vantage6

A. van Gestel, B. van Beusekom, D. Smits, F. Martin, J. van Soest,

Apr 17, 2024

CONTENTS

CHAPTER

ONE

WHAT IS VANTAGE6?

Vantage6 stands for pri**va**cy preservi**n**g federa**t**ed le**a**rnin**g** infrastructur**e** for **s**ecure **i**nsight e**x**change.

The project is inspired by the Personal Health Train (PHT) concept. In this analogy vantage6 is the *tracks* and *stations*. Compatible algorithms are the *trains*, and computation tasks are the *journey*. Vantage6 is completely open source under the Apache License.

What vantage6 does:

- delivering algorithms to data stations and collecting their results
- managing users, organizations, collaborations, computation tasks and their results
- providing control (security) at the data-stations to their owners

The vantage6 infrastructure is designed with three fundamental functional aspects of federated learning.

1. **Autonomy**. All involved parties should remain independent and autonomous.
2. **Heterogeneity**. Parties should be allowed to have differences in hardware and operating systems.
3. **Flexibility**. Related to the latter, a federated learning infrastructure should not limit the use of relevant data.

CHAPTER

TWO

OVERVIEW OF THIS DOCUMENTATION

This documentation space consists of the following main sections:

- **Overview** → *You are here now*
- *Introduction* → *Introduction to vantage6 concepts*
- *Architecture* → *An more extensive explanation of vantage6 components*
- *User guide* → *How to use vantage6 as a researcher*
- *Node admin guide* → *How to install and configure vantage6 nodes*
- *Server admin guide* → *How to configure and deploy vantage6 servers*
- *Algorithm store admin guide* → *How to configure and deploy vantage6 algorithm stores*
- *Feature descriptions* (Under construction) → *Documentation of vantage6 features*
- *Developer community* → *How to collaborate on the development of the vantage6 infrastructure*
- *Algorithm Development* → *Develop algorithms that are compatible with vantage6*
- *Function documentation* → *Documentation of the vantage6 infrastructure code*
- *Glossary* → *A dictionary of common terms used in these docs*
- *Release notes* → *Log of what has been released and when*

CHAPTER

THREE

VANTAGE6 RESOURCES

This is a - non-exhaustive - list of vantage6 resources.

Documentation

- docs.vantage6.ai → *This documentation.*
- vantage6.ai → *vantage6 project website*
- Academic papers → *Technical insights into vantage6*

Source code

- vantage6 → *Contains all components (and the python-client).*
- Planning → *Contains all features, bugfixes and feature requests we are working on. To submit one yourself, you can create a* new issue.

Community

- Discord → *Chat with the vantage6 community*
- *Community meetings* → *Bi-monthly developer community meeting*

CHAPTER

FOUR

INDEX

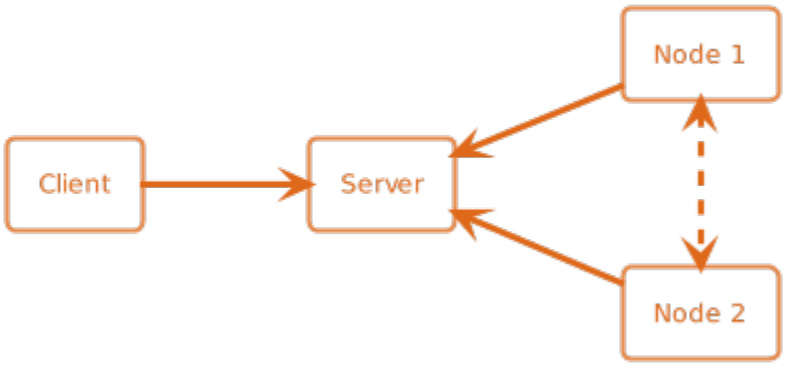
4.1 Introduction

In many research projects, data is distributed across multiple organizations. This makes it difficult to perform analyses that require data from multiple sources, as the data owners don't want to share their data with others. Vantage6 is a platform that enables privacy-enhancing analyses on distributed data. It allows organizations to collaborate on analyses while only sharing aggregated results, not the raw data.

As a user, you can use vantage6 to run your algorithms on sensitive data. In order to create the tasks to run your algorithms, it will be helpful to understand how vantage6 works. In order to help you understand this, we will first explain the basic architecture of vantage6, followed by a description of the resources that are available in vantage6. Using those concepts, we will explain give an example of a simple algorithm and explain how that is run within vantage6.

4.1.1 Vantage6 components

In vantage6, a **client** can pose a question to the central **server**. Each organization with sensitive data contributes one **node** to the network. The nodes collects the research question from the server and fetches the **algorithm** to answer it. When the algorithm completes, the node sends the aggregated results back to the server.



