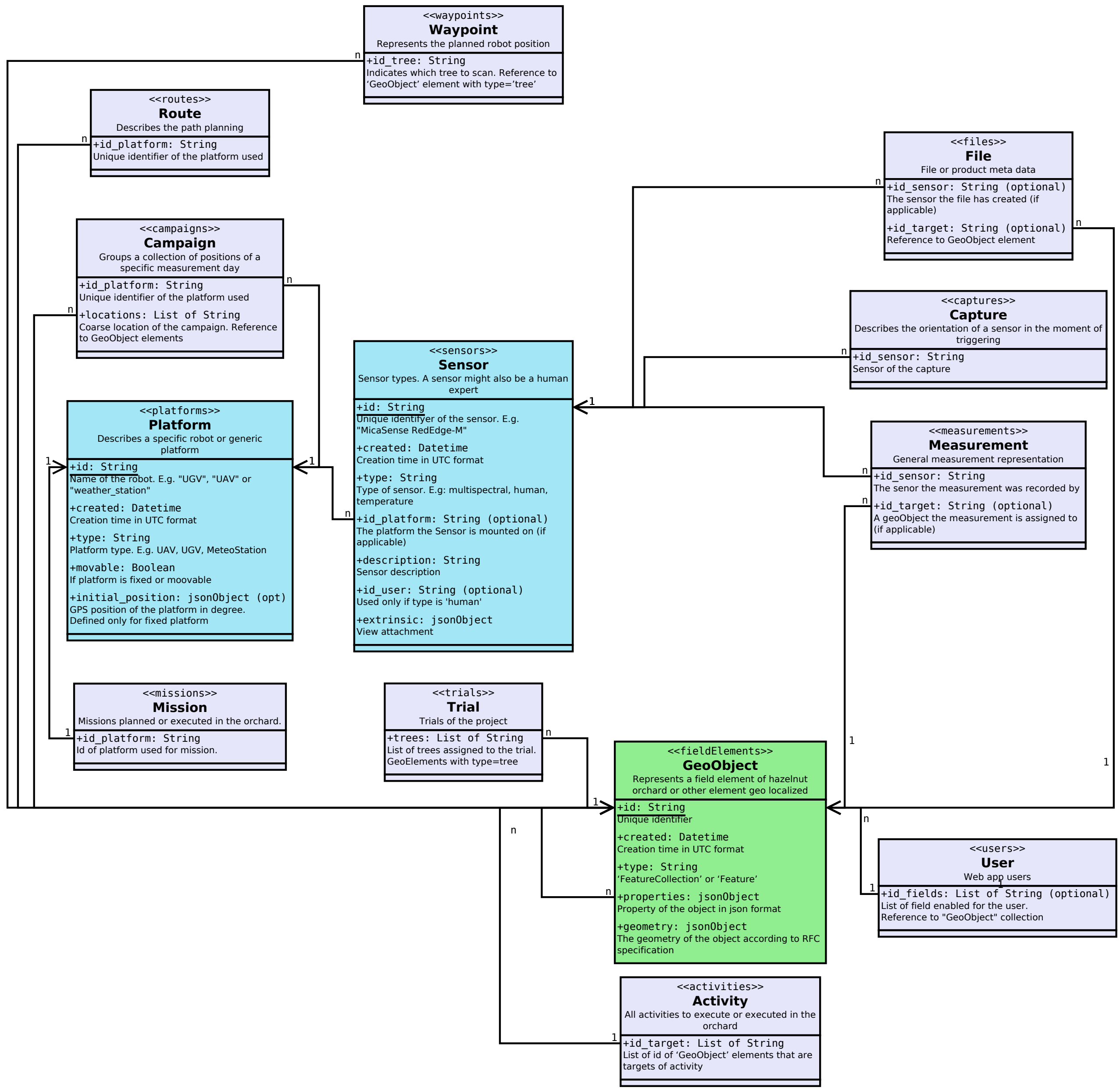


Figure 11 - Configuration group elements of the data model schema

4.2.1 GeoObject

This collection defines all the geo-located elements present in the hazelnut field, using the standard GeoJSON format.

The coordinates of the element's point or perimeter are specified in the 'geometry' field.

The attributes of the object are stored in the properties field, at least the name and type of the element must be specified, and it is also possible to specify any child elements, if the element represents a group of other elements (for example a row of trees or a plot).

It is also possible to specify further details in the properties.info field which vary according to the type of element being described.

4.2.1.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Unique identifier	Field_16
created	No	Datetime	Creation time in UTC format	2019-02-15_11-20-35.0
type	No	String	'FeatureCollection' or 'Feature'	Feature
properties	No	jsonObject	List of properties of the object	View GeoObject.properties example
geometry	No	jsonObject	The geometry of the object according to RFC specification	View GeoObject.geometry example

4.2.1.2 JSON Format – GeoObject.geometry

Key	Optional	Data type	Description	example
type	No	String	Type of geometry. 'Point' or 'Polygon' or 'MultiPolygon'	Polygon
coordinates	No	Position or List of Position	A position or a list of position that represents the geometry of the object	[[[12.297558, 42.279835], [12.297690, 42.279853], [12.298356, 42.279884], [12.299830, 42.279973],]]

4.2.1.3 JSON Format – GeoObject.properties

Key	Optional	Data type	Description	example
name	No	String	Name of the object	Field 16
type	No	Enum of String	Object type Enum value: • farm • field • plot • row	farm

			• tree	
info	Yes	jsonObject	A list of properties of the object, according to format “GeoObject.properties.info”	View ‘GeoObject.propeties.info’ example
children	Yes	List of String	List of id of ‘GeoObject’ elements contained in the object	["RYNI", "RYI", "RYTA", "RYTB", "RYTC", "RYF", "RYS"]

4.2.1.4 JSON Format – GeoObject.properties.info

GeoObject.properties.type = “farm”

Key	Optional	Data type	Description	example
total_surface	Yes	Float	SAT	1500
cultivated_surface	Yes	Float	SAU	1250
field_number	Yes	Integer	Number of fields in farm	2
plot_number	Yes	Integer	Total number of plots into farm	13
management_system	No	Enum	Enum value: • organic • conventional • integrated	integrated

GeoObject.properties.type = “field”

Key	Optional	Data type	Description	example
total_surface	Yes	Float	SAT	950
cultivated_surface	Yes	Float	SAU	850
plot_number	Yes	Integer	Total number of plots into field	6
irrigation_type	Yes	String	Type of irrigation	subirrigated
irrigation_flow_rate	Yes	Float	Irrigation flow rate in lt/h	10

GeoObject.properties.type = “plot” | “row”

Key	Optional	Data type	Description	example
total_surface	Yes	Float	SAT	5
cultivated_surface	Yes	Float	SAU	5
fruit_variety	Yes	List of FruitVariety	Embedded documents of ‘FruitVariety’ object	{ ‘name’:’Tonda romana gentile’ }
planting_year	Yes	Integer	Planting year of the trees	2010
tree_number	Yes	Integer	Number of trees in the element	10
planting_layout	Yes	String	Planting layout in mt x mt	5x5

plant_density	Yes	Integer	Plant density in nr pl/ha	3
tree_shape	Yes	String	The shape of trees	multibranches
irrigation_type	Yes	String	Type of irrigation	subirrigated
irrigation_flow_rate	Yes	Float	Irrigation flow rate in lt/h	10
soil_type	Yes	String	Type of soil	clay

GeoObject.properties.type = “tree”

Key	Optional	Data type	Description	example
fruit_variety	Yes	List of FruitVariety	Embedded documents of ‘FruitVariety’ object	{ ‘name’:’Tonda romana gentile’ }
planting_year	Yes	Integer	Planting year of the trees	2010
tree_shape	Yes	String	The shape of trees	multibranches
irrigation_type	Yes	String	Type of irrigation	subirrigated
irrigation_flow_rate	Yes	Float	Irrigation flow rate in lt/h	10
soil_type	Yes	String	Type of soil	clay

4.2.2 Platform

This table store the data of platforms used for data capture.

The platforms can be either fixed, like the weather station, or mobile like the ground robot and drone. Each of them has installed one or more sensors that will perform the measurements.

For fixed platforms, a position is defined using georeferenced coordinates, while for mobile platforms, this data is stored when data acquisition campaigns are performed.

