Description and overview of

GROW WITH THE FLOW PLATFORM

Yaniv Ben-Yosef

Milan Innovincy B.V.

Version 1.08 - November 2021

Table of Contents

1 Preface

1.1 Purpose of this document

The purpose of this document is to provide an entry point into understanding the GWTF platform, including goals, principles, architecture, software design, algorithms and operation. This document can help understand how the system works, how to deploy it and how to perform day-to-day operations of the platform.

1.2 Target audience

The main audience of this document is GWTF group members who are interested in understanding the inner working of the platform, as well as newly joined members who will want an entry point to the platform to get up to speed quickly.
The document is technical in nature, and the top priority is to address experienced software developers. However, the document strives to be as clear as possible for the non-technical person so that they can also benefit from it.

2 Introduction

2.1 What is the Grow-with-the-Flow (GWTF) project?

Water is an important resource for the Agro and Food sector. At the same time, partly due to climate change, the risk of water scarcity has increased. This has major implications for securing our food supply in the long term.

Various parties are involved in managing water resources and making water available. Each from its own discipline and with its own objectives. The efficient management of water resources and the provision of water therefore requires cooperation between the parties.

A number of these parties have found each other in a proposed collaboration under the TKI Agro & Food/TKI Deltatechnology subsidy programs to achieve a better collaboration, not limited to the current participants, with the aim of developing innovative tools that focus on better managing the water resources, considering mutual interests.

These parties are:

- Waterboard Aa & Maas
- Waterboard Vallei and Veluwe
- Achmea Agro Insurance
- Lamb Weston/Meijer
- Deltares
- Wageningen Environmental research
- Cap Gemini

2.2 What is the GWTF Platform

Since GWTF is intended to be used by different parties of several types (farmers, waterboards, corporations) there needs to be a central repository where data is stored and processed, and from which it can be consumed.

GWTF Platform is a software system designed to be that central storage and processing of GWTF, which provides ways for ingesting information from multiple sources, processing it, and providing ways to expose the information to interested parties.

2.3 Philosophies and principles

The GWTF partners have developed a set of philosophies and principles that serve as a guideline for the project. The final result was a group effort which is shown in the following page.

Principles	Preliminary	Vision	Business	Data	System	Solutions	Technical	Operation
Business view	As a system integrator, we see opportunities to deliver our expertise at the regional water authorities	Being a trustworthy, economically viable, skilled partner in the consortium that keeps adding new features to the platform and extract business value from the platform	We create a new offering in agribusiness	Farmers will remain the owner of their data. Insights gathered and calculated on the platform remains on the platform and remains owned by the platform	Safeguarding an open platform	Solution architecture, platform development according to design thinking principles	For the platform to be viable (in technical and economical terms), there needs to be a cloud provider situated in NL capable of doing sub-second calculations on big data sets against fair prices	With GwtF up and running, we can provide integration of GwtF to RWAs
Organization structure View	There will come a legal construct, fe NGO, BV, NV or...	The data domain for the irrigation	We create a new offering, businessmodel in agribusiness	we support in the safeguarding of the process best use of the data - information	support cooperation in the total eco system of partners en hard- and software	For-profit, aiming for continuous improvement of business solutions	Technical working platform remains high priority within the organization	The organization can hire workforce from Capgemini to do the maintenance work
Service	Open service: users that have been given access to the platform can use the platform to all functionalities that the platform offers	Advice on irrigation	24*7 service window	It will be made as easy as possible to deliver data, on a fit-for-purpose (F4P) principles	Integration of application system	Solutions are open for clients that want to create win-win services	Technology is selected based on the service, not the other way around	maintaining speed in the process
Architecture	use of best practices in the market	Open data structure with a highly secure and transparent architecture	Availability 99%, Transparant what is delivert by who	Data is owned by the creator . With in the platform all created data can be used	Reuse if possible existing systems. The focuses is to create reusable components for future duplication and scalability	Adopted solutions must fit to the requirements of the architecture in the sense that they are open and scalable and vice-versa	Make use of common technology that is available in the market	One Organisation is the focal point from a operational point. All other operation actifties have a back to back with the focal point.
Minimal goals	succesful sharing of selected data between all platforms	MVP	break-even on investment	Minimum data used to give trustworthy	system supports the defined use-cases	Solutions fit the local context (at first water	Technical working platform	Help the clients of the platform with the promised services
Future goals	Create the go-to platform for agricultural stakeholders to provide trustworthy advice based on data sharing and analysis	real-time advice, predictive analytics for a much larger target audience than farmers	profitable and expandable business with a	Rich data platform that enables development of	duplicable system - allow multiple data sources	Solutions not necessarily all developed by consortium can also link	Development street continously working on improvements	Exceed clients' expectations by delivering the cherry on top
Capability	Trustworthy water use advice	project is groundbreaking in complexity and there fore	create win-win-win for GwtF,	Data does not have capabilities. The	Reuse if possible existing systems. The focuses is to create reusable	Solutions are capable of precisely informing users	Technology adds to the capability of the platform	Enough workforce to have a smooth operation
Security	The platform is sufficiently secure. Security needs to be proven with an ISO9000 certificate	Despite high security standards, no impediments on business	make clear how every participating party benefits	safeguarding data compliancy	no unintended data usage	Solutions are secure and shielded against malicious attacks, but do not drive up costs	Despite high security standards, no impediments on business	All workforce complies to legislation
Compliancy	Dutch legislation	Compliant and sufficiently adjustable towards future changes in compliancy regulation	Striving for ISO9000 : do what you say and say what you do	GDPR	Compliant and sufficiently adjustable towards future changes in compliancy regulation	ISO 27000 family	Compliant, according to technical industry standards	ISO 9000 family
Application	All individual applications in the application landscape need to adress the CSR(Sustainability) for irrigational purposes.	Application sets industry standard in water irrigation management	Business desires in line with app development	Visualization happens where possible on the GwtF	System supports the functionalities of the applications in the platform	application and solutions add value to the user experience - any	Applications are developed with the latest trends in app	Application works according to Design Thinking Principles
Infrastructure	We will exclude the development of proprietary products	Party that maintains and further develops the platform works via cloud-	Short lines between parties	Data is well-contained, yet openly available on	Infrastructure allows for IoT or other sources of data	Infrastructure serves solutions and is therefore adjustable to new	Infrastructure is scalable	There's a clear task package per workforce: RACI for operations has been made

2.4 Architectural guidelines

2.4.1 The SOLID principle

SOLID is a set of 5 important architectural principles that were applied together when creating the architecture and while developing the software:

Single Responsibility Principle

Each system capability (e.g. service/module/API) should have only one responsibility and as such one reason to change. Keeping the responsibilities as narrow as possible means that the users know of the intended purpose, which leads to less errors.

Open-Closed Principle

This principle postulates that it is preferable to extend a system behaviour, without modifying it. Although it is often not a good idea to try to anticipate changes in requirements ahead of time (as it can lead to over-complex designs), being able to adapt new functionality with minimum changes to existing components is key to the application's longevity.

Liskov Substitution Principle

In Software Development, this means that derived classes must be substitutable for their base classes, but this principle's resemblance with Bertrand Meyer's *Design by Contract* is how it can be applied to Distributed Architecture: two services communicate effectively and repeatedly when there is a common 'contract' between them, which defines the inputs/outputs, their structure and their constraints. Therefore: given two distributed components with the same contract, one should be replaceable with other component with the same contract without altering the correctness of the system.

Interface Segregation Principle

Interfaces/contracts must be as fine grained as possible and client specific, so calling clients do not depend on functionality they don't use. This goes hand in hand with the Single Responsibility principle: by breaking down interfaces, we favour *Composition* by separating by

roles/responsibilities, and *Decoupling* by not coupling derivative modules with unneeded responsibilities.

Dependency Inversion Principle

High level modules should not depend on low level ones; they should both depend on abstractions. Likewise, abstractions should not depend on details, but details should depend on abstractions. As such this principle introduces an interface abstraction between higher-level and lower-level software components or layers to remove the dependencies between them.

2.4.2 Reuse

Always aim to identify existing proven solution over developing from scratch and re-inventing the wheel. Prefer open-source solutions developed by the community, and solutions that are widely used in the industry to get a high level of support and maintenance.
Try to avoid or reduce lock-in to frameworks and libraries, especially commercial ones.

2.4.3 Reactive Programming

A complementary principle being applied is *Reactive Programming.* This is a relatively new term defined as: *an architectural style that enables applications composed of multiple microservices working together as a single unit to better react to their surroundings and one another, manifesting in greater elasticity when dealing with ever-changing workload demands and resiliency when components fail*. More about reactive programming and reactive system in the Reactive Manifesto website.

3 High-level architecture

3.1 Grow with the Flow architecture

The architecture is composed of several components:

1. Modelling Platform – the modelling platform collects data from various sources and maintains models about groundwater flow and crop growth. The models are deployed in each water authority. The modelling is built on the Delft-FEWS (a.k.a. FEWS) platform developed by Deltares.
2. GWTF Platform – the platform is responsible to
    a. collect data from the Modelling platform, ground truth from farmers, and data from other sources (like KNMI)
    b. store and process the data
    c. provide a unified API to access and provide data, for use by applications built for farmers, waterboard managers, and other stakeholders.
3. GWTF App – the official application built for the main stakeholders, mainly the farmers and waterboard managers. It provides a visualization using graphs, maps and data tables. It also provides the means to collect ground truth data from the user.
4. Corporate business systems – systems developed by corporations like insurance and food processors can integrate with the GWTF Platform API in order to collect data about plots and crops.

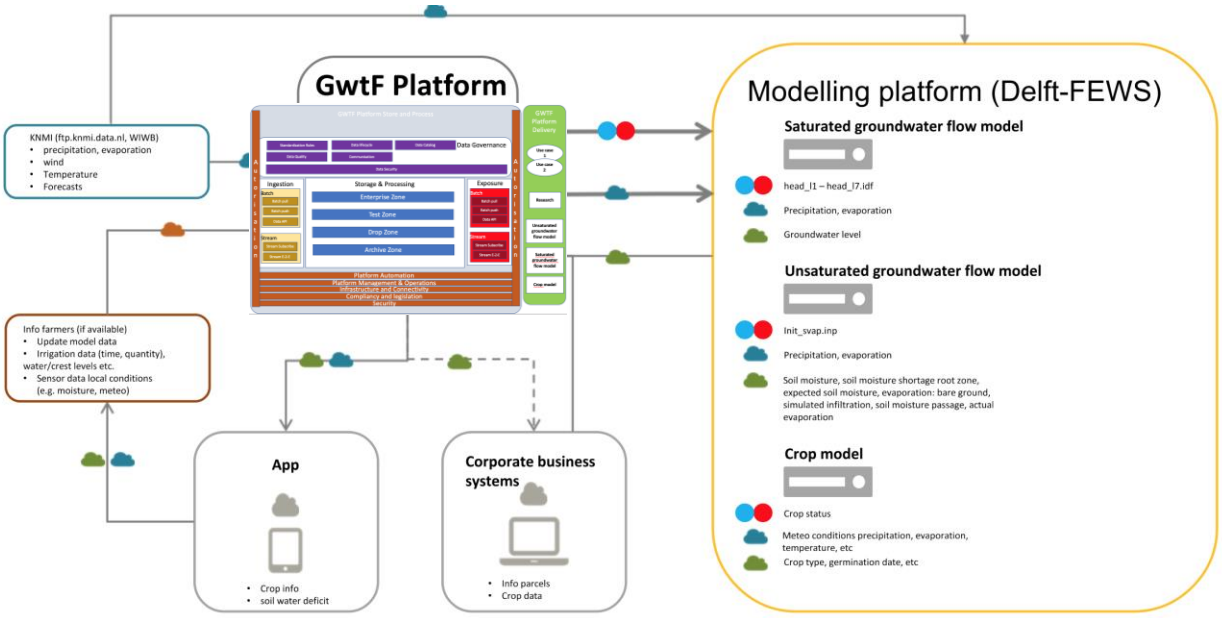


Figure 1: GWTF overall architecture

As mentioned above, the Modelling platform is responsible to maintain models that are the main source of data for the GWTF platform. There are two types of models in the Modelling platform:

1. Regional model – the model produces daily grids of variables over the modelled geographical area. In each day there are grids of a few days in the past and a few days of future forecast. See section 4.2.1.3 for more information about the structure of the model output.