The roles of these vantage6 components are as follows:

- A (central) **server** coordinates communication with clients and nodes. The server tracks the status of the computation requests and handles administrative functions such as authentication and authorization.
- **Node(s)** have access to data and execute algorithms
- **Clients** (i.e. users or applications) request computations from the nodes via the client

- **Algorithms** are scripts that are run on the sensitive data. Each algorithm is packaged in a Docker image; the node pulls the image from a Docker registry and runs it on the local data. Note that the node owner can control which algorithms are allowed to run on their data.

On a technical level, vantage6 may be seen as a (Docker) container orchestration tool for privacy preserving analyses. It deploys a network of containerized applications that together ensure insights can be exchanged without sharing record-level data.

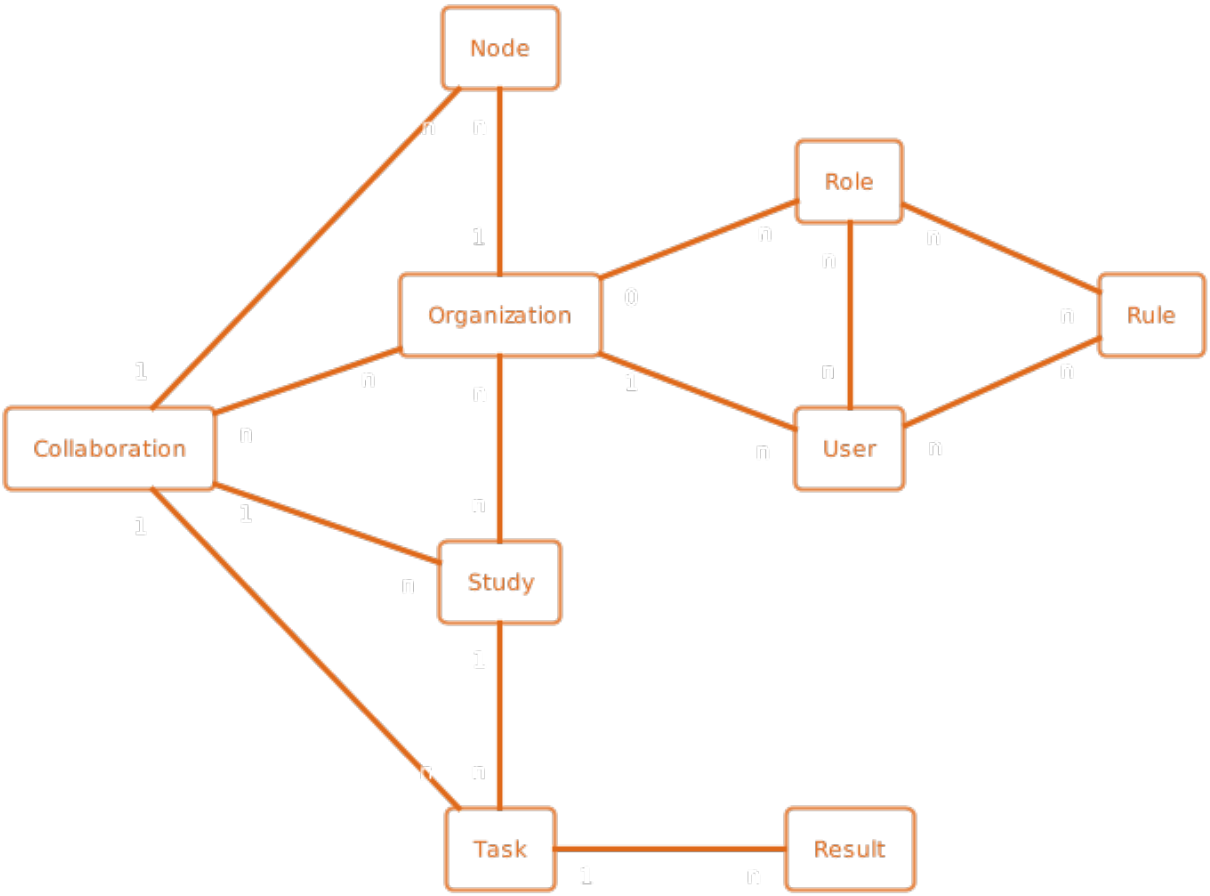
4.1.2 Vantage6 resources

There are several entities in vantage6, such as users, organizations, tasks, etc. These entities are created by users that have sufficient permission to do so and are stored in a database that is managed by the central server. This process ensures that the right people have the right access to the right actions, and that organizations can only collaborate with each other if they agree to do so.

The following statements and the figure below should help you understand their relationships.