4.2.2.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Unique identifier. Name of the robot	UAV
created	No	Datetime	Creation time in UTC format	2020-01-10_12-21-50.0
type	No	String	Platform type	UAV
movable	No	Boolean	If platform is fixed or movable	true
initial_position	Yes	jsonObject	GPS position of the platform in degree. Defined only for fixed platform	N.A.
extrinsic	Yes	jsonObject	Specifies the initial orientation of the platform.	null

4.2.2.2 JSON Format – Platform. initial_position

Platform.movable = false

Key	Optional	Data type	Description	example
latitude	No	Float	Latitude of GPS position of the platform in degree	42.28013093333333
long	No	Float	Longitude of GPS position of the platform in degree	12.297804066666666

altitude	No	Float	Altitude of GPS position of the platform in meters	282.1049995422363

4.2.3 Sensor

This collection stores the data of all the devices through which it is possible to perform data acquisition.

For automatic detections, sensors such as thermal or multispectral cameras, weather stations or soil moisture meters can be used.

Manual surveys, on the other hand, can be performed by expert operators and stored though the user application, in which case the identification of the user who performed the operation will be stored.

For the sensors equipped with it, the initial orientation position in relation to the kinematics is also stored.

4.2.3.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Unique identifier of the sensor	MicaSense RedEdge-M
created	No	Datetime	Creation time in UTC format	2019-02-15_11-20-35.0
type	No	Enum of String	Sensor type Enum value: • soil_probe • laser_scanner • weather_station • camera • image_sensor	camera
id_platform	Yes	String	The platform the Sensor is mounted on	UGV
description	No	String	Sensor description	MicaSense RedEdge-M
id_user	Yes	String	Used only if type is 'human'	null
extrinsic	Yes	jsonObject	Specifies the initial orientation of the intrinsic sensor orientation in relation to the kinematics.	null

4.2.3.2 JSON Format – Sensor.type

Sensor.type = "camera"

Key	Optional	Data type	Description	example
label	No	String	Label of the Camera when displayed.	Sony a5100 UAV
sensors	No	List of String	List of sub-sensors. The camera is seen as a collection of sensors.	["Sony_a5100_UAV_0", "Sony_a5100_UAV_1", "Sony_a5100_UAV_2"]
master	No	Integer	Specifies the master sensor.	0
extrinsics	No	List of List	Specifies the relative position and orientation of the sub-sensors in	[[[1, 0, 0, 0],

			relation to the master sensor.	[0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]], null, null]
relative_exposure	Yes	List of Float	Relative exposure times in relation to the master sensor.	10

Sensor.type = "image_sensor"

Key	Optional	Data type	Description	example
label	No	String	Label of the Camera when displayed.	Sony a5100 UAV red
device	No	String	Identifier of the physical device the imaging sensor is mounted to.	Sony_a5100_UAV
height	No	Integer	Height of the sensors image in pixels.	4000
width	No	Integer	Width of the sensors image in pixels.	6000
sensor_height	No	Float	Height of the sensor plate in meters.	0.0156
sensor_width	No	Float	Width of the sensor plate in meters.	0.0235
focal_length	No	Float	Focal length of the sensor according to the manufacturer.	0.035

intrinsics	Yes	JsonObject	Intrinsic parameters of the sensor.	{
				"c_x": 3001.2,
				"c_y": 1999.7,
				"f_x": 5000.1,
				"f_y": 5000.2,
				"s": 0.01
				}

wavelength	Yes	Float	Wavelength of maximum sensitivity in Nanometers.	610
fwhm	Yes	Float	Full width at half maximum of the sensor's sensitivity	40
spectral_sensitivity	Yes	JsonObject	Spectral sensitivity for specific wavelengths.	{ "wavelength": [500, 501, …], "transmission": [

				0.01, 0.02, …] }

distortion	Yes	JsonObject	Lens distortion parameters of the sensor. An attribute "type" specifies which distortion model has been used.	{
				"type": "opencv",
				"k1": -0.15,
				"k2:" 0.201,
				"p1": -0.001,
				"p2": 0.0,
				"p3": -0.555
				}

vignetting	Yes	JsonObject	Vignetting image or coefficients designed to create the vignetting image.	null

4.2.3.3 JSON Format – extrinsic

This information describes the initial orientation of the sensor respect to the camera, specifying the sensor rotation matrix and possibly the spatial projection system.

Key	Optional	Data type	Description	example
m	No	List of List	4x4 roto-translation matrix of the sensor.	[[0, -1, 0, 0], [-1, 0, 0, 0], [0, 0, -1, 0], [0, 0, 0, 1]]
proj4	Yes	String	Spatial projection system. String might be derived from an EPSG code.	"+proj=utm +zone=33 +ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +units=m +no_defs"

4.3 Acquisition

All the information needed to acquire tree data and their status is stored in this collection group.

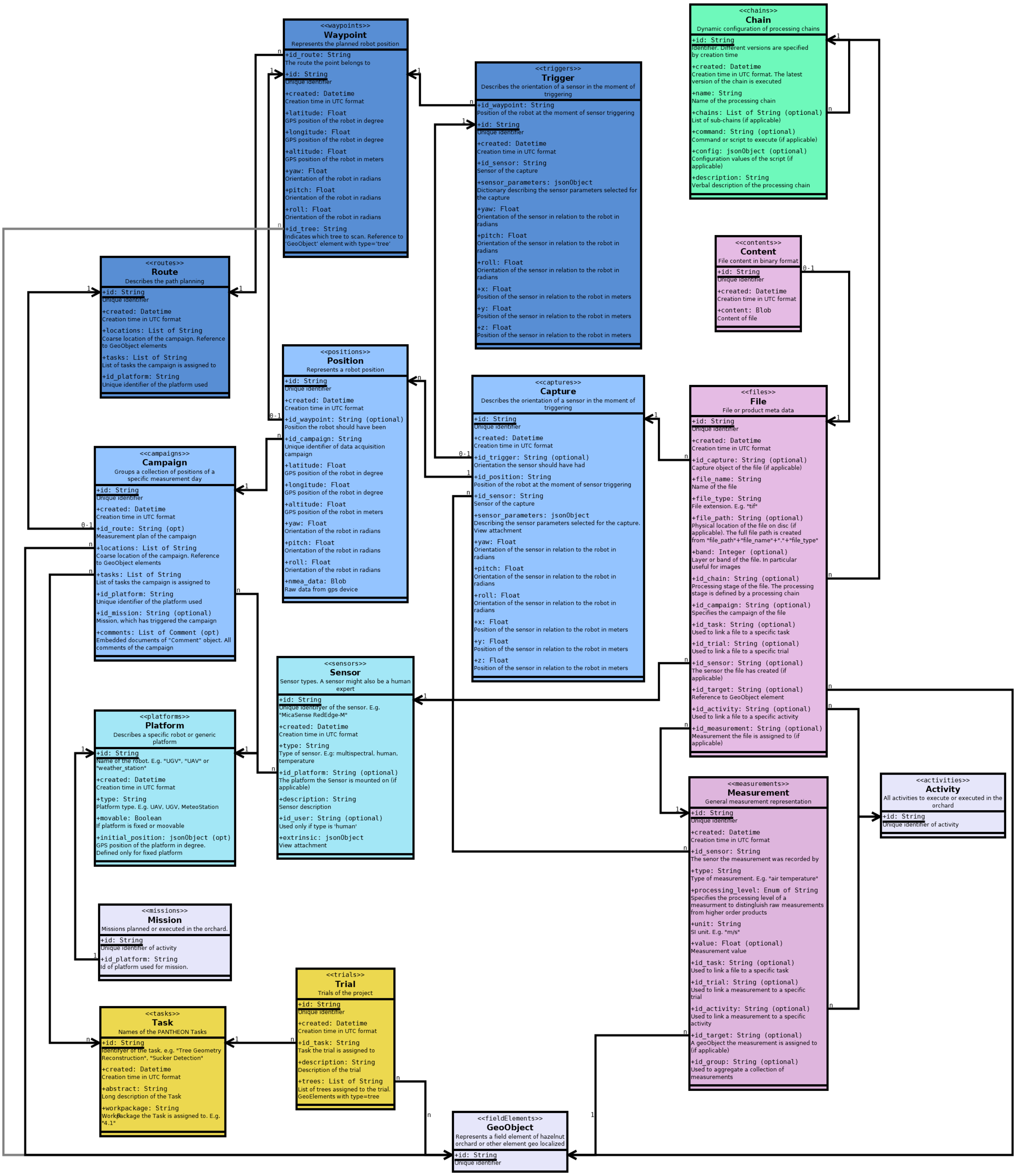


Figure 12 - Acquisition group elements of the data model schema

Some collections, as shown in Figure 12, contain the navigation plans of the devices and the definition of the sensor parameters, others contain the raw data acquired. In addition, there are collections designed to store the data processed by the processing processes and the media acquired by the users of the web apps. Real-time measurements are also stored in this group.