   In addition to the grids, there is also an output per farmer plot. The Modelling platform is fed with the BRP data, which contains coordinates of each parcel (plot) owned by farmers in the Netherlands. The coordinates are overlayed on the grid, and the average of each variable is calculated per plot. The results are also provided daily, with past and forecast data.

Figure 2: Overlaying plot coordinates over a grid

2. Local model – the local model aims to provide a more accurate forecast at the plot level, than the regional model. It uses World Food Studies (WOFOST) to simulate crop growth and produces a subset of the variables provided by the regional model. The local model takes as input ground truth information from the GWTF platform (information collected from farmers: irrigation amounts and crop development stage). This information is fed back into WOFOST and helps making the model more accurate going forward.

3.2 Grow with the Flow Platform architecture

The GWTF Platform has an architecture blueprint, with four main components:

1. Ingestion – responsible for the data coming into the system from external sources
2. Storage & Processing – responsible for processing incoming data and storing it
3. Exposure – responsible for providing data to external consumers
4. Data Governance – responsible for controlling the data in the system, ensuring it's security, integrity, and lifecycle.

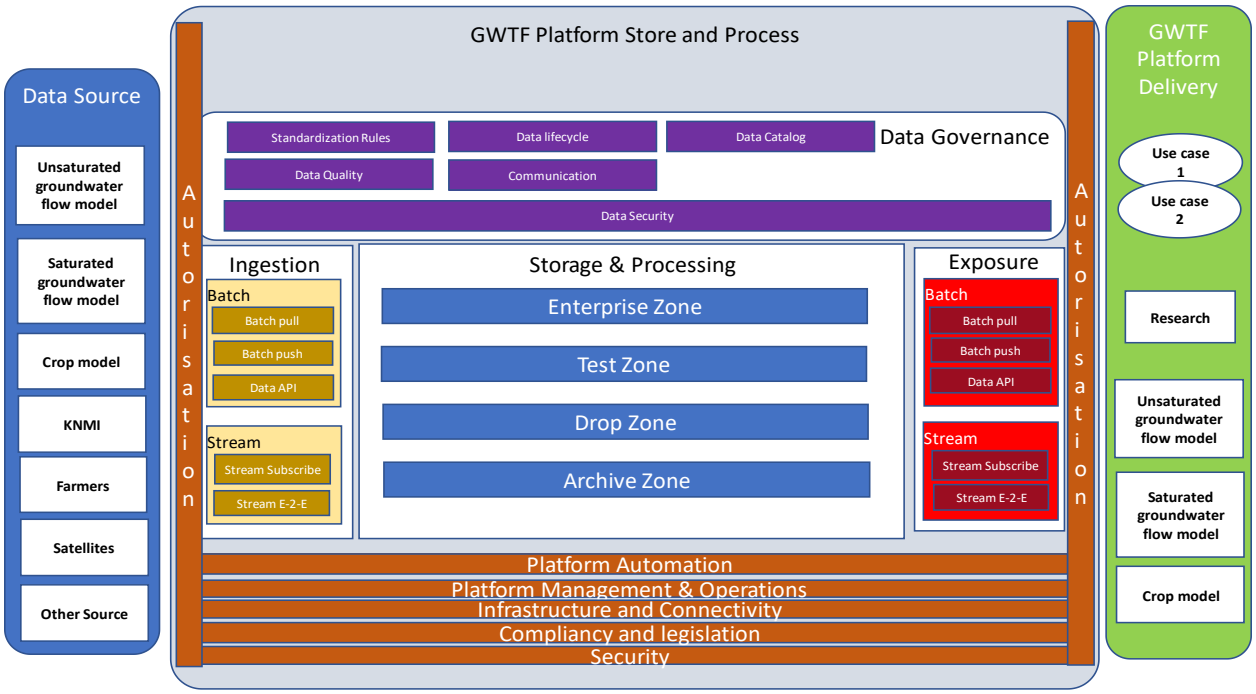
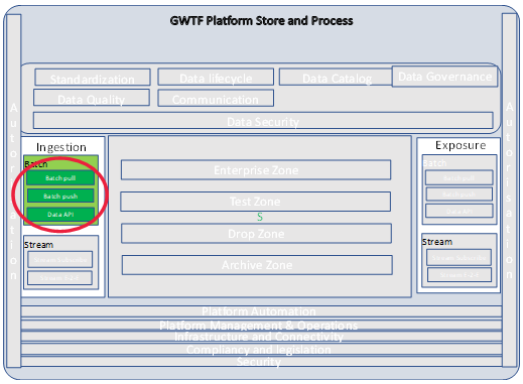


Figure 3: The GWTF Platform Architecture

The blueprint is the high-level plan used to design and develop the GWTF platform.
Not all parts of the blueprint are currently implemented. Some components were given lower priority than others because they were not needed in the existing use cases and only needed for the longer term.
The following sections go into further detail of each component and indicates the status of development.

3.2.1 Batch Ingestion

Purpose: a pipe that retrieves or receives data from a data source in bulk, prior to passing it on to further processing. The whole batch (usually that would be a data of a single day or some other fixed time) is downloaded and only then is processed and stored.



Should be used: when the data source provides data in batches, and when real-time processing is not critical.

Examples:

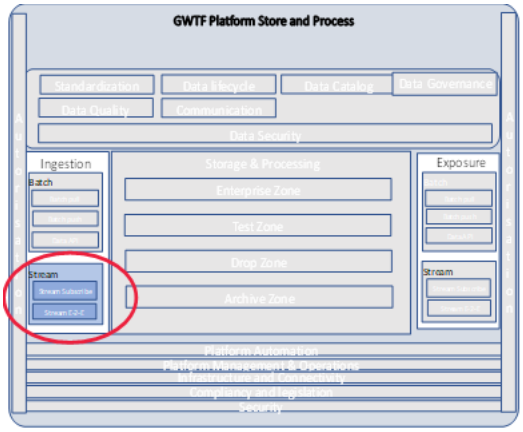
- KNMI data is in CSV format, can be retrieved in one batch per day or per hour for multiple weather stations at once (batch pull)
- Irrigation data provided by farmers is ingested by the system via push.

How:

1. Pull
   - For each pulled data source there is a schedule for retrieval
   - A scheduler handles the timings
   - Retrieved data is passed to the Storage & Processing component
2. Push
   - An HTTP endpoint is set to receive the data, optionally with additional metadata
   - Received data is passed to the Storage & Processing component upon retrieval

Status: Complete for the existing data sources

3.2.2 Stream Ingestion



Purpose: a pipe that retrieves or receives data from a data source in a continuous form (streaming), while passing it on to further processing

Should be used: when the data source provides data as a continuous stream and real-time processing is important.

Examples:

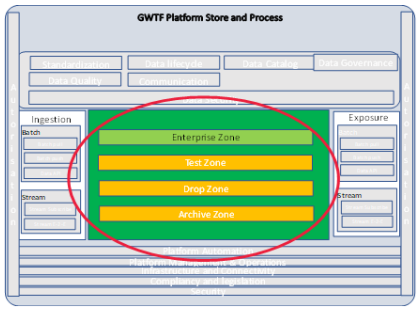
Currently there is no such data source in use, but one can think of a data stream such as one coming from real-time sensors

How:

The GWTF platform software stack already has support for streaming. The platform can use HTTP streaming, HTTP Long Polling and WebSockets in order to implement both streaming pull and push. The actual method would depend largely on the data source.

Status: Not implemented. The software stack used for developing GWTF supports streaming, however the existing data sources to be ingested do not require/need streaming.

3.2.3 Store and Process



Purpose: perform all necessary processing on raw data, which includes parsing or transforming to different formats and composing computed variables. Then index and store the data for later use or online retrieval

Examples:

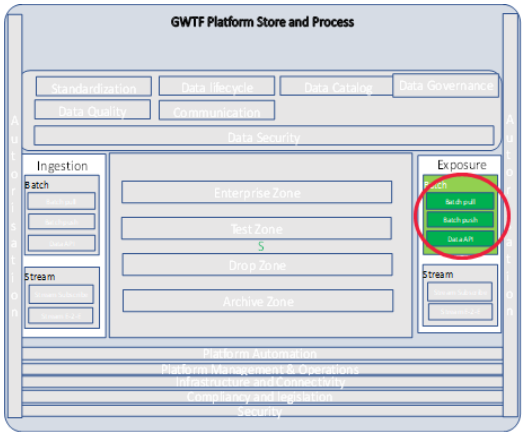
- Input from the modelling platform, in NetCDF format is parsed. The input is processed for computing additional variables and transforming units. Finally, the data is stored in the database.

Status: Implemented for the existing variables and data currently ingested.
Adding new variables usually won't require additional development unless format or data types are very much different, or unless there's any specialized algorithms needed to be applied.

3.2.4 Batch Exposure

Purpose: a pipe that sends data from the processing component to an external party, which may be the modelling platform, the GWTF Farmers' app or other 3rd party apps or consumers. The data is sent in batches, where the consumer needs to wait for each batch to arrive before processing it.

Should be used: when the data to be exposed is relatively small[1] or can be broken into smaller chunks.



Examples:

- Exposing the list of plots that the user is authorized to view. The response is a JSON array containing each plot as a JSON object.

How:

1. Pull

   The platform has a RESTful API that provides data in JSON format, which authorized clients may access.

   When data becomes too large to be fetched in one batch, it is sometimes possible to provide multiple batches by providing additional parameters to the request. E.g., `page=5&pageSize=10` would split the response into pages of 10 results each, and would return results 40-49

2. Push

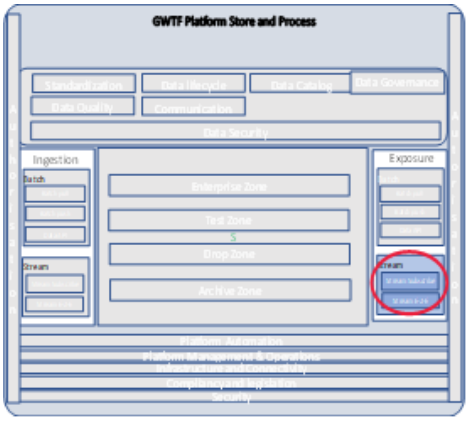
   Pushing out batch data can be accomplished using the Pub/Sub pattern, in which a client would subscribe to get data, provide an endpoint as part of the subscription, and the platform would push data by calling that endpoint. In a similar way as with Batch Pull, it is possible to split large results into smaller ones.

Status: Batch Pull is implemented for all RESTful APIs.
Batch Push is currently not implemented because there's currently no use-case for it.