- A **collaboration** is a collection of one or more **organizations**.
- For each collaboration, each participating organization needs a **node** to compute tasks. When a collaboration is created, accounts are also created for the nodes so that they can securely communicate with the server.
- Collaborations can contain **studies**. A study is a subset of organizations from the collaboration that are involved in a specific research question. By setting up studies, it can be easier to send tasks to a subset of the organizations in a collaboration and to keep track of the results of these analyses.
- Each organization has zero or more **users** who can perform certain actions.
- The permissions of the user are defined by the assigned **rules**.
- It is possible to collect multiple rules into a **role**, which can also be assigned to a user.
- Users can create **tasks** for one or more organizations within a collaboration. Tasks lead to the execution of the algorithms.
- A task should produce an algorithm **run** for each organization involved in the task. The **results** are part of such an algorithm run.

The following schema is a *simplified* version of the database. A *1-n* relationship means that the entity on the left side of the relationship can have multiple entities on the right side. For instance, a single organization can have multiple vantage6 users, but a single user always belongs to one organization. There is one *0-n* relationship between roles and organizations, since a role can be coupled to an organization, but it doesn't have to be. An *n-n* relationship is a many-to-many relationship: for instance, a collaboration can contain multiple organizations, and an organization can participate in multiple collaborations.



4.1.3 A simple federated average algorithm

To compute an average, you usually sum all the values and divide them by the number of values. In Python, this can be done as follows:

```
x = [1,2,3,4,5]
average = sum(x) / len(x)
```

In a federated data set the values for x are distributed over multiple locations. Let's assume x is split into two parties:

```
a = [1,2,3]
b = [4,5]
```

In this case we can compute the average as:

```
average = (sum(a) + sum(b))/(len(a) + len(b))
```

The goal is to compute the average without sharing the individual numbers. In the case of an average algorithm, each node therefore shares only the sum and the number of elements in the dataset. The server then computes the average by summing the sums and dividing by the sum of the number of elements. This way, the individual numbers are never shared.

4.1.4 How to run the algorithm in vantage6

The average algorithm explained above can be separated in a central part and a federated part. The federated part uses the data to compute the sum and the number of elements. The central part is the aggregation of these results. In order to do so, it is also responsible to start the federated parts and to collecting their results. Note that for more complex algorithms, this can be an iterative process: the central part can send new tasks to the federated parts based on the results of the previous round of federated tasks.

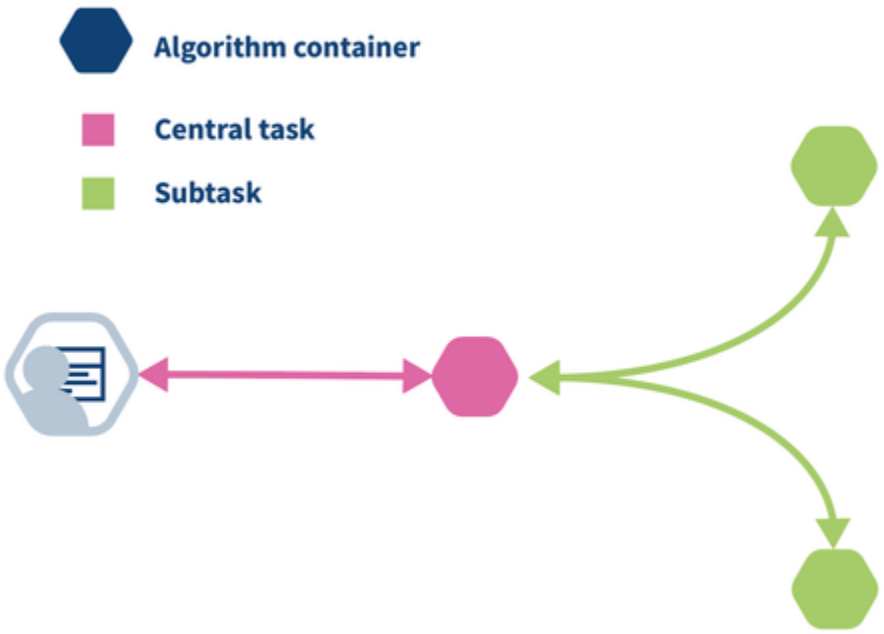


Fig. 4.1: Common task hierarchy in vantage6. The user (left) creates a task for the central part of the algorithm (pink hexagon). The central part creates subtasks for the federated parts (green hexagons). When the subtasks are finished, the central part collects the results and computes the final result, which is then available to the user.

Now, let's see how this works in vantage6. It is easy to confuse the central server with the central part of the algorithm: the server is the central part of the infrastructure but not the place where the central part of the algorithm is executed (Fig. 4.2). The central part is actually executed at one of the nodes, because it gives more flexibility: for instance, an algorithm may need heavy compute resources to do the aggregation, and it is better to do this at a node that has these resources rather than having to upgrade the server whenever a new algorithm needs more resources.

Note that is also possible for the user to create the subtasks directly, and to compute the central part of the algorithm themselves. This is however not the most common approach as it is in general easier to let the central algorithm do the work.