4.3.1 Task

This collection stores the list of tasks of Pantheon project, in order to associate them as a goal of the campaigns and trials.

4.3.1.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Identifier of the task	Tree_Geometry_Reconstruction
created	No	Datetime	Creation time in UTC format	2019-02-15_11-20-35.0
abstract	No	String	Long description of the Task	Tree Geometry Reconstruction
workpackage	No	String	Work package the Task is assigned to	4.1

4.3.2 Trial

All the tests set for data acquisition and association with the target test trees are stored in this collection.

4.3.2.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Unique identifier	pruning_Yo
created	No	Datetime	Creation time in UTC format	2019-02-15_11-20-35.0
id_task	No	String	Task the trial is assigned to	Pruning_Management_Protocol
description	No	String	Description of the trial	Pruning variant A of young trees.
trees	No	List of String	List of trees assigned to the trial. GeoElements with type=tree	["Yo_S1", "Yo_S2"]

4.3.3 Campaign

The Campaign collection contains information from the various acquisition campaigns.

The information stored is used to identify the day on which this campaign was carried out, the task and the target of the acquisition. Furthermore, the platform which must carry out these measurements and the positions it must take (id_platform and id_route) can be specified.

The campaign can be connected, via the id_mission field, to a planned mission using user application.

4.3.3.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Unique identifier	camp_2020-01-10_12-21-47
created	No	Datetime	Creation time in UTC format	2020-01-10_12-21-47.0
id_route	Yes	String	Measurement plan of the campaign	null

locations	No	List of String	Coarse location of the campaign. Reference to 'GeoObject' elements	[" ZYT"]
tasks	No	List of String	List of tasks the campaign is assigned to	["Tree_Geometry_Reconstruction", "Suckers_Detection"]
id_platform	No	String	Unique identifier of the platform used	UGV
id_mission	Yes	String	Mission, which has triggered the campaign	M_2020_01_13_1035
comments	Yes	List of Comment	Embedded documents of "Comment" object. All comments of the campaign	View Comment example

4.3.4 Route

In this collection is stored the navigation plan of the platform that will acquire data in a campaign.

4.3.4.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Unique identifier	route_tree_geometry_field16
created	No	Datetime	Creation time in UTC format	2020-01-10_12-21-47.0
locations	No	List of String	Coarse location of the campaign. Reference to 'GeoObject' elements	["Field_16"]
tasks	No	List of String	List of tasks the campaign is assigned to. Reference to 'Task' elements	["Tree_Geometry_Reconstruction", "Suckers_Detection"]
id_platform	No	String	Unique identifier of the platform used. Reference to 'Platform' element	UGV

4.3.5 Waypoint

In this collection are stored the GPS positions that the data acquisition platform should assume during the measurement campaign.

For each waypoint is also specified the yaw-pitch-roll rotation that the robot must have and possibly the target tree of the acquisition.

4.3.5.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Unique identifier	waypoint_1
created	No	Datetime	Creation time in UTC format	2020-01-10_12-21-47.0
id_route	No	String	The route the point belongs to. Reference to 'Route' element	route_tree_geometry_field16
latitude	No	Float	GPS position of the robot in degree	42.2801309

longitude	No	Float	GPS position of the robot in degree	12.29780407
altitude	No	Float	GPS position of the robot in meters	282.0
yaw	No	Float	Orientation of the robot in radians	0.1
pitch	No	Float	Orientation of the robot in radians	0.0
roll	No	Float	Orientation of the robot in radians	-0.02
id_tree	Yes	String	Indicates which tree to scan. Reference to 'GeoObject' element with type='tree'	"Yo_S1"

4.3.6 Position

The real positions in which the robot acquired the data are stored in this collection, each real position can be associated with a predetermined position of the navigation plan through the id_waypoint field.

In addition, the raw data sent by the robot's GPS device is stored in the nmea_data field.

4.3.6.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Unique identifier	pos_2020-01-10_12-21-47
created	No	Datetime	Creation time in UTC format	2020-01-10_12-21-47.0
id_waypoint	Yes	String	Position the robot should have been	Ad_S1-XX-Ad_S2-XX-South
id_campaign	No	String	Unique identifier of data acquisition campaign	camp_2020-01-10_12-21-47
latitude	Yes	Float	GPS position of the robot in degree	42.28013093333333
longitude	Yes	Float	GPS position of the robot in degree	12.297804066666666
altitude	Yes	Float	GPS position of the robot in meters	282.1049995422363
yaw	Yes	Float	Orientation of the sensor in relation to the robot in radians	null
pitch	Yes	Float	Orientation of the sensor in relation to the robot in radians	null
roll	Yes	Float	Orientation of the sensor in relation to the robot in radians	null
nmea_data	No	Blob	Raw data from gps device	$GPGGA,123519,4807.038, N,01131.000, E,1,08,0.9,545.4,M,46.9,M, ,*47

4.3.7 Capture

All the information concerning the single capture event is stored in this collection.

With each capture, the relative position is memorized, which sensor has been used, its orientation and its positioning with respect to the position of the robot.

Each sensor model has different acquisition parameters, the data stored for each model used in the project are listed below.

4.3.7.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Unique identifier	rgb_2020-01-10_12-21-50

created	No	Datetime	Creation time in UTC format	2020-01-10_12-21-50.0
id_trigger	Yes	String	Orientation the sensor should have had	N.A.
id_position	No	String	Position of the robot at the moment of sensor triggering	pos_2020-01-10_12-21-47
id_sensor	No	String	Sensor of the capture	Sony_a5100_UGV
sensor_parameters	No	jsonObject	Describing the sensor parameters selected for the capture	View example in 'Capture.sensor_parameters'
yaw	Yes	Float	Orientation of the sensor in relation to the robot position in radians	-1.1934650215387383
pitch	Yes	Float	Orientation of the sensor in relation to the robot position in radians	0.2635410250749983
roll	Yes	Float	Orientation of the sensor in relation to the robot position in radians	-0.005423234509639068
x	Yes	Float	Position of the sensor in relation to the robot position in meters	-1.51344653701792
y	Yes	Float	Position of the sensor in relation to the robot position in meters	0.038482575267721586
z	Yes	Float	Position of the sensor in relation to the robot position in meters	0.5680666632859868

4.3.7.2 JSON Format – Capture.sensor_parameters

Capture.id_sensor = "Sony_a5100_UGV"

Key	Optional	Data type	Description	example
aspectratio	Yes	String	Aspect ratio	3:2
capturemode	Yes	String	Capture mode	Single Shot
exposurecompensation	Yes	String	Exposure compensation	0
exposuremetermode	Yes	String	Exposure meter mode	Average
expprogram	Yes	String	Exposure program	Intelligent Auto
f_number	Yes	String	Frame number	16.0
focusmode	Yes	String	Focus mode	Automatic
imagequality	Yes	String	Image quality	RAW
shutterspeed	Yes	String	Shutter speed	1/125
imagesize	Yes	String	Image size	Large
iso	Yes	String	Iso	Auto ISO
whitebalance	Yes	String	White balance	Automatic

Capture.id_sensor = "MicaSense_RedEdge-M"

Key	Optional	Data type	Description	example
exposureAuto	Yes	Bolean	Automatic exposure	true

Capture.id_sensor = “Faro_Focus-S70”

Key	Optional	Data type	Description	example
resolution	Yes	String	Resolution	1/8
quality	Yes	String	Quality	2
distance	Yes	String	Distance	near

4.3.8 Trigger

In this collection are stored all the information that has been planned concerning the single capture event.

It contains data like that of the Capture collection but represents the planning of the acquisition events.