[1] Small enough that the consumer can afford to wait for it and has enough resources to hold it in memory or disk in entirety.

3.2.5 Streaming Exposure

Purpose: a pipe that sends out data from the GWTF platform to external parties in a continuous form (streaming).



Should be used: when the the information is ongoing or changing rapidly, or when the data is very large and can be processed by the client in a stream (in other words, the client doesn't need all the information as one block)

Examples:

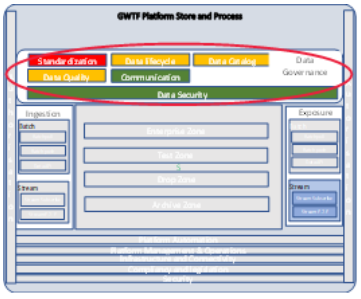
Currently there is no such data source in use, but one can think of a data stream that indicates the status of a large number of sensors in real time, possibly augmented with computation results done by the platform

Status:

Currently this is not implemented, but the software stack supports streaming when needed.

3.2.6 Data Governance

Purpose: Data Governance is a large topic and means different things to different people and different projects. We define it as the means to centralize the data in the organization and enforce a common set of data types, rules and quality such that the data has a uniform meaning across the organization and is easier to manage and maintain.



We view the platform's data governance as one that should ultimately include at least the following:

- Data Security – ensuring the data is encrypted and therefore safer against attacks on physical servers
- Communication/Interface management – planning communication protocols and data formats between the interfaces
- Data Catalog – a database of metadata, describes the data within the platform, and allows parties to discover what data is available, which properties and data types it has
- Data Quality – keeps track of metadata about last update times of each data source, define criteria for quality, alert when data isn't up-to-date, or is of low quality, etc.
- Data lifecycle – metadata about stage of each data element in the processing & storage, e.g., does it need to be processed, stored, archived, deleted, etc.
- Standardization Rules – a set of customizable rules for enforcing organizational standards of data (e.g., validation, formats, business rules, security rules)

Status:

Some of the building blocks are already available:

Figure 4: Status of Data Governance components in GWTF Platform

- Data security is implemented via the storage platform (MongoDB[2])
- Communication is implemented via Akka HTTP[3] (for RESTful services) and Ngnix[4]

The remaining building blocks are not available or partially available, meaning the technology is in place but there need to be some additional steps for integration and/or development, as can be seen in Figure 4.

[2] MongoDB is described in section 4.1.2
[3] Akka HTTP is described in sections 4.2.2 and 4.3.14.2.2
[4] Ngnix is described in section 5.1

3.2.7 General activities

In addition to the above components, the blueprint has a set of components whose responsibilities evolve around the operation and management of the platform.

This includes the following:

- Authentication and authorization
- Platform automation – automating operations and tasks, automatically ensuring the system can scale up or down based on level of use
- Management and Operation – the responsibility of day-to-day support and management, maintaining interfaces with data sources, status page for users
- Compliance and legislation – auditing & monitoring based on compliance and legislation requirements (e.g. GDPR)
- Infrastructure and connectivity – the infrastructure on which the platform is running on (e.g. Google Cloud, AWS, Azure)
- Security – maintaining and monitoring security requirements, developing procedures and periodic security tests

Status:

- Authentication and Authorization

  Implemented using an open-source platform (Keycloak).
- Infrastructure and connectivity

  Currently using Google Cloud Platform
- Platform automation – components in the architecture are deployed in Docker components, which make it easy to transition to automation
- Compliance, Legislation and security

  Google Cloud Platform already has partial built-in support for these aspects, but more work needs to be done to customize and adapt to the GWTF platform

3.3 The relation to Spatiotemporal Agribusiness Framework (SAF) platform

The Spatiotemporal Agribusiness Framework (SAF) is a framework developed by Milan Innovincy B.V. for collecting and processing Geo-based and time-series data related to the agriculture domain. It is designed to support Big data geo datasets, support for various crop models and machine-learning models with batch and stream processing.

The GWTF platform was designed with a similar architecture as the SAF, while reusing several infrastructure libraries and components for:

- Data Ingestion
- Processing pixel and plot-based data
- Data Exposure
- Storage
- Authentication and authorization
- Platform automation
- Common data types: geographic entities, domain entities (e.g., plots, famers)

4 GWTF Platform Architecture in detail

4.1 Data and Storage

4.1.1 Data stored in the system

The GWTF platform currently stores three main types of data:

1. Plot analytics – time series variables about plots (e.g., precipitation, soil moisture, water stress). This includes both calculated data and feedback from users (currently irrigation amounts and crop status).
2. Pixel analytics – time series variables applied to a geographical grid (similar variables as for the plots)
3. Non-temporal data – information that relates to the analytical data. Currently it is only the collections of users and plots.

The full list of variables of the plot/pixel data can be found in the Appendix 6.1

4.1.2 Storage

Data in the platform is stored in a database called MongoDB. MongoDB is a document database, and as such is a NoSQL database. In other words, data is not stored in relational tables as with traditional SQL databases.
Instead, in MongoDB the data is organized in a hierarchical structure, with a flexible schema.

In MongoDB the equivalent of a table row is a document, which in itself may contain multiple fields. Documents are organized in collections, which are similar to relational tables in concept. Unlike relational databases, documents can be nested to contain embedded documents, and fields are not limited to a specific set.

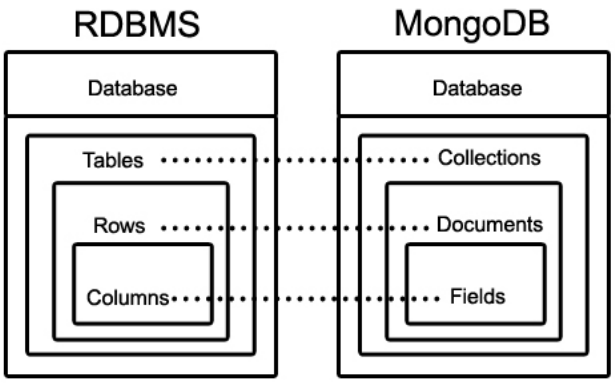


Figure 5: Concepts of RDBMS vs MongoDB

There are several advantages when using MongoDB for this platform:

- Flexible schema – the fact that documents do not adhere to a specific schema is an asset when handling real-world data, as well as in handling changes in requirements.
- Scalability – MongoDB is designed for easy horizontal scale up, with built-in sharding and real-time replication.
- Native Geo-location support – GIS data (like points, lines and polygons) can be stored semantically within documents and indexed, so Geo-based queries can be made natively and fast (e.g., one can query all documents that have points contained within a given polygon or query all documents that have points within a given distance to a given point).
- GridFS – MongoDB has a feature called GridFS (Grid File System) that makes it fast and easy to store large binary files. GridFS is currently used to store the pixel data.

4.2 Data Flows

4.2.1 From FEWS to the GWTF Platform

4.2.1.1 Operation scripts

FEWS exports model output nightly in the form of NetCDF files, which are placed in an FTP server. Each waterboard's data is put in a different directory.
The GWTF platform has an ingest command that can ingest that data for a particular date (or range of dates) and a particular waterboard (or a collection of waterboards).
The command parameters are:

--date	A date for which to ingest.
--date-range	A range of dates for which to ingest (colon delimited)
--waterboards	One or more waterboard names to ingest (comma delimited)

When running with –date, one can provide an absolute date in the format of YYYY-MM-DD or a relative date, e.g., -3 (which means "three days ago").

There are two scripts available in the production server (also available in the project source code repository), to make it easier to run the ingestion manually when needed:

run-ingest.sh <date> <waterbards>
Runs the ingestion for a particular date and waterboards

run-ingest-range.sh <from-date> <to-date>
Runs the ingestion for the given date range for Aa en Maas and Vallei en Veluwe (currently hard coded within the script).

Running the ingestion manually is useful for various cases, such as re-running the process or troubleshooting, but more important is the requirement to run it automatically every night.
For this purpose, we use https://systemd.io which is a modern suite of tools for Linux systems, that can do scheduling and is a more contemporary replacement of the Linux cron scheduler.
The production machine has a 24ystem service that is configured to call

```
run-ingest.sh -1 AaenMaas,ValleienVeluwe
```

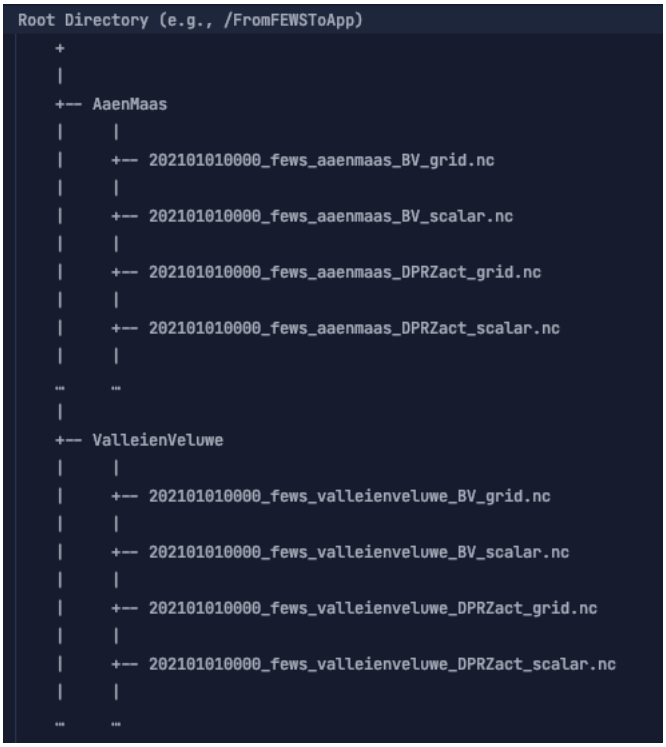
every night (currently at 2am UTC).

The ingest command, regardless of the parameters passed to it, starts a new process that performs the ingestion, processing, and storage in the database. This process will be further detailed in following sections.

4.2.1.2 FTP structure

As mentioned before, the data from FEWS reside in an FTP server.

The directories and files' structure are as follows:



An FTP server may contain data from more than one waterboard. Under the root there is a directory for each waterboard.
The dataset for each waterboard is placed in that directory in a flat structure. The intention is to keep this as a temporary location until it is processed by the platform, and periodically delete older files over time. This is done by the modeling platform, outside the scope of the GWTF platform.

In general, the naming convention of each file is:

```
<timestamp>_<model>_<RWA>_<variable>_<type>.nc
```

Where:

timestamp – the analysis date-time, i.e., the timestamp in which the model performed the analysis. The format is YYYYMMDDHHMM

Model – can be either ***fews*** (for the regional model) or ***swap*** (for the local model).

RWA – the regional water authority (a.k.a. waterboard)

Variable – the variable that is described in the file. E.g., P_Input describes precipitation, and Etact describes the actual evapotranspiration.

Type – can be either:

1. scalar – data about individual plots (i.e., single value for each plot ID)
2. grid – the geographical location of the RWA is divided into a grid. Data is provided per pixel in the grid.

Currently the ingest command can only fetch data from a single FTP server, but this can be enhanced once there's a need for multiple servers, by adding another set of parameters to this command.

Side note: Pros and cons of using FTP as the form of data transfer between the model and GWTF Platform

Using FTP as the form to transfer data between the platforms was chosen because it was a native way for the modeling team to export data and was therefore the quickest way to start up.