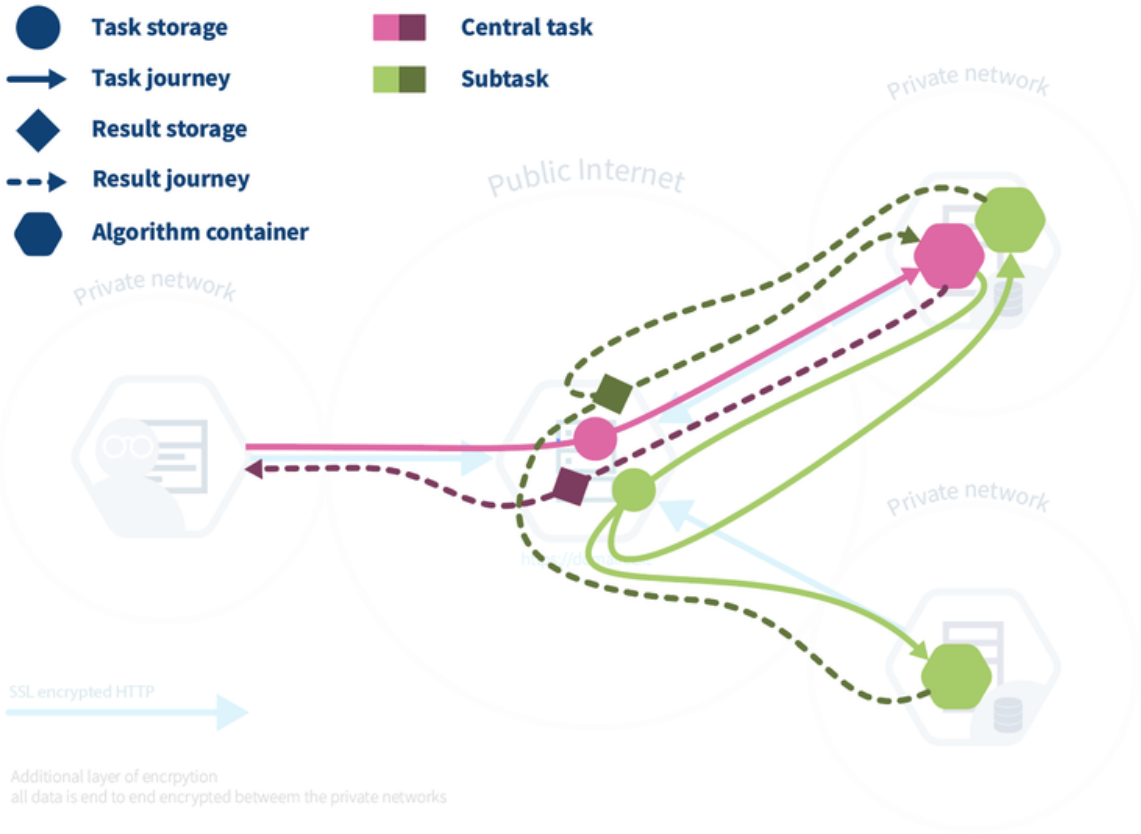


Fig. 4.2: The flow of the average algorithm in vantage6. The user creates a task for the central part of the algorithm. This is registered at the server, and leads to the creation of a central algorithm container on one of the nodes. The central algorithm then creates subtasks for the federated parts of the algorithm, which again are registered at the server. All nodes for which the subtask is intended start their work by executing the federated part of the algorithm. The nodes send the results back to the server, from where they are picked up by the central algorithm. The central algorithm then computes the final result and sends it to the server, where the user can retrieve it.

4.1.5 Running your own algorithms

Of course, the average algorithm is just an example. In practice, you can run many other algorithms in vantage6. The only requirement is that you package the algorithm in a Docker image that vantage6 can run. The focus of vantage6 is on setting up an infrastructure to run algorithms on sensitive data and ensuring that the data is kept private - the algorithm implementation is kept highly flexible.

The freedom in defining the code also allows you to use federated learning libraries such as PySyft, TensorFlow or Flower within your vantage6 algorithm. Also, it is not only possible to run federated algorithms, but also MPC algorithms or other protocols.

Note: Vantage6 tries to limit the definition of algorithms as little as possible. This means that within a project, it should be established which algorithms are allowed to run on the nodes. Review of this code - or trust in persons that have created the algorithm - is the responsibility of each node owner. They are ultimately in control over which algorithms are run on their data.

Vantage6 is designed to be as flexible as possible, so you can use any programming language and any libraries you like. Python is the most common language to use within the vantage6 community, and also has the most *tools* available to help you with your work.

4.2 Architecture

4.2.1 Network Actors

As we saw before, the vantage6 *network* consists of a central server, a number of nodes and a client. This section explains in some more detail what these network actors are responsible for, and which subcomponents they contain.

4.2.1.1 Server

The vantage6 server is the central point of contact for all communication in vantage6. However, it also relies on other infrastructure components to function properly. The following components are required for proper functioning of the server.