4.3.8.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Unique identifier	trigger_1
created	No	Datetime	Creation time in UTC format	2020-01-10_12-21-50.0
id_waypoint	No		Position of the robot at the moment of sensor triggering	waypoint_1
id_sensor	No	String	Sensor of the capture	Sony_a5100_UGV
sensor_parameters	No	jsonObject	Sensor parameters selected for the capture according to the sensor type	{}
yaw	No	Float	Orientation of the sensor in relation to the robot in radians.	0.0
pitch	No	Float	Orientation of the sensor in relation to the robot in radians.	0.2
roll	No	Float	Orientation of the sensor in relation to the robot in radians.	0.0
x	No	Float	Position of the sensor in relation to the the robot in meters	0.2
y	No	Float	Position of the sensor in relation to the the robot in meters	0.0
z	No	Float	Position of the sensor in relation to the the robot in meters	0.0

4.3.8.2 JSON Format – Trigger.sensor_parameters = Capture.sensor_parameters

This format is the same of the “Capture” element, see paragraph 4.3.7.2.

4.3.9 File

This collection stores the metadata of the files related to the project such as the images of the captures, the intermediate files of the elaboration processes, the media acquired through the application user interface and other attachments.

The files can be stored on the file system, in this case path and filename will be specified, or stored in binary format directly within the database in the 'Content' collection, in this case they will be identified by the same unique identifier. In addition, when possible, the acquisition target will be specified (be it a single tree or a larger set).

If the files derive from a capture, the sensor, the identification of the capture and the acquisition campaign and possibly the related task and trial will be specified.

For files derived from elaboration processes, it will be specified through the step identifier (id_chain) which phase of the process they are related to.

For the files acquired via the web interface, the activity to which they are connected (id_activity) and possibly the related measurement will be specified.

4.3.9.1 Field description

Key	**Optional**	**Data type**	**Description**	**example**
id	No	String	Unique identifier	faro_2020-02-12_12-52-36
created	No	Datetime	Creation time in UTC format	2020-02-12_12-52-36.0
id_capture	Yes	String	Capture object of the file (if applicable)	faro_2020-02-12_12-52-36
file_name	No	String	Name of the file	Scan_408
file_type	No	String	File extension	fls
file_path	Yes	String	Physical location of the file on disc (if applicable). The full file path is created by file_path" + “/” + "file_name" (+ “_” + “band”) + "." + "file_type"	./Faro_Focus-S70/raw
band	Yes	Integer	Layer or band of the file. In particular useful for images	0
id_chain	Yes	String	Processing stage of the file. The processing stage is defined by the processing chain the file has been created by	file_conversion
id_campaign	Yes	String	Specifies the campaign of the file	camp_2020-02-12_12-52-30
id_task	Yes	String	Used to link a file to a specific task	Water_Stress_Measurement
id_trial	Yes	String	Used to link a file to a specific trial	pruning_Yo
id_activity	Yes	String	Used to link a file to a specific activity	105
id_sensor	Yes	String	The sensor the file has created (if applicable)	Faro_Focus-S70
id_target	Yes	String	Reference to GeoObject element	Yo_S1
id_measurement	Yes	String	Measurement the file is assigned to (if applicable)	measurement_1

4.3.10 Content

The contents of the files are stored in binary format in this collection.

The unique identifier stored in this table is the same used to store the related metadata in the File collection.

4.3.10.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Unique identifier, the same for the 'File' collection	faro_2020-02-12_12-52-36
created	No	Datetime	Creation time in UTC format	2020-02-12_12-52-36.0
content	No	Blob	Content of file	

4.3.11 Chain

This collection contains the definition of all the steps necessary for data elaboration processes, starting from the raw data acquired up to obtaining the outputs of the individual processes.

For each step, both the script to be executed and the configuration parameters are specified.

4.3.11.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Identifier. Different versions are specified by creation time	file_conversion
created	No	Datetime	Creation time in UTC format. The latest version of the chain is executed	2020-02-12_12-52-36.0
name	No	String	Name of the processing chain. Used to identify the latest version of a chain.	FileConversion
chains	Yes	List of String	List of sub-chains (if applicable)	["create_dem", "classify"]
command	Yes	String	Command or script to execute (if applicable)	python fileConversion.py
config	Yes	jsonObject	Configuration values of the script (if applicable).	{ "update": true, "q_file": { "file_type": "arw" } }
description	No	String	Verbal description of the processing chain	Converts files to a more useful file format

4.3.12 Measurement

Collection for storing all the measurements made, both through automatic sensors (for example weather station or soil sensor) and through manual surveys by experts.

If the measurements are acquired through the webapp interface, they can be connected to the activity stored in the Activity collection.

In addition, each measurement can be assigned to a target that identifies the tree or area of the field to which the measurement refers.

4.3.12.1 Field description

Key	Optional	Data type	Description	Example
id	No	String	Unique identifier	soil_42_temp
created	No	Datetime	Creation time in UTC format	2020-02-12_12-52-36.0
id_sensor	No	String	The senor the measurement was recorded by	TN_01
type	No	String	Type of measurement • soil_moisture • soil_temperature • air_pressure • air_temperature • air_humidity • wind_speed • wind_direction • solar_radiation • number_of_suckers • NDVI • CWSI • 3D_model • ...	soil_temperature
processing_level	No	Enum of String	Specifies the processing level of a measurement to distinguish raw measurements from higher order products. Processing levels: • Raw • processed	Raw
unit	No	String	SI unit	°
value	Yes	Float	Measurement value	13
id_task	Yes	String	Used to link a file to a specific task	Tree_Geometry_ Reconstruction
id_trial	Yes	String	Used to link a measurement to a specific trial	pruning_Yo
id_activity	Yes	String	Used to link a measurement to a specific activity	105
id_target	Yes	String	A geoObject the measurement is assigned to (if applicable)	Field_16
id_group	Yes	String	Used to aggregate a collection of measurements	soil_42

4.4 Agronomical activities

Collections dedicated to the management of agronomic activities in the hazelnut field.

The collection activity, as shown in Figure 13, is populated both by the process of processing the data acquired automatically and through the web application.

The collection mission allows to group together homogeneous activities and plan operations to be carried out in the orchard.

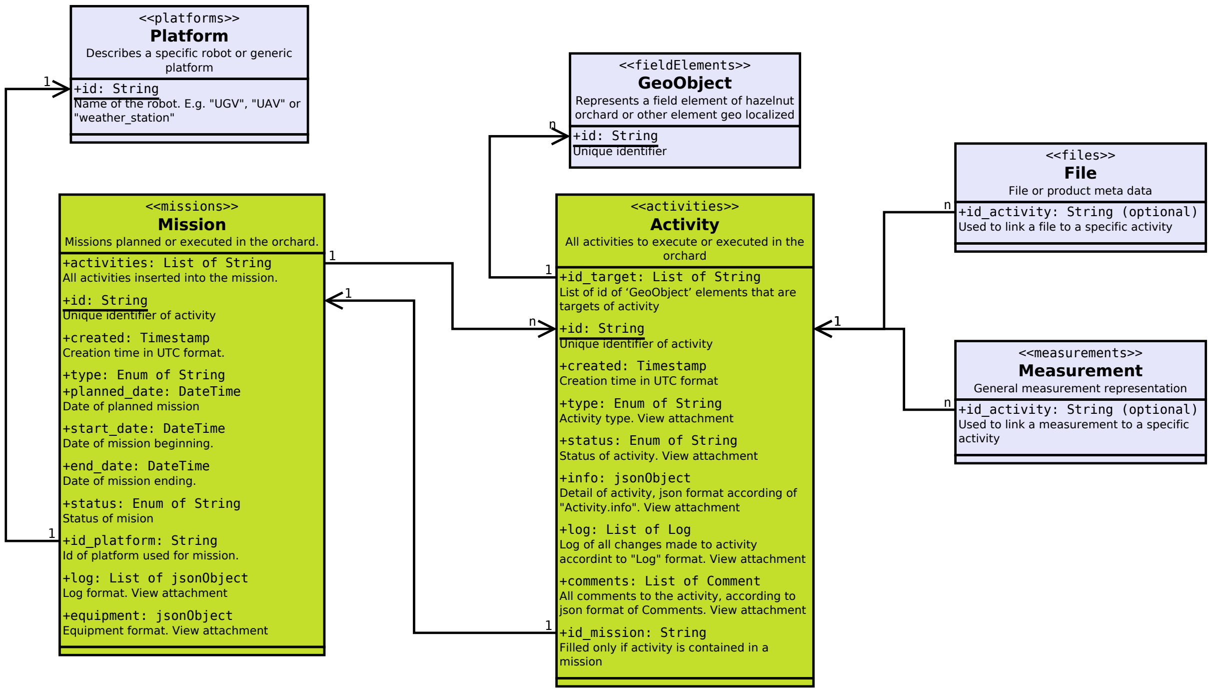


Figure 13 - Agronomical activities group elements of the data model schema

4.4.1 Activity

This collection contains all the activities performed, or to be performed, in the hazelnut field. These activities can be generated automatically by the acquired data elaboration processes or can be entered manually using the application user interface. Furthermore, through the activities it is also possible to manage new data acquisition operations.