However, there are some disadvantages in using FTP for this purpose:

1. It is less resilient and robust than some of the modern API protocols
2. It is more difficult to communicate meta-data (need to use file name conventions)
3. FTP is not optimal for such use cases. E.g.: a difficult to understand issue occurred when the FTP server and client were on different time zones, and due to a daylight-saving clock change, dates on the FTP server were not able to be recognized by the FTP client, and the transfer failed.
4. There is currently no easy way for the GWTF platform to signal to the modeling team that a file has been ingested and processed, and therefore can be deleted from the FTP.
5. Entire files need to be downloaded prior to processing, whereas with an API processing can potentially be streamed and processed more efficiently.

An alternative to FTP can be RESTful APIs over HTTP, which provide more flexibility, robustness and efficiency.
Note: In the opposite direction (where GWTF platform provides data to the local model), data is already transferred via a RESTful API.

4.2.1.3 NetCDF File structure

Each NetCDF file generated by the modeling platform contains a time dimension that may span over multiple dates, both historical data (past dates) and forecast data (future dates).
As previously mentioned, there are two types of NetCDF files (Scalar and Grid), each has a slightly different structure.

Scalar files

Scalar files contain data of a single model variable about multiple plots over time. Typically, one file will contain data for all plots of all registered users of a single Regional Water Authority (RWA). However, the processing algorithm is designed to handle multiple scalar files per variable and RWA, in case a single file becomes too large for processing.

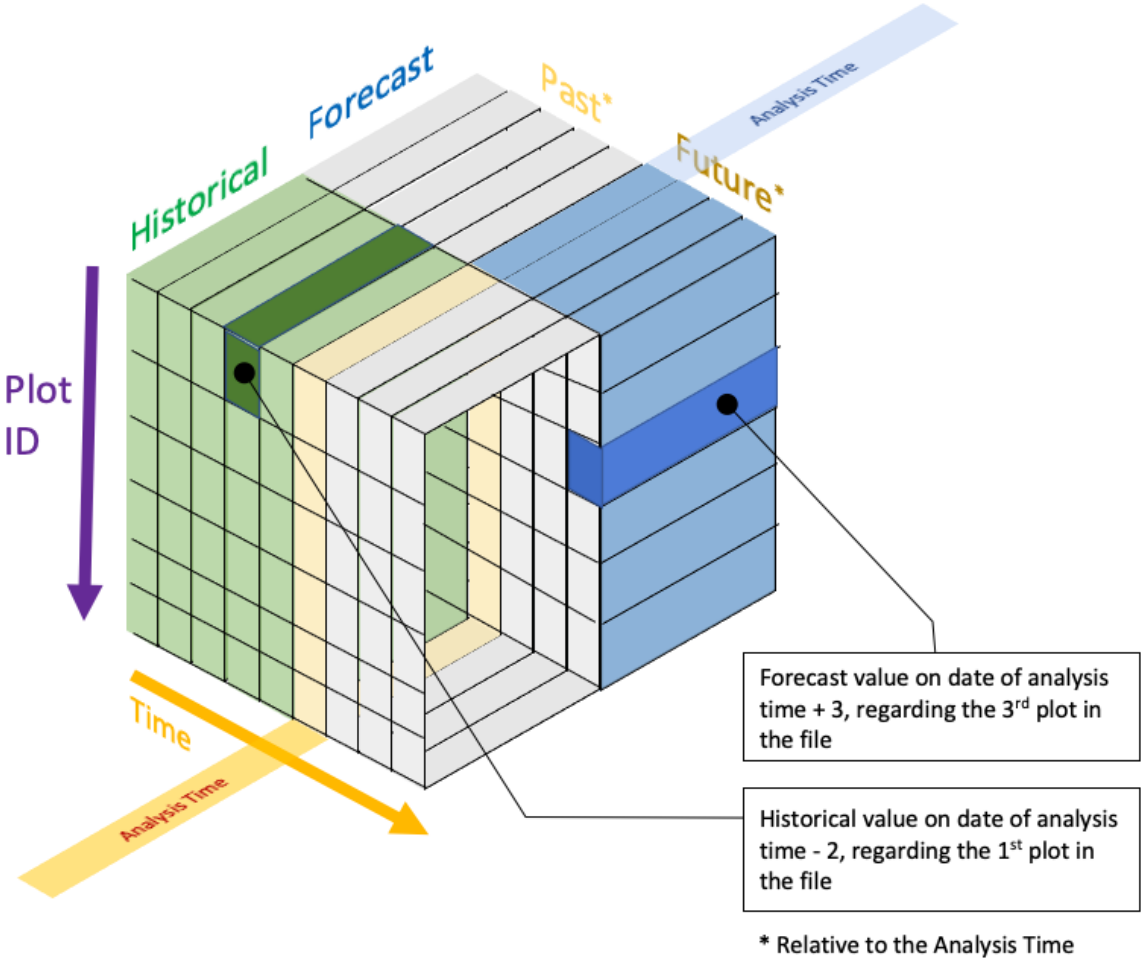


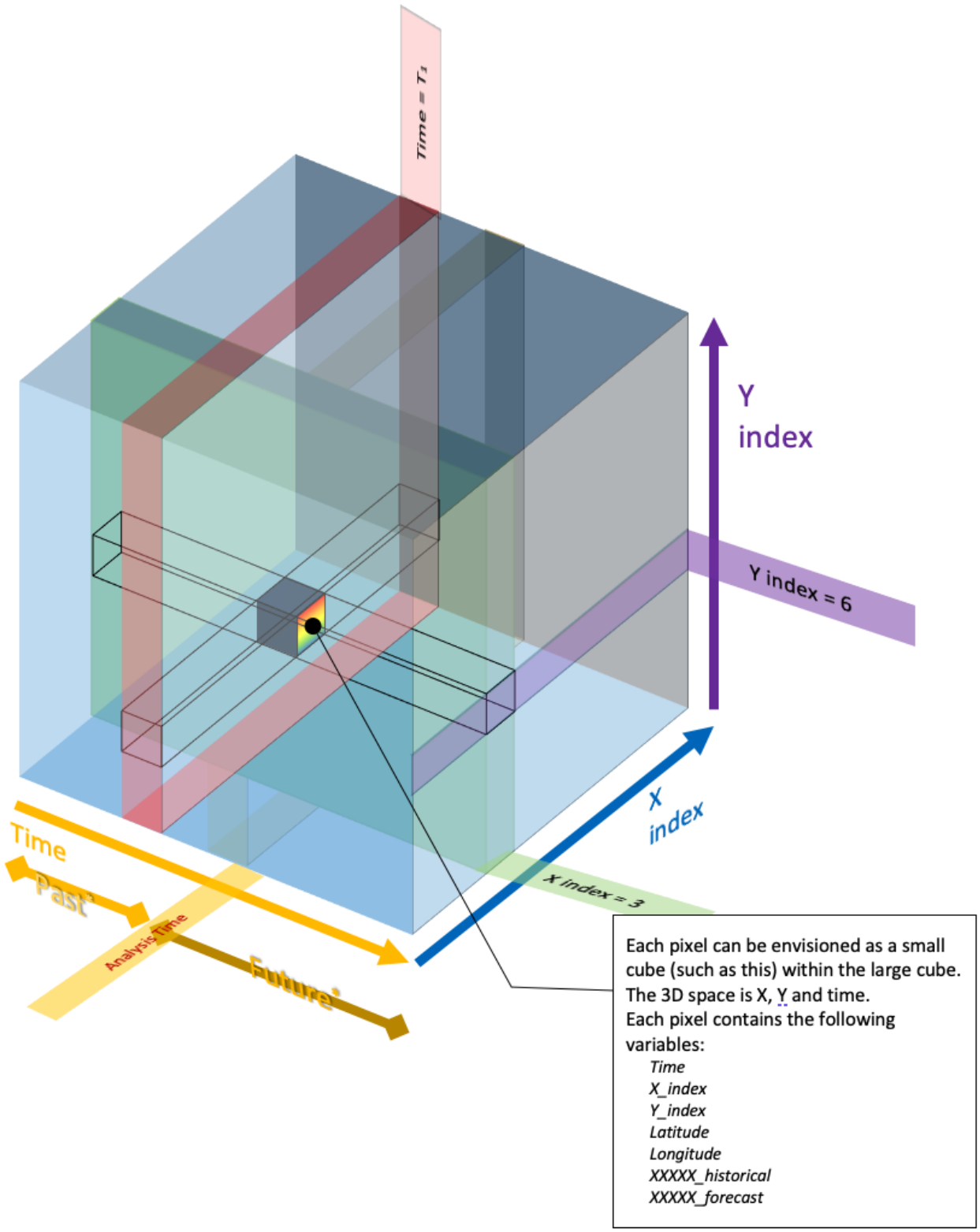
Figure 6: Scalar file structure

The structure of the file can be viewed as a 2.5 dimensions cube, with: time, plot ID and 2 variables for each: historical and forecast. E.g., for precipitation, the file would contain a P_Input__historical variable and P_Input_forecast variables. The file contains two values (both historical and forecast) for each timestamp and plot ID. However, there is a valid value in P_Input_historical only for timestamps lower than the analysis time (also provided as a variable in the file). Likewise there is a valid value in P_Input_forecast only for timestamps higher or equal to the analysis time.

Grid files

Grid files contain similar data as the scalar files, but the data is related to each pixel in the grid rather than a plot. Grids span over a geographical area, where each cell has a fixed size.

The structure of the file has 3.5 dimensions: time, longitude, latitude, and then historical and forecast variables for each 3-dimensional cell – similar as the way it is structured in the scalar files. Grid files also have an analysis_time variable to indicate when it was generated, and what is the boundary of historical vs. forecast data.


Time = T1
Y
index
Y index = 6
X
index
X index = 3
Time
Past
Analysis Time
Future
Each pixel can be envisioned as a small cube (such as this) within the large cube.
The 3D space is X, Y and time.
Each pixel contains the following variables:
Time
X_index
Y_index
Latitude
Longitude
XXXXX_historical
XXXXX_forecast

4.2.1.4 Ingestion

The method used for ingestion is *Batch Pull*. That's due to the nature of the FTP protocol.
Pull – because the GWTF platform is a client of the FTP server.
Batch – because the FTP serves distinct files that can be downloaded on demand (in our case, daily).

As previously mentioned, the ingestion command receives a list of waterboards and a date or a range of dates as input.
Then for each waterboard, the ingest command then requests the list of all files under the waterboard's directory. It then filters the list of files based on the date or range of dates in the input.

The files that pass the filter are downloaded from the FTP server to a temporary directory on a disk of the GWTF platform's ingest service.
Each of these files that are downloaded are then passed on for processing.

4.2.1.5 Processing the plot data (Scalar type files)

As described in previous sections, for each waterboard and analysis time there is a set of scalar files, each file describes a different variable.
For example, for analysis time of 2021-01-01 in the Aa en Maas waterboard, these would be all files of the form

`202101010000_fews_aaenmaas_`**`<VarName>`**`_scalar.nc`

All the files of that form are collected, parsed, and converted into a structure of the form:

```
Table[PlotId, Table[Timestamp, Table[VarName, Value]]]
```

Where *Table[K,V]* is a data structure that maps unique keys K to a value V.

Following that some validations are made to ensure that the data is indeed consistent as expected, e.g., that all the files in that dataset contain the same analysis time. When that is not the case, it is difficult to reason what is historic vs. forecast data and processing will stop with an error.