Vantage6 server
: Contains the users, organizations, collaborations, tasks and their results. It handles authentication and authorization to the system and is the central point of contact for clients and nodes.

Docker registry
: Contains algorithms stored in images which can be used by clients to request a computation. The node will retrieve the algorithm from this registry and execute it.

Algorithm store (optional but recommended)
: The algorithm store is intended to be used as a repository for trusted algorithms within a certain project. Algorithm stores can be coupled to specific collaborations or to all collaborations on a given server. Note that you can also couple the community algorithm store (https://store.cotopaxi.vantage6.ai) to your server. This store contains a number of community algorithms that you may find useful.

Note: The algorithm store is required when using the *user interface*. If no algorithm store is coupled to collaborations, no algorithms can be run from the user interface. It is also possible to couple collaborations to an algorithm store that you do not host yourself.

VPN server (optional)
If algorithms need to be able to engage in peer-to-peer communication, a VPN server can be set up to help them do so. This is usually the case when working with MPC, and is also often required for machine learning applications.

RabbitMQ message queue (optional)
The *vantage6 server* uses the socketIO protocol to communicate between server, nodes and clients. If there are multiple instances of the vantage6 server, it is important that the messages are communicated to all relevant actors, not just the ones that a certain server instance is connected to. RabbitMQ is therefore used to synchronize the messages between multiple *vantage6 server* instances.

4.2.1.2 Data Station

Vantage6 node
The node is responsible for executing the algorithms on the **local data**. It protects the data by allowing only specified algorithms to be executed after verifying their origin. The **node** is responsible for picking up the task, executing the algorithm and sending the results back to the server. The node needs access to local data. For more details see the *technical documentation of the node*.

Database
The database may be in any format that the algorithms relevant to your use case support. The currently supported database types are listed *here*.

Algorithm
When the node receives a task from the central server, it will download the algorithm from the Docker registry and execute it on the local data. The algorithm is therefore a temporary component that is automatically created by the node and only present during the execution of a task.

4.2.1.3 User or Application

Users or applications can interact with the vantage6 server in order to create tasks and retrieve their results, or manage entities at the server (i.e. creating or editing users, organizations and collaborations).

The vantage6 server is an API, which means that there are many ways to interact with it programatically. There are however a number of applications available that make is easier for users to interact with the vantage6 server. These are explained in more detail in the *User guide* but are also briefly mentioned here:

User interface
The *user interface* is a web application that allows users to interact with the server. It is used to create and manage organizations, collaborations, users, tasks and algorithms. It also allows users to view and download the results of tasks. Use of the user interface recommended for ease of use.

Python client
The *vantage6 python client <python-client>* is a Python package that allows users to interact with the server from a Python environment. This is helpful for data scientists who want to integrate vantage6 into their existing Python workflow.

API
It is also possible to interact with the server *using the API directly*.

Note: There is also an *R client* but this is not actively maintained and does not support all functionality.

4.2.1.4 Learn more?

If you want to learn more about specific components or features of vantage6, check out the *feature section* of the documentation. It contains detailed information about the different features of vantage6 and how to use them.

4.3 User guide

In this section of the documentation, we explain how you can interact with vantage6 servers and nodes as a user.

There are four ways in which you can interact with the central server: the *User interface* (UI), the *Python client*, the *R client*, and the *API*. In the sections below, we describe how to use each of these methods, and what you need to install (if anything).

For most use cases, we recommend to use the *User interface*, as this requires the least amount of effort. If you want to automate your workflow, we recommend using the *Python client*.

Warning: Note that for some algorithms, tasks cannot yet be created using the UI, or the results cannot be retrieved. This is because these algorithms have Python-specific datatypes that cannot be decoded in the UI. In this case, you will need to use the Python client to create the task and read the results.

Warning: Depending on your algorithm it *may* be required to use a specific language to post a task and retrieve the results. This could happen when the output of an algorithm contains a language specific datatype and or serialization.

4.3.1 User interface

The User Interface (UI) is a website where you can login with your vantage6 user account. Which website this is, depends on the vantage6 server you are using. If you are using the Cotopaxi server, go to https://portal.cotopaxi.vantage6.ai and login with your user account.

Using the UI should be relatively straightforward. There are buttons that should help you e.g. create a task or change your password. If anything is unclear, please contact us via Discord.

Note: If you are a server administrator and want to set up a user interface, see *this section* on deploying a UI.

Note: If you are *running your own server* with `v6 server start`, you can start the UI locally with `v6 server start --with-ui`, or you may specify that the UI should always be started in the `ui` section of the *server configuration file*.