The activities also include collection and sales operations which will be used as inputs to estimate the production and profit of the following years.

Through the status of the activity it is possible to manage the phases of the activity, approve or discard activities that have been automatically generated by the system, plan them and keep track of the history of the activities carried out.

Each activity is associated with the target that identifies the objective of the operation to be carried out.

By grouping several homogeneous activities in a mission, it is possible to plan the execution date of the various operations, in which case the id_mission field will be enhanced.

Through the 'info' field, detailed information is specified for each type of operation, for example the branches to be cut for the pruning activity or the pesticide to be administered in the case of pest control activities. In addition, the collection will contain the logs of changes made to the activity over time and the comments that operators will have entered through the application.

4.4.1.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Unique identifier	105
created	No	Datetime	Creation time in UTC format	2020-01-12_14-41-23.0
type	No	Enum of String	Activity type. Enum value: • pruning • sucker • water • pest • uav_data_acquisition • ugv_data_acquisition • manual • harvest • sales	pruning
status	No	Enum of String	Status of activity. Enum value: • suggested • ready • planned • executed • rejected	ready
info	Yes	jsonObject	Detail of activity, json format according of “Activity.info”	View Activity.type = “pruning” example
log	Yes	List of Log	Log of all changes made to activity. Embedded documents of “Log” object	View Log example
comments	Yes	List of Comment	Embedded documents of “Comment” object. All comments to the activity	View Comment example
id_mission	Yes	String	External reference “Mission.id”. Filled only if activity is contained in a mission	null
id_target	No	List of String	List of id of ‘GeoObject’ elements that are targets of activity	[‘YOA9’]

4.4.1.2 JSON Format – Activity.info

Activity.type = “pruning”

Key	Optional	Data type	Description	example
branches	No	List of String	List of branches to cut	[2, 7]
model	No	String	Link to a measurement object, representing the 3D model of the tree.	tree_geometry_5

Activity.type = “sucker” | “pest”

Key	Optional	Data type	Description	example
id_chemical	No	String	Identifier of the chemical product to give	SULPH
quantity	No	Float	Ml of product to give	20

Activity.type = “water”

Key	Optional	Data type	Description	example
id_valve	No	String	Identifier of the irrigation valve	valve_1
time	No	Integer	Minutes for m3 hectare	30

Activity.type = “uav” | “ugv”

Key	Optional	Data type	Description	example
info	Yes	String	Other detail and note about activity	Geometry tree reconstruction campaign acquisition

Activity.type = “manual”

Key	Optional	Data type	Description	example
type	No	Enum	Activity type. Enum value: • pruning_required • sucker_detection • damage_detection • pest_detection • disease_detection • water_required • other	pest_detection

info	Yes	jsonObject	JSON Format “Activity.info”	{
				“bug” : “Hazelnut mite”,
				“quantity” : 25
				}

Activity.type = "harvest"

Key	Optional	Data type	Description	example
farm	No	String	Id of Farm. Reference to 'GeoObject' element	Field_16
date	No	DateTime	Date of harvest operation	2019-09-03_00-00-00.0

Activity.type = "sales"

Key	Optional	Data type	Description	example
date	No	DateTime	Date of sale operation	2019-10-15_00-00-00.0
quantity	No	Float	Quantity of product of the sale	30
quality_band	No	String	Reference to quality band of product	TRG_INT_FQ
total_price	No	Float	Total price of the sale	27450
id_targets	Yes	List of String	List of id of 'GeoObject' elements	Field_18
id_price	Yes	String	Reference to priceData applied. Id of 'PriceData' element	P1_859678

4.4.2 Mission

The mission data, their planning and the log of their execution are stored in this collection.

The missions represent groupings of activities of the same type, which have an assigned planning date.

To start the mission, it is necessary to specify the platform that will perform the mission, for example UGV, and its equipment that varies according to the mission (for example herbicide and quantity loaded in case of detection of sucker).

After the start of the mission, all the status changes and its execution are tracked in the logs.

4.4.2.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Unique identifier	M_2020_01_13_1035
created	No	Datetime	Creation time in UTC format	2020-01-13_10-35-58.0
type	No	Enum of String	Enum value: • pruning • sucker • water • pest • uav • ugv • manual	sucker
planned_date	No	DateTime	Mission planning date	2020-01-15_11-00-00.0
start_date	Yes	DateTime	Real date of beginning of mission	2020-01-15_11-18-45.0
end_date	Yes	DateTime	End date of mission	null
status	No	Enum	Enum value: • planned • in_progress • paused • executed_partially	in_progress

			• executed	
id_platform	Yes	String	Platform used for mission execution	UGV

log	Yes	List of Log	Embedded documents of 'Log' object	[
				{
				"created" : 2020-01-13_10-35-58.0,
				"id_user" : 5,
				"operation" : "create_mission",
				"note" : null
				},
				{
				"created" : 2020-01-15_11-18-45.0,
				"id_user" : 5,
				"operation" : "start_mission",
				"note" : null
				}
				]

activities	No	List of String	List of id of 'Activity' elements	["115","123","148"]

equipment	Yes	jsonObject	Detail of platform equipment. Json object in "Mission.equipment" format	{
				"id_chemical" : "SULPH"
				"quantity" : 150
				}

4.4.2.2 JSON Format – Mission.equipment

Mission.type = "sucker" | "pest" | "nutrition"

Key	Optional	Data type	Description	example
id_chemical	No	String	Name of product	SULPH
quantity	No	Float	Total quantity of product	200

4.5 User application

All support information for the user application (shown in Figure 14).