Then this structure is converted into a collection of PlotAnalytics objects, which has the following schema:

<table>
<tr><th>Name</th><th>Data type</th></tr>
<tr><td>_id</td><td>ObjectId</td></tr>
<tr><td>Timestamp</td><td>Long</td></tr>
<tr><td>isForecast</td><td>Boolean</td></tr>
<tr><td>Waterboard</td><td>String</td></tr>
<tr><td>localPlotId</td><td>String</td></tr>
<tr><td>attributeValues</td><td>
<table>
<tr><td>$Name_1$</td><td>$Value_1$</td></tr>
<tr><td>$Name_2$</td><td>$Value_2$</td></tr>
<tr><td>…</td><td>…</td></tr>
</table>
</td></tr>
<tr><td>analysisDate</td><td>ISODate</td></tr>
<tr><td>updatedAt</td><td>ISODate</td></tr>
</table>

These objects are stored in the database with an upsert operation. “Upsert” is a combination of insert and update, meaning – the object is inserted if it doesn’t exist, or updated if it already exists. “Existance” is determined based on some key, either the unique ID of the object or some other attribute (or combination of attributes). The benefit of upsert is that it can be performed in a single database operation.

A PlotAnalytics object exists in the database if there’s already a document with the same timestamp and plot ID. This should be intuitive, given that a single plot cannot have more than one analytics object in each point in time.

In other words, using this mechanism the GWTF platform ensures that in case the Modeling Platform outputs more accurate data for a given plot on a later date, the platform will store and expose the most recent data. The way it works is depicted in the diagram below.

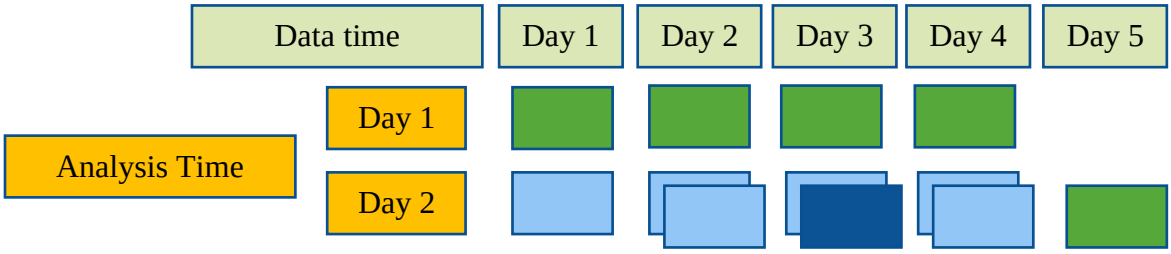


Figure 7: insertion and update of plot analytics

For simplicity of the diagram we will assume that the NetCDF files contain only 4 days of data. On Day 1, the platform gets the first 4 days from the Modeling Platform of a plot P_1. All of them are inserted since the database didn't previously have any data about Day 1 through 4 for P_1. On Day 2, the platform gets the data about day 2, 3, 4 and 5. Days 2 – 4 already exist and so they are updated. In practice, Day 2 and 4 contain the same data as it was received on the previous day, so effectively nothing happens. Day 3 is different in some values, and the latest data is stored. Day 5 is new, so it is inserted.

Note that the attribute values are stored as-is, without any transformations. Transformations such as unit conversions are applied upon request of those variables by the API.

4.2.1.6 Processing the grid data

Unlike plot analytics, which are stored in a document per plot and timestamp, grids are significantly larger and therefore stored as grid files with only the metadata stored in MongoDB documents that are used for performing queries.

The mechanism for storing the files is by GridFS, a feature in MongoDB which is a way to store large datasets.

With GridFS, each file being stored is like a single BLOB that you can write or read, but there's also a regular MongoDB document that is linked to it, where one can store any type of metadata. That metadata is used for queries. With this mechanism you can make a single query and retrieve all the BLOBs at once.
More information about GridFS can be found here.

We store the following meta data for each grid file:

- waterAuthority – the RWA for which this grid is related.
- analysisDate – the date in which the data was analyzed by the Modeling Platform
- date – the date for which this grid relates to
- isForecast – whether this grid contains historic data or a forecast
- attribute – what variable this grid is describing (e.g., P_Input, DVS)
- boundingBox – the WGS-84 coordinates of the rectangle in which this grid is located
- dimensions – the dimensions of this grid as a pair: [width, height]

The figure below shows an example of the metadata linked to one of the grids.

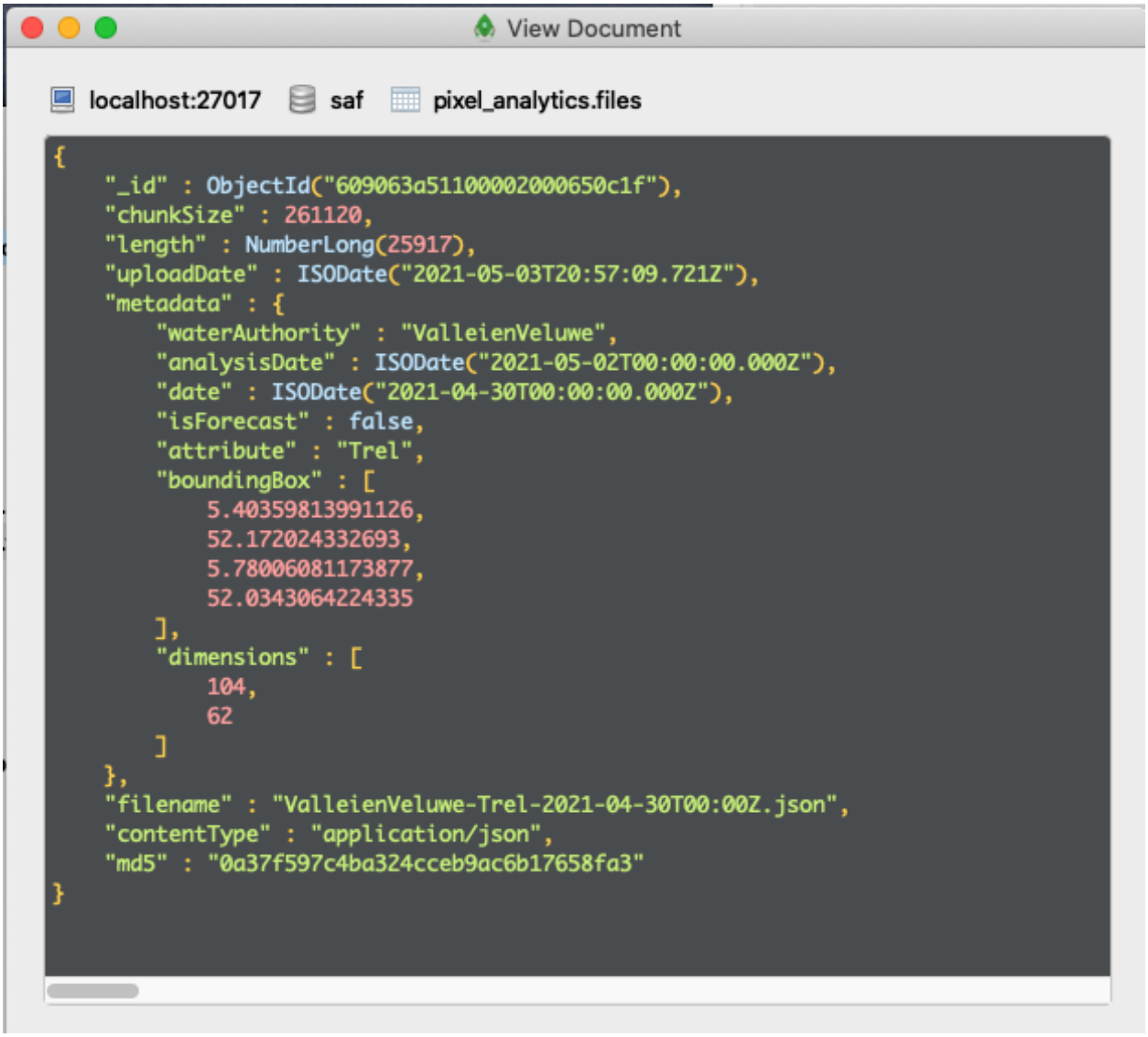


Figure 8: JSON representation of a MongoDB document that contains the metadata of one grid file. The grid file contains the Trel attribute for Vallei en Veluwe, on 30th April 2021, as analyzed on 2nd May 2021

The rest of the attributes that can be seen in Figure 8 (namely _id, chunkSize, length, uploadDate, filename, contentType and md5) are automatically maintained by MongoDB.

The grid itself goes through two types of transformations:

1. **Value transformations**
    a. Models may expose a special “empty” value, which means that there is no actual value available. In FEWS, this value is typically -999.0. The first transformation

in the GWTF platform converts this value to zero. Meaning, if the model cannot provide for some reason an actual value, the app will show 0.
Note that there is room for improvement here, in which the platform will also represent "empty" values in some way (either leave the -999.0 or use some other more generic representation supported by the database, such as NaN or null). Clients of the GWTF platform could then visualize such values differently (say, with dotted lines in graphs or a different color).

b. Each value in each cell is converted to the default unit as desired in the GWTF app. For example, the water deficit Ssdtot, is converted from millimeters to meters. To facilitate this, the GWTF platform has a catalog that lists all model parameters. The complete list of parameters and their units in the model vs. the unit in the app can be found in the appendix 6.1
c. All negative values are converted to 0.

2. **Coordinates Transformation** – The bounding box of the grid is converted into WGS-84 so that it is correctly displayed in mapping software.

3. **JSON transformation** – the grid is then transformed into standard JSON array format. This is done because JSON is a standard de-facto web format, which is the most convenient for use by the app. In this way, the API retrieves the file and serves it as-is, without further need of processing.

Figure 7: insertion and update of plot analytics

These transformations are done at the processing stage for convenience, because the API can then simply stream the files directly in the response without parsing the file, and without applying further processing upon request.

The disadvantage of this approach is that some transformations cannot be reversed (1a and 1c) so the raw data is lost.

A future improvement could be to move those transformations to the API level (as it is done with the plot analytics), which will ensure allow for more flexibility. However, for the current stage it is not a priority.

Following the transformations, the grid (in JSON format) is then stored as a file in GridFS (one file for each waterboard, date and parameter), together with the meta-data.
Similar to the plot analytics, the file is inserted if no file already exists for that waterboard/date/parameter, or replaces any existing one (see Figure 7: insertion and update of plot analytics)

4.2.1.7 Processing the local model

The local model contains data about plots only, and the file structure is exactly the same as the regional model's scalar files.
Unlike the regional model, the local model doesn't have one file per day per variable, but rather there is a file per variable, and data is accumulated in each file for the entire timeline, starting 1st January 2021, until the analysis time + forecast period. Each day the same files are re-written with updated results and an additional day of forecast
Therefore, the ingestion command is the same as with the regional model, but with different parameters:

- The FTP root directory is `/FromLocalmodelToApp`
- The date: `2021-01-01`, rather than `-1` ("yesterday") as with the regional model.

Consequently, with the local model the GWTF platform reads and updates the results starting from 1st January 2021.

4.2.1.8 Orchestration of the ingestion

The automation is run by 37ystem.io, as explained above. It runs both the regional and local model ingestion serially.
The daily ingest runs the regional ingest at first, and only then the local model ingest. This way the local model results (which should be more accurate) override the regional plot-level model results.

4.2.2 The GWTF Platform API

The API of the platform is the way in which it exposes data to clients, as well as a one of the ways to ingest data from clients. Clients can be the GWTF application, as well as other application or external parties.
Because the ingestion from the Modeling platform is currently based on FTP and not a RESTful API, this section doesn't cover that here (see section 4.2.1.4).