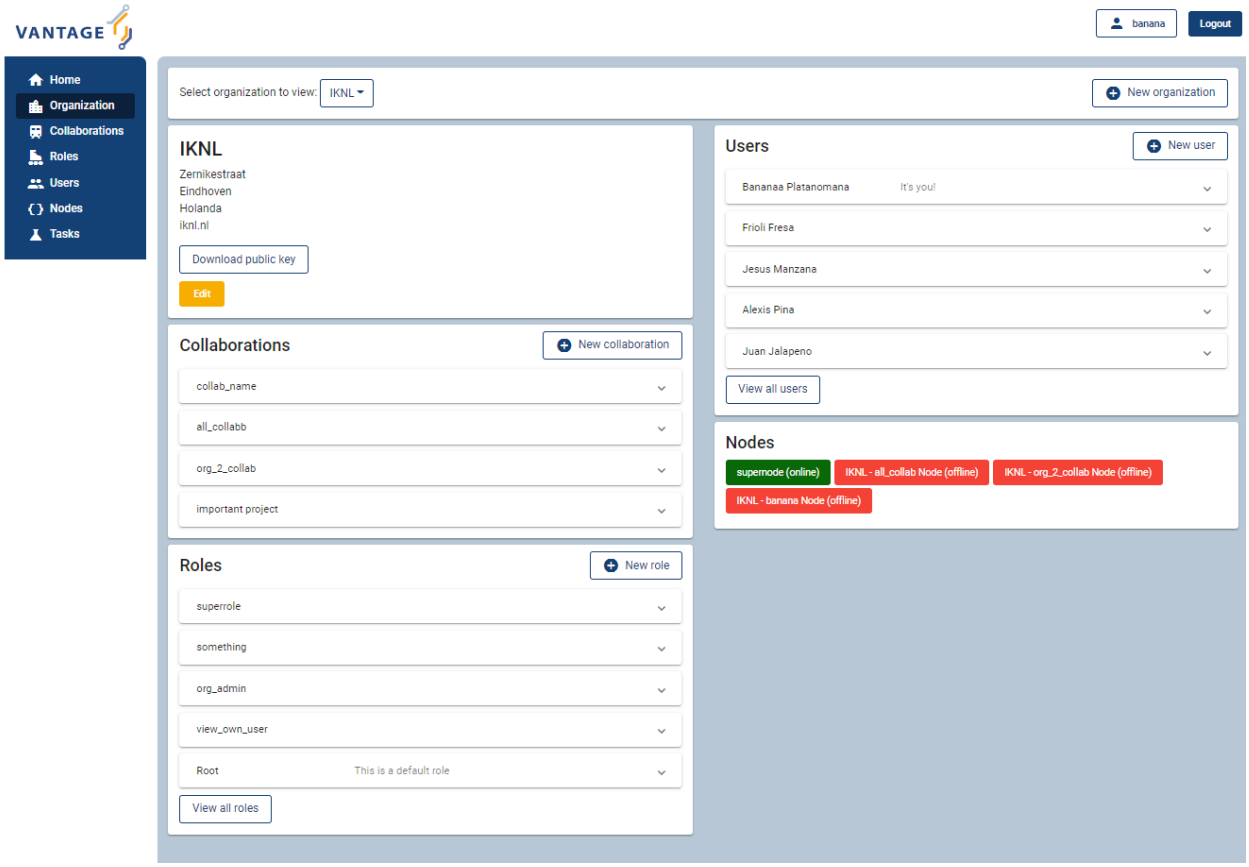


Fig. 4.3: Screenshot of the vantage6 UI

4.3.2 Python client

The Python client is the recommended way to interact with the vantage6 server for tasks that you want to automate. It is a Python library that facilitates communication with the vantage6 server, e.g. by encrypting and decrypting data for tasks for you.

The Python client aims to completely cover the vantage6 server communication. It can create computation tasks and collect their results, manage organizations, collaborations, users, etc. Under the hood, the Python client talks to the server API to achieve this.

4.3.2.1 Requirements

You need Python to use the Python client. We recommend using Python 3.10, as the client has been tested with this version. For higher versions, it may be difficult to install the dependencies.

> **Warning:** If you use a vantage6 version older than 3.8.0, you should use Python 3.7 instead of Python 3.10.

4.3.2.2 Install

It is important to install the Python client with the same version as the vantage6 server you are talking to. Check your server version by going to `https://<server_url>/version` (e.g. *https://cotopaxi.vantage6.ai/version* or *http://localhost:5000/api/version*) to find its version.

Then you can install the `vantage6-client` with:

```
pip install vantage6==<version>
```

where you add the version you want to install. You may also leave out the version to install the most recent version.

4.3.2.3 Use

First, we give an overview of the client. From the section *Authentication* onwards, there is example code of how to login with the client, and then create organizations, tasks etc.