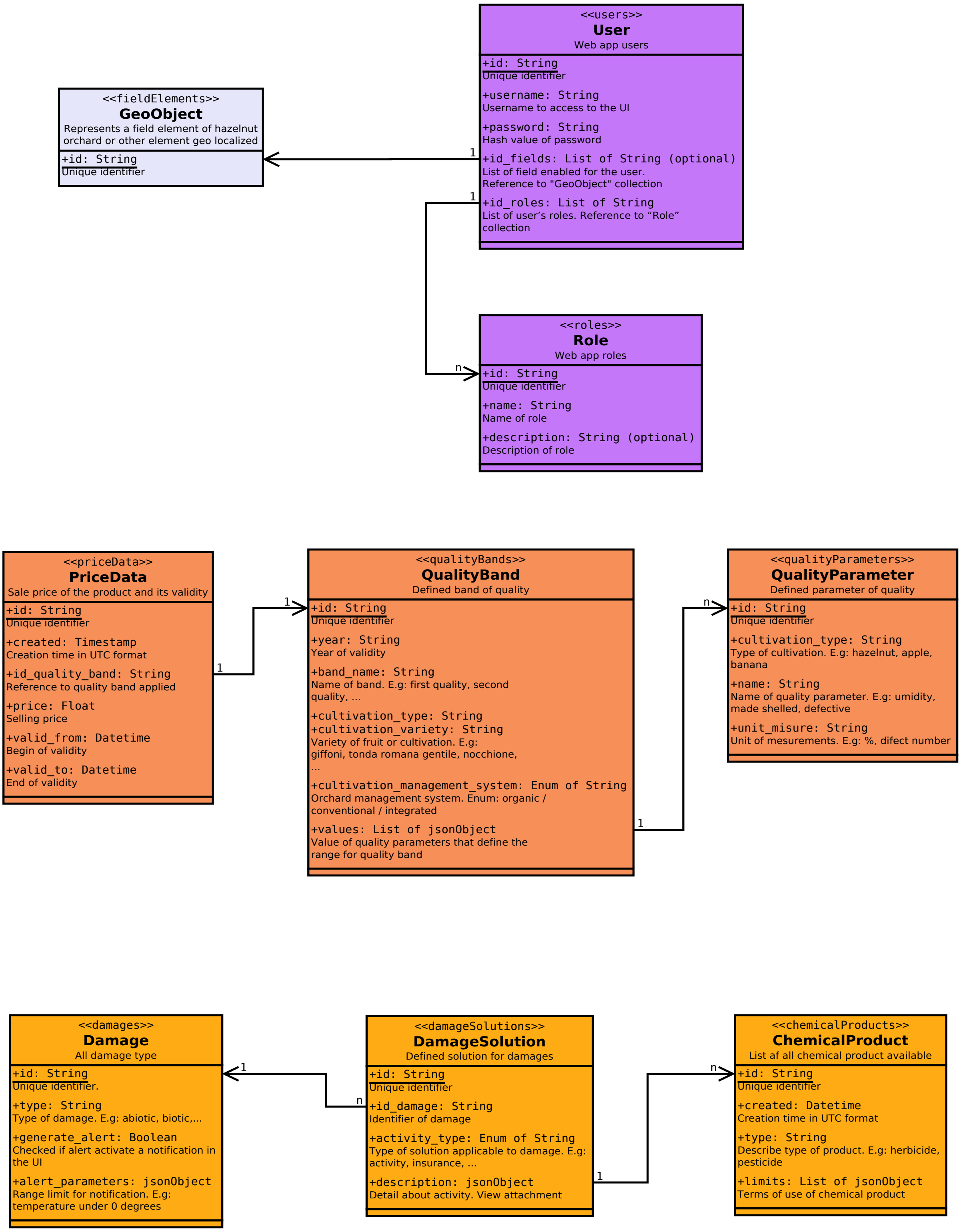


Figure 14 - User application group elements of the data model schema

Users and roles to allow secure access to the application and to the individual functions, qualitative parameters and price history to define sales prices and to make forecasts, damages and chemicals available to support the execution of agronomical activities.

4.5.1 Role

This collection stores the roles of the user application that discriminate the features that will be activated for the various users of the application.

4.5.1.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Unique identifier	agronomist_01
name	No	String	Name of role	agronomist
description	Yes	String	Description of role	Role of expert agronomist

4.5.2 User

The user collection contains the list of users registered in the system.

Each user can be assigned one or more roles and the fields for which it is enabled.

4.5.2.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Unique identifier	10
username	No	String	Username to access to the UI	nick_jones@pantheon.com
password	No	String	Hash value of password	9e3bc74930b431c77afaf99 c2902ea1f302d0083
id_fields	Yes	List of String	List of id_field enabled for the user. Reference to “GeoObject” collection	[“pantheon_field”]
id_roles	No	List of Sting	List of user’s roles. Reference to “Role” collection	[“agronomist”, “user_admin”]

4.5.3 QualityParameter

Through this collection, the qualitative parameters used to define the quality and price of sales of the harvest are defined.

4.5.3.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Unique identifier	H_MSH
cultivation_type	No	String	Type of cultivation	hazelnut
name	No	String	Name of quality parameter. E.g: humidity, made shelled, defective	made shelled
unit_measure	No	String	Unit of measurement. E.g: %, defect number	%

4.5.4 QualityBand

This collection stores the quality bands to define the sale price of the harvest.

The quality bands change annually based on the cultivation management system.

For each quality band, a set of quality parameters is stored with the minimum and maximum reference values that identify their characteristics.

4.5.4.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Unique identifier	TGR_INT_FQ
year	No	String	Year of validity	2019
band_name	No	String	Name of band	first quality, second quality
cultivation_type	No	String		hazelnut
cultivation_variety	No	String	Reference to "Cultivation_variety" id	TGR
cultivation_management_system	No	Enum of String	Orchard management system. Enum value: • organic • conventional • integrated	integrated
values	No	List of jsonObject	Value of quality parameters that define the range for quality band. Embedded json of format 'QualityBand.value'	View example of 'QualityBand.value'

4.5.4.2 JSON Format – QualityBand.value

Key	Optional	Data type	Description	example
id_quality_parameter	No	String		H_MSH
min_value	No	Float		80
max_value	No	Float		90

4.5.5 PriceData

In this collection the prices established for the product are saved based on the quality ranges and, consequently, on the type of cultivation.

For each price, the validity must be specified as it can vary over time, in this way the history of price changes is also stored.

4.5.5.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Unique identifier	P1_859678
created	No	Datetime	Creation time in UTC format	2019-09-20_18-05-33.0
id_quality_band	No	String	Reference to quality band applied	TGR_INT_FQ
price	No	Float	Selling price	15.20
valid_from	No	Datetime	Begin of validity	2019-10-01_00-00-00.0

valid_to	No	Datetime	End of validity	2019-12-01_00-00-00.0

4.5.6 CultivationVariety

List of product varieties managed in the project.

4.5.6.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Unique identifier of variety	TGR
name	No	String	Name of variety	Tonda gentile romana

4.5.7 ChemicalProduct

Collection that stored the list of chemicals available in cultivation.

Each document contains the history of all its variations of the application limits.

The application limit is defined in a json structure included in the document that specifies the terms of validity, the minimum and maximum quantity of product that can be administered and the maximum number of doses.

The limits of application of a chemical differ according to the cultivation method.

4.5.7.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Unique identifier	SULPH
created	No	Datetime	Creation time in UTC format	2018-11-15_13-40-05.0
type	No	String	Describe type of product. e.g.: herbicide, pesticide	Sulphur
limits	Yes	List of jsonObject	JSON Format "ChemicalProduct.limit"	View JSON Format – ChemicalProduct.limit example

4.5.7.2 JSON Format – ChemicalProduct.limit

Key	Optional	Data type	Description	example
created	No	Datetime	Creation time in UTC format	2018-11-15_13-40-05.0
valid_from	No	Datetime	Start of validity for the administration limit	2019-01-01_00-00-00.0
valid_to	No	Datetime	End of validity for the administration limit	2019-12-31_00-00-00.0
min	Yes	Float	Minimum quantity that can be administered	15
max	Yes	Float	Maximum quantity that can be administered	20
num_apply	Yes	Float	Maximum number of administrations allowed	null
cultivation_management_system	No	Enum of String	Orchard management system. Enum value: • organic	integrated

			• conventional • integrated	

4.5.8 Damage

This collection stores information on the various damages that may occur in the orchard, it is possible to indicate whether it must provide notification in the user application and the range of threshold values that trigger the alarm.

4.5.8.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Unique identifier	PEST_HM
description	No	String	Description of damage	Hazelnut mite
type	No	String	Type of damage. E.g: abiotic, biotic	pest
generate_alert	No	Boolean	Checked if alert activate a notification in the UI	false
alert_parameters	Yes	List of jsonObject	Range limit for notification. Each one in and condition. E.g: Temperature under 0 degrees for 3 hours	null

4.5.8.2 JSON Format – Damage.alert_parameter

Key	Optional	Data type	Description	example
subject	Yes	String	Type of damage	Temperature
min_value	Yes	String	Minimum threshold of acceptable value range.	0
max_value	Yes	String	Maximum threshold of acceptable value range.	45
unit	Yes	String	SI unit of value	°

4.5.9 DamageSolution

The collection stores the activities, and the related details, which must be implemented to correct the damage that can occur in cultivation.

This information is used to configure automatic processes for generating suggested activities.

4.5.9.1 Field description

Key	Optional	Data type	Description	example
id	No	String	Unique identifier	Damage_01
id_damage	No	String	Identifier of damage	PEST_HM
activity_type	No	Enum of String	Type of solution applicable to damage. Enum value: • activity • insurance • nothing	activity
description	Yes	jsonObject	Detail about activity. According to "DamageSolution.description"	

4.5.9.2 JSON Format – DamageSolution.description

DamageSolution.activity_type = "activity"

e.g.: agronomical activity or pesticide administration

Key	Optional	Data type	Description	example
activity_type	No	Enum of String	Type of activity Enum value: • pruning • sucker • water • pest • manual	pest
id_chemical_product	Yes	String	Unique identifier of a chemical product to use	SULPH
quantity	Yes	Float	Quantity of product to administer	15
unit	Yes	String	Unit measure of product	Ml
note	Yes	Text	Note and other operation to do	Only 15-20% of buds

DamageSolution.activity_type = "insurance"

Key	Optional	Data type	Description	example
name	No	String	Name of assurance	Assurance_0 1
policy_number	No	String	Policy number	123456
valid_from	No	DateTime	Start of validity	01/01/2020
valid_to	No	DateTime	End of validity	31/12/2020
note	Yes	Text	Note and other operation to do	

activity_type = "nothing" | null

Key	Optional	Data type	Description	example
note	Yes	String	Note and other operation to do	A simple note

4.6 Embedded data

Structure of embedded document, like logs and comments (Figure 15).