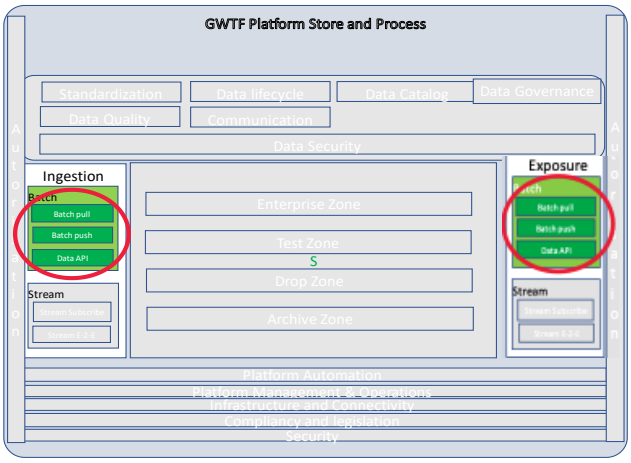


Figure 9: The API within the platform's architecture.
The API is used for Ingestion and Exposure

Technically, the API is deployed as a service called *webapi* which implements all API endpoints currently available (both of ingestion and exposure). To implement this, Webapi uses an open-source software library called Akka HTTP which makes it extremely easy to build HTTP request handlers (called *Routers*). Akka HTTP handles all the parsing and intricates of the HTTP protocol, giving the developer the freedom to focus on the business logic. It provides support for virtually all HTTP protocol features, as well as handling RESTful API payloads in JSON format (as well as other formats).

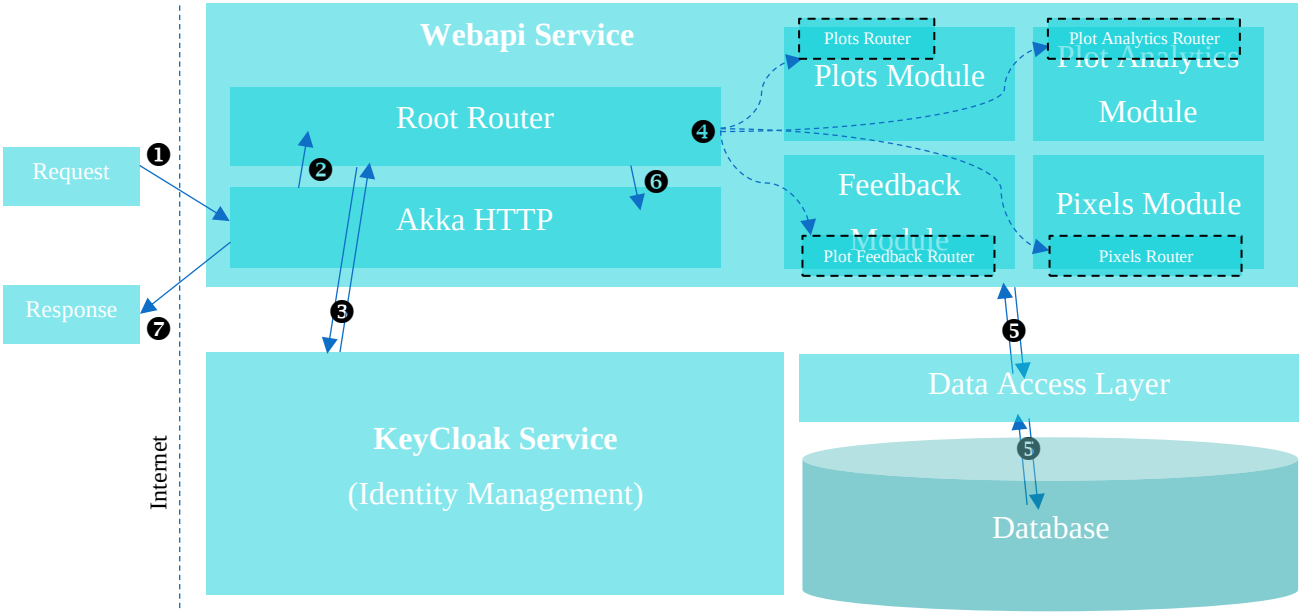


Figure 10: Data flow of the API

The HTTP request handling is done based on the URL and method of the request (GET, POST, DELETE, and PUT) ❶. Webapi has a root router which gets all HTTP requests ❷

The HTTP request is expected to bear a header containing an encrypted authorization key. This key is verified in the KeyCloak service ❸

If the key is invalid or expired, then the API response ❼is 401 (Unauthorized) – which instructs the user of the API to get a new authorization token.

If the key is valid, then KeyCloak provides the root router with the identity of the API caller. The root router then forwards the request to a service router ❹ based on the URL prefix. E.g., if the URL starts with /plots then the request is forwarded to the "plots" router. The plots router in turn, knows how to handle the specific request, based on the HTTP method and URL. It then activates the relevant business logic within the respective module, and data access layer ❺ in order to retrieve the relevant data.

For example, if the request was GET /plots, then the plots router will simply request the list of plots that the authenticated user is allowed to receive.

Whatever the result of the API call might be, it is passed back to calling router, which is responsible to transform it to the format required by the API (in most cases, JSON). For more information about the API specification see 6.2.

That response payload is passed back to the root router, and then to Akka HTTP ❻ which responds in standard HTTP protocol ❼

This separation of a root router, multiple modules and a router for each module makes it easier to decouple different services and modules from each other. As the platform grows over time, we may decouple the modules even further using more granular modules or possibly with microservices.

4.2.2.1 GET requests

The following APIs are relatively simple GET requests:

- /me – Gets the user information of the authenticated user (name, waterboard, user type)
- /plots – Gets the list of plots visible to the authenticated user
- /plot-analytics – Gets model information about the plots of the authenticated user, over a given time range
- /pixels – Gets grid-level data for the waterboard of the authenticated user

4.2.2.2 Farmer feedback

The purpose of the feedback API is to make it possible to receive information from farmers (namely, ground truth) that can help calibrate the models.
The feedback API has PUT methods, in order to send information from clients to the platform, and GET methods, so that clients can check what was previously sent. GWTF platform serves as the single source of truth of farmer feedback, and clients are not required to store this information permanently).
Currently there are two types of information that can be collected from farmers:

- Irrigation data – how much water was irrigated (in millimeters) on a specific date in a particular plot. The irrigation amount can be any positive integer. There is currently no validation on the platform side to ensure that the value is not too large or small.

- Crop status – what is the development stage of the crop growing on a particular plot. The API accepts a string value for the status, and there is currently no validation or check that on the platform side that the value is a valid status, or that it matches the crop type being grown on that plot. It is therefore currently up to the client-side to validate and ensure the correctness of these values.

Irrigation data

Storing and receiving the irrigation data is straightforward.
The PUT API gets the plot ID, the irrigation date and the amount, and stores it in the database in the following schema in the plot_feedback collection:

Name	*Data type*	*Description*
_id	ObjectId	The unique ID of the entry
type	String	A discriminator field for the type of feedback. Holds "irrigation"
plotId	ObjectId	Refers to the plot in the GWTF platform database
date	ISODate	The date in which the plot was irrigated
irrigationMM	Int32	The amount irrigated
updatedAt	ISODate	The date in which this entry was created by the user
updatingUser	String	The username that created this entry

The GET API gets a range of dates as input parameters. It returns all the irrigation records that are stored in the database, for all the plots of the authenticated user, within the given date range. The API also has two additional and optional parameters:

1. waterboard – if the authenticated user has access to more than one waterboard, providing this parameter will limit the results to plots within the given waterboard. The local model uses this API to retrieve irrigation data for each waterboard at a time.
2. withLastUpdateBeforeStartOfRange – if this parameter is true, then the result will include the last irrigation entry whose date is **before** the given date range (if such exists).

This result is placed as the first one in the list. The GWTF application uses this information to indicate when was the last time a plot's irrigation data was updated.

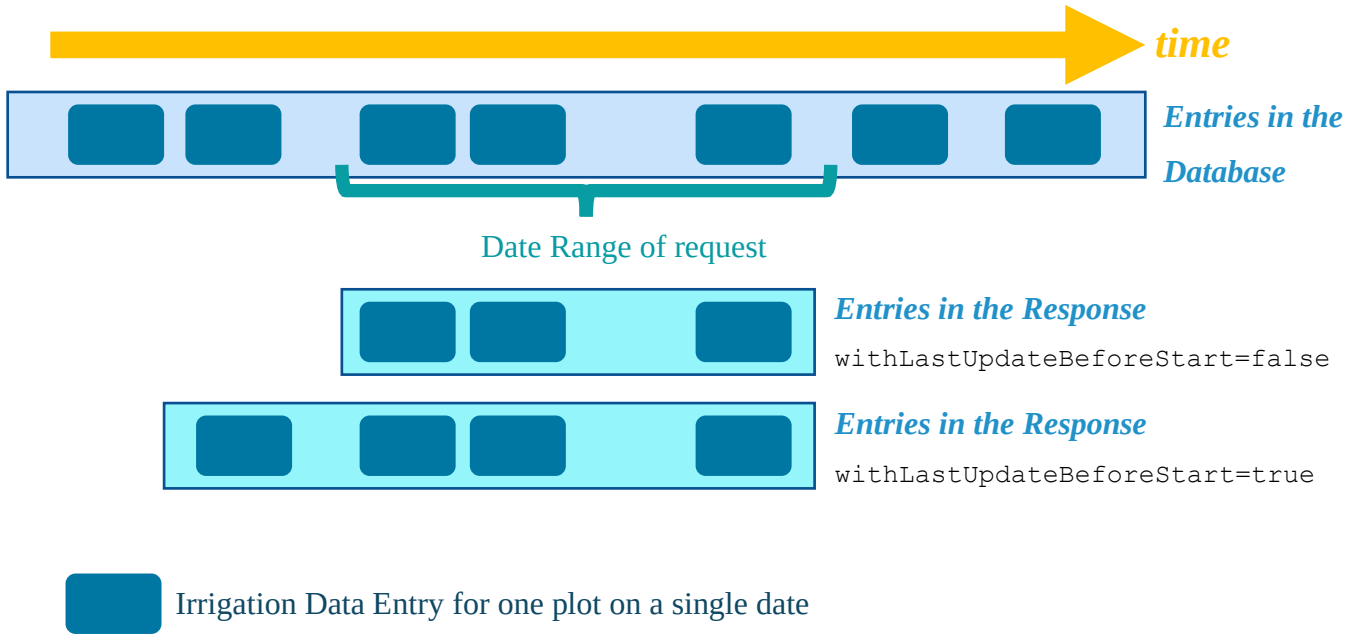


Figure 11: Irrigation data GET API

Crop status

The PUT operation for crop status is very similar to that of irrigation.

The API gets the plot ID, the date for which the new status is applicable. For example, if the farmer noticed that germination first happened on April 3rd – this would be the date (even if the actual feedback was received on, say, April 10th.)

This data is stored in the database in the following schema, in the plot_feedback collection:

Name	*Data type*	*Description*
_id	ObjectId	The unique ID of the entry
Type	String	A discriminator field for the type of feedback. Holds "crop-status"
plotId	ObjectId	Refers to the plot in the GWTF platform database
Date	ISODate	The date in which the plot was irrigated
cropStatus	String	The crop status
updatedAt	ISODate	The date in which this entry was created by the user
updatingUser	String	The username that created this entry

The GET API also works quite similarly to irrigation data. It requires a range of dates as input, and has two optional parameters: waterboard and withLastUpdateBeforeStartOfRange. The functionality and implementation is the same as with irrigation.