Overview

The Python client contains groups of commands per resource type. For example, the group `client.user` has the following commands:

- `client.user.list()`: list all users
- `client.user.create(username, password, ...)`: create a new user
- `client.user.delete(<id>)`: delete a user
- `client.user.get(<id>)`: get a user

You can see how to use these methods by using `help(...)` , e.g. `help(client.task.create)` will show you the parameters needed to create a new user:

```
help(client.task.create)
#Create a new task
#
#    Parameters
#    ----------
#    collaboration : int
#        Id of the collaboration to which this task belongs
#    organizations : list
#        Organization ids (within the collaboration) which need
#        to execute this task
#    name : str
#        Human readable name
#    image : str
#        Docker image name which contains the algorithm
#    description : str
#        Human readable description
#    input : dict
#        Algorithm input
#    database: str, optional
#        Name of the database to use. This should match the key
#        in the node configuration files. If not specified the
#        default database will be tried.
#
#    Returns
#    -------
#    dict
#        Containing the task information
```

The following groups (related to the *Vantage6 resources*) of methods are available. They usually have `list()`, `create()`, `delete()` and `get()` methods attached - except where they are not relevant (for example, a rule that gives a certain permission cannot be deleted).

- `client.user`
- `client.organization`
- `client.rule`
- `client.role`
- `client.collaboration`
- `client.task`
- `client.run`
- `client.result`
- `client.node`

Finally, the class `client.util` contains some utility functions, for example to check if the server is up and running or to change your own password.

Authentication

This section and the following sections introduce some minimal examples for administrative tasks that you can perform with our *Python client*. We start by authenticating.

To authenticate, we create a config file to store our login information. We do this so we do not have to define the `server_url`, `server_port` and so on every time we want to use the client. Moreover, it enables us to separate the sensitive information (login details, organization key) that you do not want to make publicly available, from other parts of the code you might write later (e.g. on submitting particular tasks) that you might want to share publicly.

```
# config.py

server_url = "https://MY VANTAGE6 SERVER" # e.g. https://cotopaxi.vantage6.ai or
                                          # http://localhost for a local dev server
server_port = 443 # This is specified when you first created the server
server_api = "" # This is specified when you first created the server

username = "MY USERNAME"
password = "MY PASSWORD"

organization_key = "FILEPATH TO MY PRIVATE KEY" # This can be empty if you do not want␣
↪to set up encryption
```

Note that the `organization_key` should be a filepath that points to the private key that was generated when the organization to which your login belongs was first created (see *Creating an organization*).

Then, we connect to the vantage 6 server by initializing a Client object, and authenticating

```
from vantage6.client import UserClient as Client

# Note: we assume here the config.py you just created is in the current directory.
# If it is not, then you need to make sure it can be found on your PYTHONPATH
import config

# Initialize the client object, and run the authentication
client = Client(config.server_url, config.server_port, config.server_api,
                log_level='debug')
client.authenticate(config.username, config.password)

# Optional: setup the encryption, if you have an organization_key
client.setup_encryption(config.organization_key)
```

Creating an organization

After you have authenticated, you can start generating resources. The following also assumes that you have a login on the Vantage6 server that has the permissions to create a new organization. Regular end-users typically do not have these permissions (typically only administrators do); they may skip this part.

The first (optional, but recommended) step is to create an RSA keypair. A keypair, consisting of a private and a public key, can be used to encrypt data transfers. Users from the organization you are about to create will only be able to use encryption if such a keypair has been set up and if they have access to the private key.

```
from vantage6.common import warning, error, info, debug, bytes_to_base64s
from vantage6.client.encryption import RSACryptor
from pathlib import Path

# Generated a new private key
# Note that the file below doesn't exist yet: you will create it
private_key_filepath = r'/path/to/private/key'
private_key = RSACryptor.create_new_rsa_key(Path(private_key_filepath))

# Generate the public key based on the private one
public_key_bytes = RSACryptor.create_public_key_bytes(private_key)
public_key = bytes_to_base64s(public_key_bytes)
```

Now, we can create an organization

```
client.organization.create(
    name = 'The_Shire',
    address1 = '501 Buckland Road',
    address2 = 'Matamata',
    zipcode = '3472',
    country = 'New Zealand',
    domain = 'the_shire.org',
    public_key = public_key   # use None if you haven't set up encryption
)
```

Users can now be created for this organization. Any users that are created and who have access to the private key we generated above can now use encryption by running

```
client.setup_encryption('/path/to/private/key')
# or, if you don't use encryption
client.setup_encryption(None)
```

after they authenticate.

Creating a collaboration

Here, we assume that you have a Python session with an authenticated Client object, as created in *Authentication*. We also assume that you have a login on the Vantage6 server that has the permissions to create a new collaboration (regular end-users typically do not have these permissions, this is typically only for administrators).

A collaboration is an association of multiple organizations that want to run analyses together. First, you will need to find the organization id's of the organizations you want to be part of the collaboration.

```
client.organization.list(fields=['id', 'name'])
```

Once you know the id's of the organizations you want in the collaboration (e.g. 1 and 2), you can create the collaboration:

```
collaboration_name = "fictional_collab"
organization_ids = [1,2] # the id's of the respective organizations
client.collaboration.create(name = collaboration_name,
                            organizations = organization_ids,
                            encrypted = True)
```

Note that a collaboration can require participating organizations to use encryption, by passing the `encrypted = True` argument (as we did above) when creating the collaboration. It is recommended to do so, but requires that a keypair was created when *Creating an organization* and that each user of that organization has access to the private key so that they can run the `client.setup_encryption(...)` command after *Authentication*.