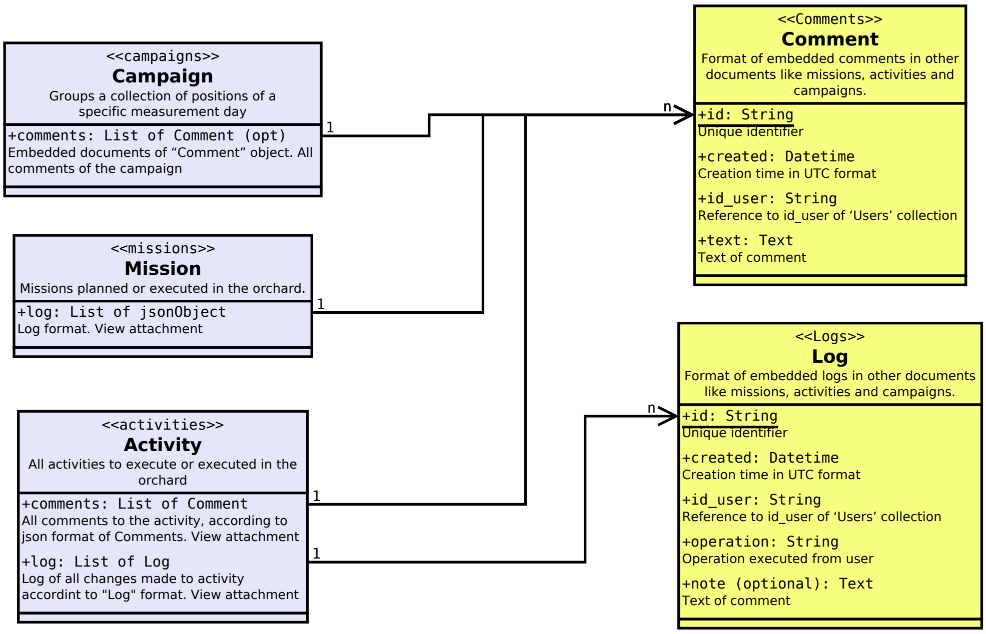


Figure 15 - Embedded data group elements of the data model schema

4.6.1 Comment

This scheme represents the JSON format for storing the comments of the users to various types of content such as missions, activities and campaigns.

Using a MongoDB database, a separate collection is not defined for this type of data but will be included directly embedded in the documents to which they refer.

4.6.1.1 Field description

Key	Optional	Data type	Description	example
created	No	Datetime	Creation time in UTC format	2020-01-15_11-18-45.0
id_user	No	String	Reference to id_user of 'Users' collection	5
text	No	Text	Text of comment	Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

4.6.2 Log

This scheme represents the json format for storing logs of various types of content such as missions, activities and campaigns.

Using a MongoDb database, a separate collection is not defined for this type of data but will be included directly embedded in the documents to which they refer.

4.6.2.1 Field description

Key	Optional	Data type	Description	example
created	No	Datetime	Creation time in UTC format	2020-01-15_11-18-45.0
id_user	No	String	Reference to id_user of user's collection	5
operation	No	String	Operation executed from user	start_mission
note	Yes	Text	Other detail about operation	Null

4.7 Configuration data

4.7.1 Disease

This data will populate the "Damage", "DamageSolution" and "ChemicalProduct" collections for the part of the data that relates to pest and disease and other damages.

Disease (causal agent)	2019	Agronomic actions	Active Ingredients for control	Intervention threshold
«Mal dello stacco» (Cytospora corylicola)	present	Removal and destruction of affected plant parts (burning) After pruning, disinfection of cuts and protection with sealing compounds	Copper compounds (max 4 Kg ha/year) 2 scheduled treatments (late summer and vegetative restart)	Not available
Nut Grey Necrosis (Fusarium lateritium)	present	-	Pyraclostrobin + Boscalid max 2 treatments/year (according to symptoms appearence)	Not available
Brown rot of nuts (Monilia fructigena)	present	Removal and destruction of affected hazelnuts. Protection of plants from injuries	Thiophanate-methyl Only in wet and warm seasons and during early fruiting	Not available
Gleosporiosi (Piggotia coryli)	present	-	Thiophanate-methyl max 1 treatment/year (early autumn - before leaves fall)	Not available
Bacterial blight (Xanthomonas arboricola pv. corylina)	present	Removal and destruction of affected plant parts (burning) Sterilization of tools during pruning and disinfection of cuts	Copper compounds (max 4 Kg ha/year) 2 scheduled treatments (late summer and vegetative restart) Additional treatment in case of late frost damages	Not available
Bacterial canker «Moria» (Pseudomonas avellanae)	absent	Suckers removal After pruning, disinfection of cuts and protection with sealing compounds	Copper compounds (max 4 Kg ha/year) Acibenzolar-S-methyl severe symptoms: 2 treatments in autumn (begin of leaves fall and half leaves fall) 1 or 2 additional treatments at vegetative restart. slight symptoms: 1 treatment at leaves fall and 1 at vegetative restart.	Not available

4.7.2 Pest

This data will populate the "Damage", "DamageSolution" and "ChemicalProduct" collections for the part of the data that relates to parasites.

Insect Pests	2019	Agronomic actions	Active Ingredients for control	Intervention threshold
Cimici (Gonocerus acuteangulatus, Palomena prasina etc.)	present		• Piretrum • Mineral oil • Azadiractin A • Indoxacarb • Lambda-cyhalothrin • Etofenprox	2 specimens/tree
Halyomorpha halys	present		• Deltametrin • Etofenprox	----
Hazelnut mite (Phytoptus avellanae)	present		• Sulphur • Mineral oil	15/20% of buds
Hazelnut weevil (Curculio nucum)	absent		• Clorpirifos • Deltametrin • Lambda-cyhalothrin • Mineral oil • Indoxacarb • Fosmet • Etofenprox • Metam potassium	2 specimens/tree

4.7.3 Other damages

Additional types of damage that may occur in the harvest with an indication of the activity to be undertaken and the condition for activating the notification.

Damage	Agronomic actions	Insurance	Alert
cold damage	Yes	yes	sub-zero temperature (-1 degree)
wind damage	No	yes	higher than X knots
drought damage	No	yes	temperature exceeds 35 for total hours (e.g. 3 hours)
high temperature	No	yes	temperature (above 35 degrees)
ungulates (wild boar and roe deer)	No	yes	presence
dormouse and squirrel	No	yes	presence