4.2.3 API documentation

The API is likely to evolve and change over time, probably faster than this document. The API documentation was developed in Swagger (an online collaboration tool and language for RESTful APIs). For the latest version see 6.2

4.3 Software, Open-source software and libraries

The GWTF platform is written in the Scala language and compiled into standard Java Virtual Machine (JVM) bytecode.

There are several open-source software libraries currently being used, detailed below.

4.3.1 Akka

Akka (https://akka.io) is a toolkit for building highly concurrent, distributed, and resilient message-driven applications for Java and Scala.

On top of the core Akka library there is Akka HTTP, which is used by the platform to implement the RESTful API (see 4.2.2 – The GWTF Platform API)

Akka is well-proven in production, and used by several well-known companies such as PayPal, LinkedIn, Shopify, and Tesla.

Current version used:

Akka 2.6

Akka HTTP 10.2

4.3.2 GeoTools

GeoTools (https://www.geotools.org) is an open-source Java library that provides a wide range of tools for geospatial data. Some notable examples are:

- Reading/writing GIS data in many file formats and spatial databases
- Performing operations on topological shapes, such as filtering, intersecting, and unifying.
- Applying shapes on raster data (e.g., cropping an image based on a polygon).
- Coordinate reference system and transformation support

Current version used: 24.1

4.3.3 NetCDF Java

NetCDF Java (https://www.unidata.ucar.edu/software/netcdf/) is a library created by UCAR (https://www.ucar.edu), whose purpose is to read and write files in NetCDF format. This library is used by the GWTF platform to read NetCDF files provided by the Modeling Platform.

Current version used: 4.6

4.3.4 Keycloak

Keycloak (https://www.keycloak.org) is an open-source identity and access management tool. Out of the box it can store users, authenticate and authorize. It also has advanced features like user federation, identity brokering and social login.

Current version used: 11.0

4.3.5 ReactiveMongo

ReactiveMongo (http://reactivemongo.org) is a Scala driver for MongoDB that provides fully non-blocking asynchronous I/O operations. This works well with the reactive programing architecture of the system.

Current version used: 1.0.3

4.3.6 Logback

Logback (http://logback.qos.ch) is a logging library for Java and JVM applications. It supports a wide range of logging policies, dynamically change the logging level and provides high performance.

Current version used: 1.2.3

5 Cloud infrastructure, configuration, and deployment

The GWTF platform is designed to operate in a cloud environment for easier deployment, scalability and maintenance. The services are dockerized (i.e., built as Docker containers), so each service is isolated from the others, and can be scaled up or down regardless of other services.

5.1 Docker

The build scripts of the GWTF platform automatically generates docker images for each service, currently there are two: ingest and webapi.
There are additional containers that are not automatically generated upon each build:

- Mongodb – the database (for development environments only)
- Keycloak – 3rd party identify management service
- Ngnix – a 3rd party web server that can also be used as a reverse proxy, load balancer and HTTP cache. In production it is used as a reverse proxy server that handles incoming HTTP requests, and redirects to the right service

Finally, there's a docker container for serving the front end. It is also based on an Ngnix image, and it simply serves the static resources of front-end (javascript files, HTML, CSS, images, etc.). This container is generated automatically by the front-end deployment script.

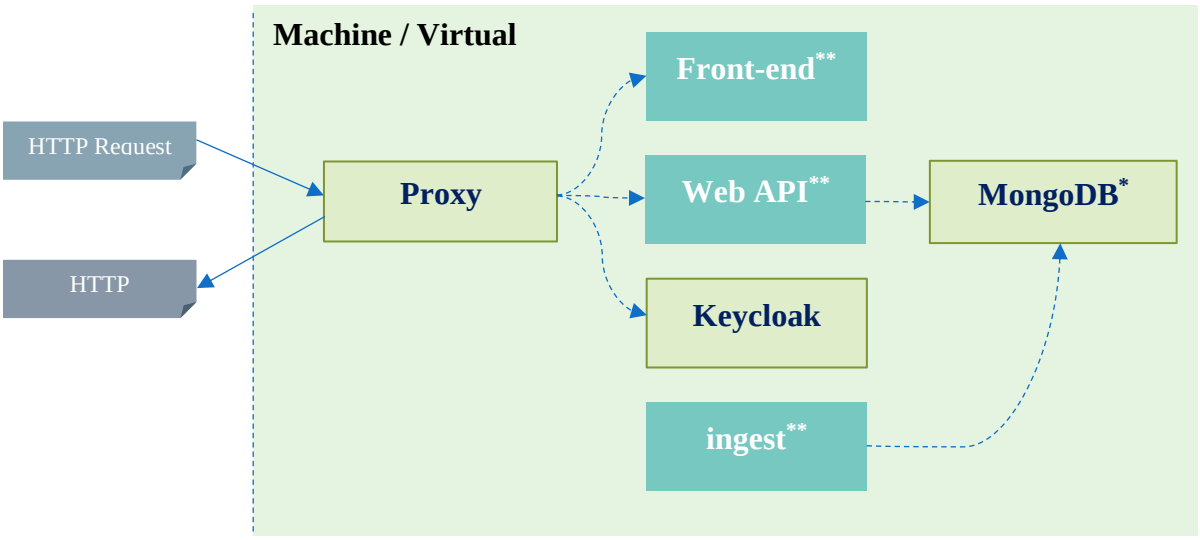


Figure 12: Docker containers

* The MongoDB database runs as a docker container only in development environments

** Containers that are re-created upon deployment of a new version

5.2 The production environment

The production environment of the GWTF platform resides in Google Cloud Platform, a suite of cloud computing services that runs on the same infrastructure that Google uses internally for its end-user products (such as Search, Gmail, Drive and YouTube). One of these services (Computing Engine) provide the ability to spawn virtual machines of many types of server operating systems. One of them is a Container Optimized OS (built by google) which is a variant of Linux especially built for running Docker containers.
At present we only have one such virtual machine running, whose specs are detailed in 6.3.1
The data center is physically located in Eemshaven, Netherlands.

With the Google Cloud platform the specs of the machine can be easily modified via a console user-interface and via API. It is also possible to spawn additional virtual machines as needed.

The database is hosted in MongoDB Atlas, a cloud database service. Under the hood, the machines on which the database resides are also on Google Cloud Platform. The Atlas service

ensures high availability, performance optimization, security and backups. The database is a cluster (in MongoDB this is called a “Replica Set”) of 3 nodes (1 primary and 2 secondary) that ensures high availability.

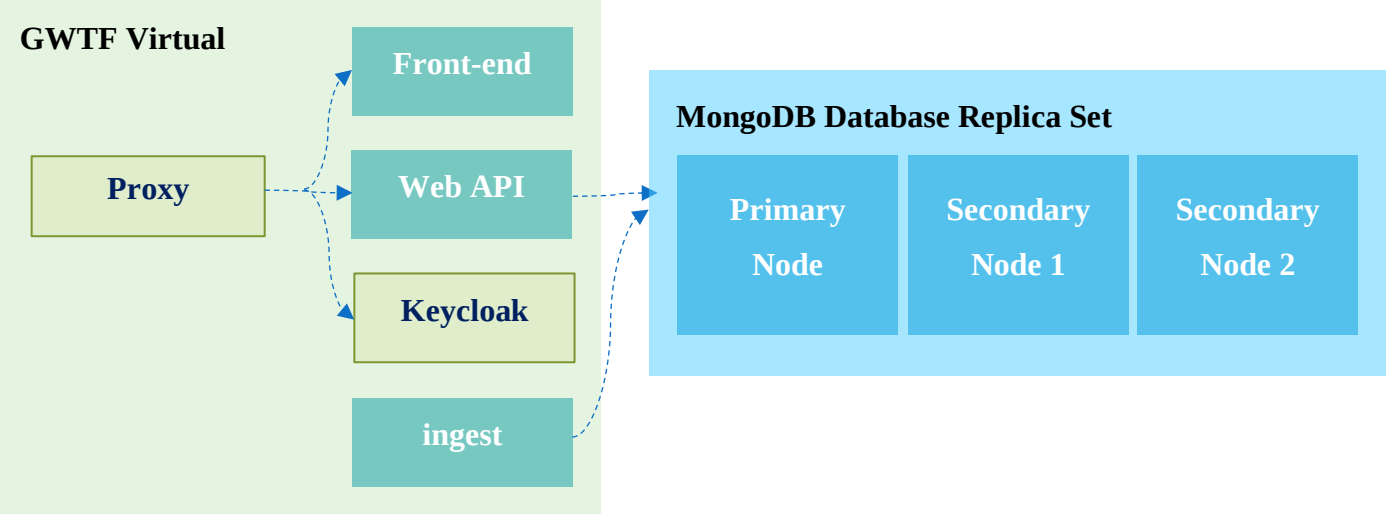


Figure 13: The production environment

The database specs are detailed in 0

Note that the front-end is not considered part of the GWTF platform. It is a web client of the platform. It is deployed in the same production environment for simplifying maintenance and operation, and for saving resources, but the codebase is completely decoupled.
In the future it is quite possible to deploy the front end in a separate machine, different cloud platform and/or under a different domain name, as needed.

The proxy of the GWTF platform is configured to route requests using the following table:

Table 1: Reverse proxy redirection table

Request URL Prefix	*Service*
api.irrigation.live	Web API
app1.irrigation.live	Front end
auth.irrigation.live	Keycloak

5.3 Database Disk utilization

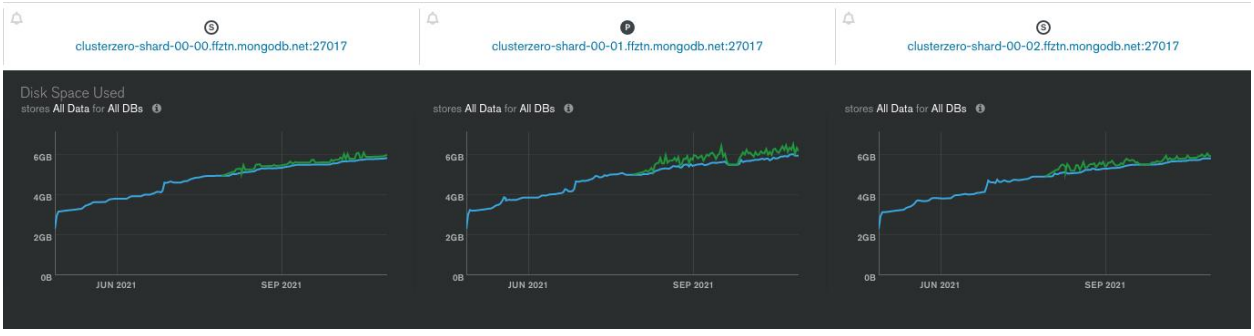
The database is a cluster of three servers, primary, and two secondary servers.

The database has 5 collections (a collection is similar in concept to a table in a relational database):

Collection name	**Current size**
plots	120 MB
plot_owners	0.0015 MB
plot_analytics	100 MB
plot_feedback	0.05 MB
pixel_analytics	14,800 MB

This gives an idea about the proportions of each collection, and how much growth could affect the database size.

Below is a chart showing the growth in disk usage over time.



The current growth is roughly 1 GB per month overall, assuming the current scale is maintained. There are currently 35 variables in the regional model, therefore each variable contributes roughly 28 MB per month.
Adding additional waterboards will increase the growth in direct proportion of the number of pixels that it covers. V&V currently has 183x148 pixels, and Aa en Maas has 332x256 pixels. This means roughly 250 bytes per pixel per variable per month.

5.4 Disk utilization on the server

The production server has two permanent storage disks:
Disk 0 – used for operating system, installed software, docker containers and system configuration files (30 GB in size)
Disk 1 – used for GWTF platform log files as a temporary location for NetCDF files downloaded from the Modelling Platform (30 GB in size)

The use of Disk 0 shouldn't change significantly over time.
The use of Disk 1 may increase depending on several factors:

a. Number of days for which we keep the temporary data
b. Total number of plots per waterboard
c. Number of variables for each plot in the regional model
d. Total number of cells in the grids per waterboard
e. Number of variables for each grid cell
f. Total number of days in the local model
g. Total number of variables in the local model

To make a rough estimation of how much the disk space can grow, we can use the following notations.
For each waterboard *i*, let the average file size of a scalar NetCDF regional model file be S_i and the average file size of a grid NetCDF regional model file be G_i. *N* is the total number of waterboards.
With the local model, the NetCDF files grow each day indefinitely.

Let L_i be the file size of a scalar NetCDF local model file per day.

The amount of disk space would then be roughly:

$$Size = \sum_{i=0}^{N-1} [a(cS_i + eG_i) + fgL_i]$$

At present, the file sizes are as follows[5] (in bytes):

	Regional model Scalar (S_i)	*Regional model Grid (G_i)*	*Local model (L_i)*
Aa en Maas	95,864	14,287,048	1,889
Vallei en Veluwe	74,804	4,445,872	1,475

The regional model has 35 variables (c = 35, e = 35)

The local model has 20 variables (g = 20)

This means that the size of the temporary storage grows at about 650 MB per day with the current parameters.

With this growth rate, the policy is to delete the files every 30 days.

5.5 Deployment process

To deploy the docker containers described in the previous section, we have a script for each service (webapi, ingest and front-end) that does the following automatically:

1. Build the service. The result is a set of compiled classes and finally JAR files (archives that are loaded by Java Virtual Machines)
2. Create a local copy of a docker image for this service
3. Submit the image to a private docker repository on the Google Cloud Platform

[5] The size can vary a bit between different files but I've observed that the variation is usually not more than 1KB.

4. Run a script on the virtual machine that pulls the latest docker image of that service
5. In case of the Web API and front-end services there are always running containers. The script therefore needs to stop and remove the running container, create a new container based on the new image and start it.
6. In case of the ingest service, the container is created and run only when triggered by the scheduler, or upon request. Therefore, no further action is needed.

5.6 SSL and HTTPS in production

At present the GWTF platform accepts only plain HTTP. Adding a layer of encryption requires either:

1. obtaining and installing an SSL certificate and configuring the Ngnix reverse proxy to accept HTTPS requests
2. setting up a built-in service in Google Cloud Platform that can take over the reverse proxy and also supports HTTPS

Option 1 needs a bit more effort but unlike option 2, there is no lock-in to Google.
Option 2 means that there's slightly more work to do in case of a transition to a different cloud platform. However, the added lock-in is minor.
The option that will be used is TBD.

5.7 Monitoring the system

5.7.1 Monitoring the servers

The Google Cloud Platform has an extensive console that can help monitor the servers and their performance. It also allows to perform operations such as starting/stopping and making changes to the configuration of the servers (e.g., increasing RAM or disk space).
In addition to the console, the Google Cloud Platform has an API that can be used to integrate the operations and monitoring with other systems, as well as perform automation.
Moreover, there is an extensive command-line interface, that can also make automation much easier. The deployment scripts of GWTF make use of the command-line interface in order to

push new versions to the server and restart them. The documentation of the command line interface can be found here.

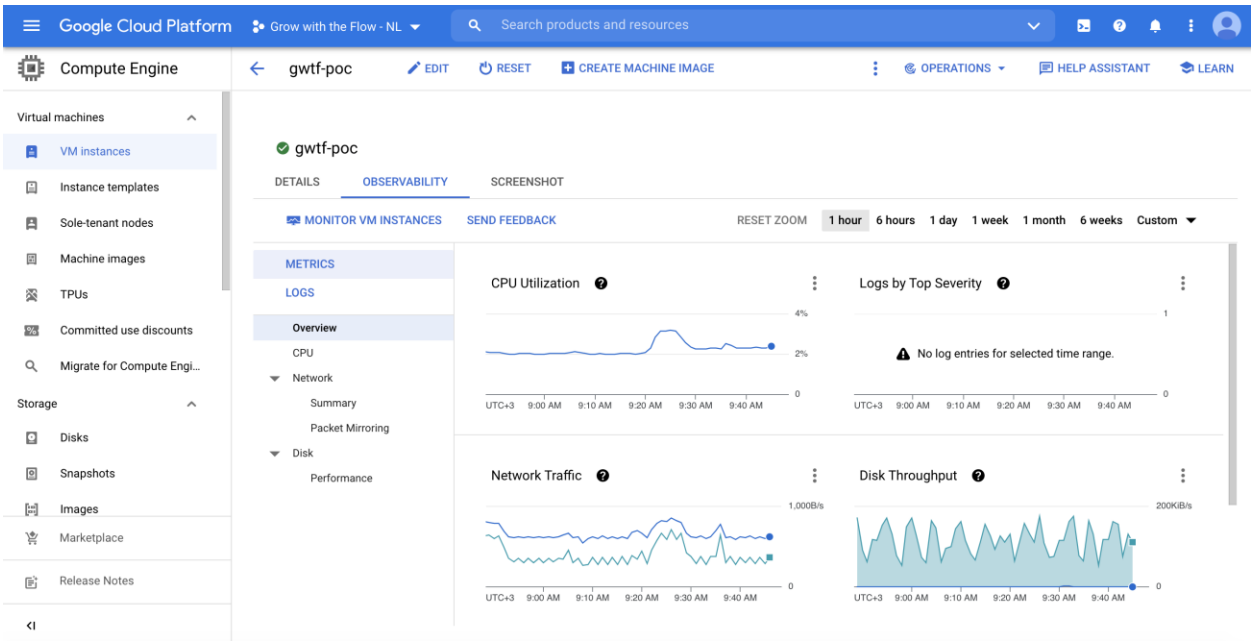


Figure 14: The Google Cloud Platform console: server monitoring dashboard

5.7.2 Log files

The GWTF platform has a logging mechanism with multiple levels of severity:

ERROR	Indicates a failure or an unexpected issue
WARN	Not an error but it is likely that the issue should be addressed so to avoid future errors
INFO	Informational log entries. Indicates that some expected event has occurred
DEBUG	Logs that can be used to troubleshoot issues
TRACE	Even lower-level entries than DEBUG, usually for debugging purposes

The logs are written to files on the server's disk, in a rolling file policy. Meaning, there is a file for each day, and each file is kept up to 30 days. In other words, logs are kept 30 days back. The ingest and webapi services keep separate log file sets.

Logback, the logging library used by GWTF, lets the user determine the desired logging level. Setting the logging level means that anything of that level and above it will be logged. For

example, if the log level is set to WARN, then both ERROR and WARN will be logged, but INFO, DEBUG and TRACE will not.

It is possible to change the logging level as needed.

By default, the logging is INFO.

In addition, it is possible to set the logging level at a more granular level, so it is possible to keep the default logging as INFO but set a certain component to log at DEBUG. This way all other components log at INFO level, and it is easier to read the logs with less noise.

6 Appendixes

6.1 Catalog of the Model Variables

https://deltares.sharepoint.com/:x:/r/teams/dlt1071/_layouts/15/Doc.aspx?sourcedoc=%7B56e8935a-0d0f-4881-b5b6-72b751e2cb49%7D

6.2 GWTF Platform API documentation

https://app.swaggerhub.com/apis/GrowWithTheFlow/GWTF

6.3 Production Specs

6.3.1 GWTF Platform Server on Google Cloud Platform

CPUs	2 vCPUs
CPU platform	Intel Skylake
RAM	6 GB
Zone	europe-west4-b
Location	Eemshaven, Netherlands
External IP address	34.90.118.148
Disk 0 (OS)[6]	30 GB
Disk 1 (Data)[7]	30 GB

[6] See section 5.3 for more information
[7] See section 5.3 for more information

6.3.2 MongoDB Database Specs on mongodb.com

Cloud provider	Google Cloud Platform
Region	Netherlands (europe-west4)
Cluster type	Dedicated[8]
RAM	1.7 GB per node
Storage	10 GB with storage scaling[9]
Network	1500 max connections
MongoDB version	4.4

6.3.3 Database backup policy

Snapshots of the database are automatically being taken for backup in 4 types of periods: Hourly (every 6 hours), Daily, Weekly (every Saturday) and Monthly (last day of month).
The retention time shows for how long each snapshot period is maintained.

Frequency Unit	Every	Retention Time	
Hourly Snapshot	6 hours	2	days
Daily Snapshot	N/A	7	days
Weekly Snapshot	Saturday	4	weeks
Monthly Snapshot	Last day of month	12	months

[8] The resources are not shared with other projects, i.e., they are exclusively used by GWTF Platform
[9] If storage exceeds 90% of disk capacity, it auto expands to maintain < 70% up to 128 GB. Beyond that, the database is upgraded to the next tier level.

6.4 Settings

The GWTF platform has a few settings that can be modified. Below is a description of them with the default values.

6.4.1 Web API service settings

Setting	Description	Default value
Container memory	The amount of RAM allocated to the Docker container	256 MB
HTTP host	The hostname/IP the server listens on	0.0.0.0
HTTP port	The TCP/IP port the server listens on	9090
DB URI	The MongoDB database address	
Log directory	The directory on the server where the logs will be written to	

There are additional settings that are defined in Akka HTTP, including HTTP request timeout, maximum number of HTTP connections and many others. For the complete list of Akka HTTP settings, their description and default values see the latest documentation of Akka HTTP.

6.4.2 Ingest service settings

Setting	Description	Default value
Container memory	The amount of RAM allocated to the Docker container	2.5 GB
DB URI	The MongoDB database address	
Log directory	The directory on the server where the logs will be written to	

6.5 Future steps

The following headings are subjects that should be part of future phases of the project. Also included are some recommendations for next steps for each subject.

6.5.1 Data Governance

Next steps: investigate which existing products can be integrated into the system to provide the different aspects of data governance, evaluate which one is the best fit, and estimate the amount of effort for integration.

6.5.2 Security

Next steps:

- Perform penetration tests periodically
- Evaluate risks and ensure they are mitigated
- Periodically evaluate status of security patches for each 3rd party dependency

6.5.3 Compliance and Legislation

Next steps:

- define the requirements for compliance and registration
- evaluate existing tools and examine how they can be integrated, what are the gaps.

6.5.4 Testing procedures

6.5.4.1 Functional test suites

6.5.4.2 Security test suites

6.5.4.3 Stress and performance test suites

Next steps:

- Plan the tests
- Automate where possible.
- Schedule and allocate resources for manual tests.

6.5.5 Maintenance procedures

Next steps: define maintenance procedures that adhere requirements in terms of availability, resilience, security, legislation, and regulations.

6.5.6 FAQ and Troubleshooting guide

Next steps: collect common user/dev issues from issue tracker (Jira) as well as collect feedback from users and compile into FAQ/Troubleshooting guides.