5 Appendix

5.1 Annex 1 - Complete data model schema

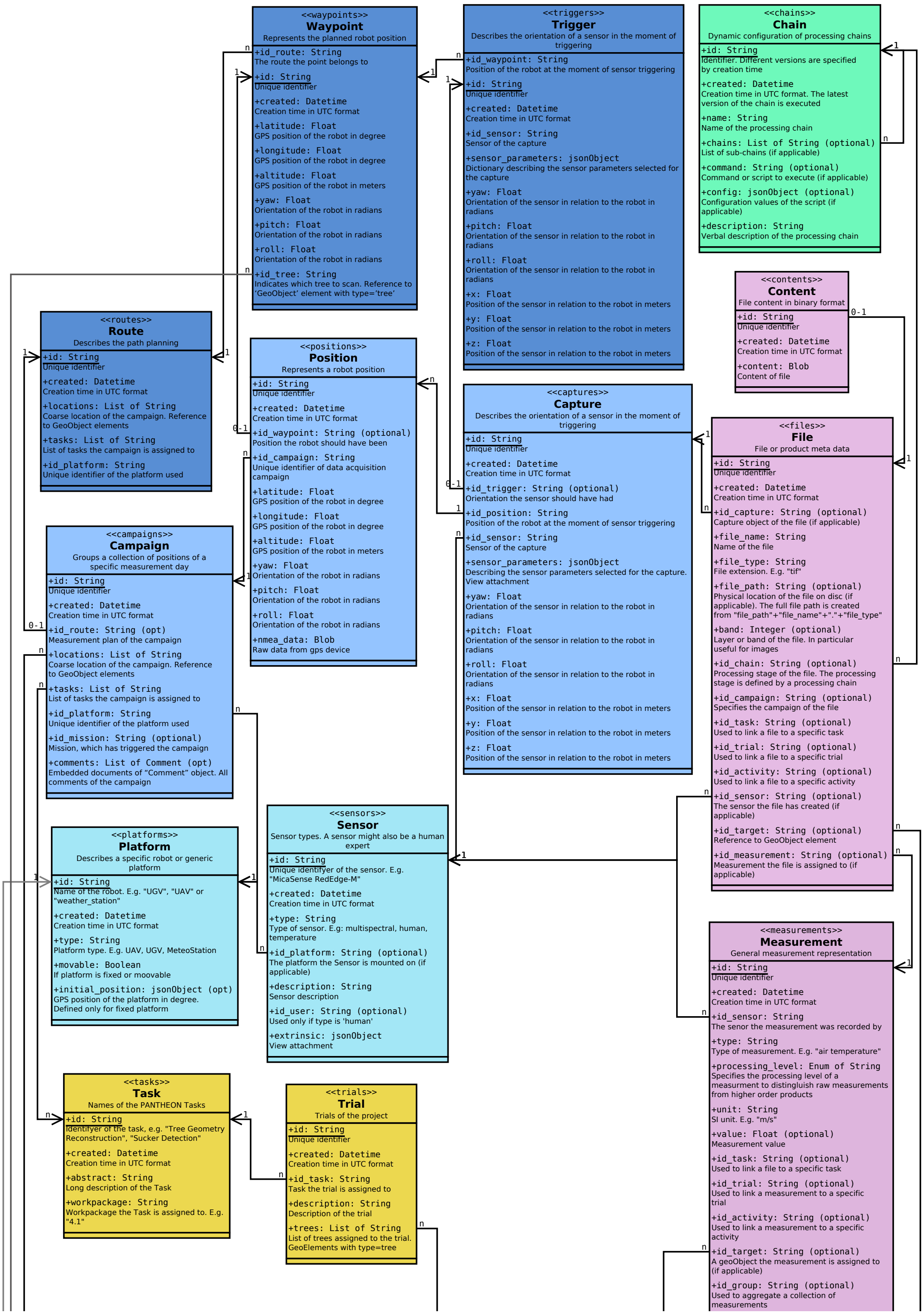


Figure 16 - Full data model schema (part 1 of 2)

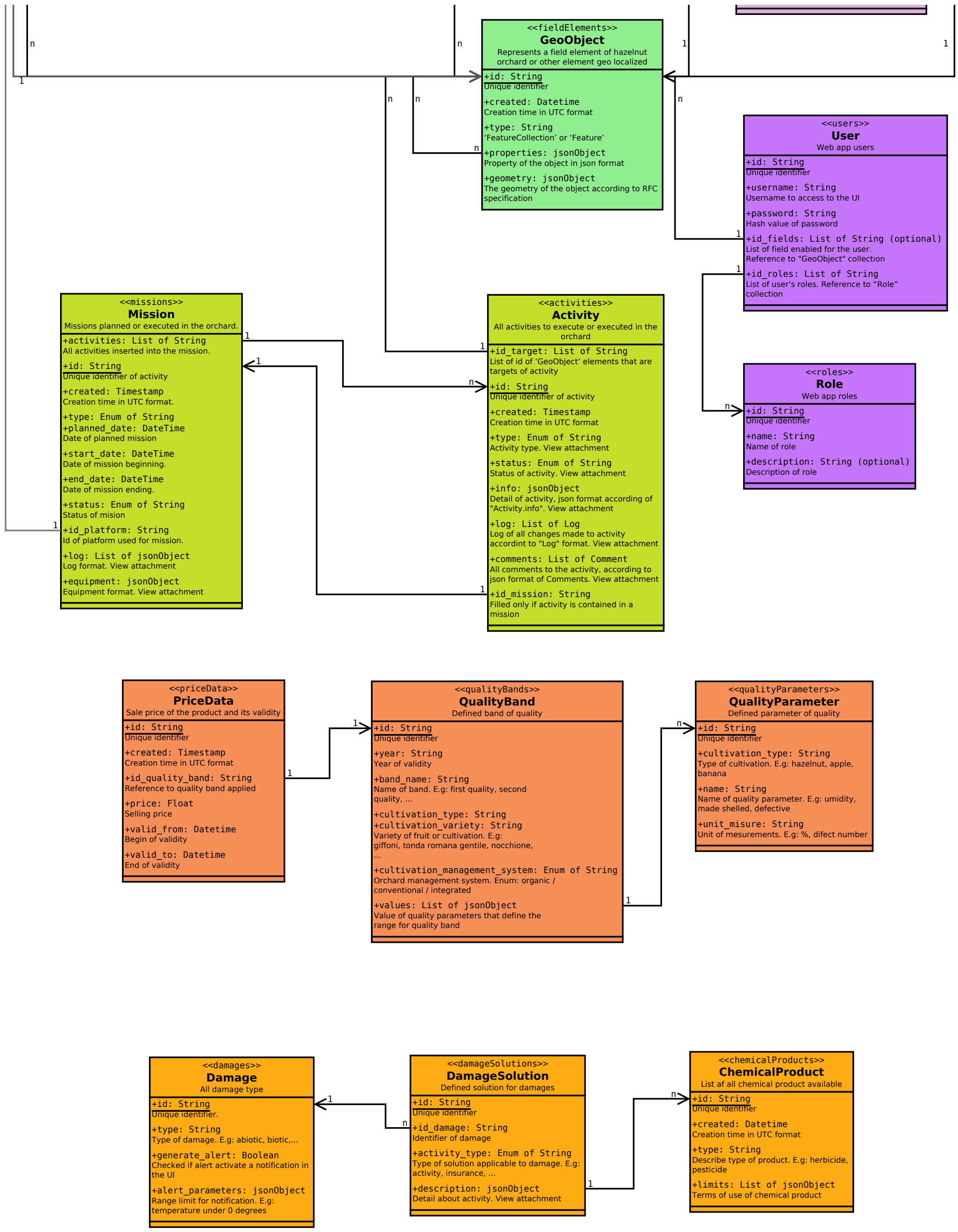


Figure 17 - Full data model schema (part 2 of 2)

6 References

[1] L. Giustarini, S. Lamprecht, R. Retzlaff, T. Udelhoven, N. Rossellò Bono, E. Garone, V. Cristofori, M. Contarini, M. Paolocci, C. Silvestri, S. Speranza, E. Graziani, R. Stelliferi, R. F. Carpio, J. Maiolini, R. Torlone, G. Ulivi and A. Gasparri, “PANTHEON: SCADA for Precision Agricolture,” in *Handbook of Real-Time Computing*, Springer, 2019.

[2] W. Shi and S. Dustdar, “The Promise of Edge Computing,” *Computer,* vol. 49, no. 5, p. 81, 2016.

[3] J. Maiolini, C. Potena, R. F. Carpio, E. Garone and A. Gasparri, “MP-STSP: A Multi-Platform Steiner Travelling Salesman Problem Formulation for Precision Farming in Large-Scale Orchards,” in *ICRA*, 2020.

[4] C. Chasseur, Y. Li and J. Patel, “Enabling JSON Document Stores in Relational System,” WebDB, 2013.

[5] IETF (Internet Engineering Task Force), “The GeoJSON Format,” 2016.

[6] C. Fairchild and T. Harman, “ROS nodes, topics, and messages,” in *ROS Roborics by Example*, Packt, 2017, p. 484.

[7] A. Pulter, S. Johnston and S. Cox, “Using the MEAN stack to implement a RESTful service fon an Internet of Things application,” in *2015 IEEE 2nd World Forum on Internet of Thinghs (WF-IoT)*, Milan, Italy, 2015.

[8] S. Lamprecht, “Pyoints: A Python package for point cloud, voxel and raster processing,” *Journal of Open Source Software,* vol. 4, no. 36, p. 990, 2019.

[9] Agisoft LLC, *Metashape Python Reference (Release 1.6.2),* 2020.

[10] Agisoft LLC, *Agisoft Metashape User Manual (Version 1.6),* 2020.

[11] M. Dirolf, *PyMongo - the Python driver for MongoDB,* 2020.

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot and E. Duchesnay, “Scikit-learn: Machine Learning in Python,” *Journal of Machine Learning Research,* vol. 12, pp. 2825--2830, 2011.