Registering a node

Here, we again assume that you have a Python session with an authenticated Client object, as created in *Authentication*, and that you have a login that has the permissions to create a new node (regular end-users typically do not have these permissions, this is typically only for administrators).

A node is associated with both a collaboration and an organization (see *Vantage6 resources*). You will need to find the collaboration and organization id's for the node you want to register:

```
client.organization.list(fields=['id', 'name'])
client.collaboration.list(fields=['id', 'name'])
```

Then, we register a node with the desired organization and collaboration. In this example, we create a node for the organization with id 1 and collaboration with id 1.

```
# A node is associated with both a collaboration and an organization
organization_id = 1
collaboration_id = 1
api_key = client.node.create(collaboration = collaboration_id, organization =␣
↪organization_id)
print(f"Registered a node for collaboration with id {collaboration_id}, organization␣
↪with id {organization_id}. The API key that was generated for this node was {api_key}")
```

Remember to save the `api_key` that is returned here, since you will need it when you *Configure* the node.

Creating a task

Preliminaries

Here we assume that

- you have a Python session with an authenticated Client object, as created in *Authentication*.
- you already have the algorithm you want to run available as a container in a docker registry (see here for more details on developing your own algorithm)
- the nodes are configured to look at the right database

In this manual, we'll use the averaging algorithm from `harbor2.vantage6.ai/demo/average`, so the second requirement is met. We'll assume the nodes in your collaboration have been configured to look as something like:

```
databases:
  - label: default
    uri: /path/to/my/example.csv
    type: csv
  - label: my_other_database
    uri: /path/to/my/example2.csv
    type: excel
```

The third requirement is met when all nodes have the same labels in their configuration. As an end-user running the algorithm, you'll need to align with the node owner about which database name is used for the database you are interested in. For more details, see *how to configure* your node.

Determining which collaboration / organizations to create a task for

First, you'll want to determine which collaboration to submit this task to, and which organizations from this collaboration you want to be involved in the analysis

```
>>> client.collaboration.list(fields=['id', 'name', 'organizations'])
[
 {'id': 1, 'name': 'example_collab1',
 'organizations': [
     {'id': 2, 'link': '/api/organization/2', 'methods': ['GET', 'PATCH']},
     {'id': 3, 'link': '/api/organization/3', 'methods': ['GET', 'PATCH']},
     {'id': 4, 'link': '/api/organization/4', 'methods': ['GET', 'PATCH']}
 ]}
]
```

In this example, we see that the collaboration called `example_collab1` has three organizations associated with it, of which the organization id's are `2`, `3` and `4`. To figure out the names of these organizations, we run:

```
>>> client.organization.list(fields=['id', 'name'])
[{'id': 1, 'name': 'root'}, {'id': 2, 'name': 'example_org1'},
 {'id': 3, 'name': 'example_org2'}, {'id': 4, 'name': 'example_org3'}]
```

i.e. this collaboration consists of the organizations `example_org1` (with id `2`), `example_org2` (with id `3`) and `example_org3` (with id `4`).

Creating a task that runs the central algorithm

Now, we have two options: create a task that will run the central part of an algorithm (which runs on one node and may spawns subtasks on other nodes), or create a task that will (only) run the partial methods (which are run on each node). Typically, the partial methods only run the node local analysis (e.g. compute the averages per node), whereas the central methods performs aggregation of those results as well (e.g. starts the partial analyses and then computes the overall average). First, let us create a task that runs the central part of the `harbor2.vantage6.ai/demo/average` algorithm:

```
input_ = {
    'method': 'central_average',
    'kwargs': {'column_name': 'age'}
}

average_task = client.task.create(
   collaboration=1,
   organizations=[2,3],
   name="an-awesome-task",
   image="harbor2.vantage6.ai/demo/average",
   description='',
   input_=input_,
   databases=[
      {'label': 'default'}
   ]
)
```

Note that the `kwargs` we specified in the `input_` are specific to this algorithm: this algorithm expects an argument `column_name` to be defined, and will compute the average over the column with that name. Furthermore, note that

here we created a task for collaboration with id 1 (i.e. our `example_collab1`) and the organizations with id 2 and 3 (i.e. `example_org1` and `example_org2`). I.e. the algorithm need not necessarily be run on *all* the organizations involved in the collaboration.

Finally, note that you should provide any databases that you want to use via the `databases` argument. In the example above, we use the `default` database; using the `my_other_database` database can be done by simply specifying that label in the dictionary. If you have a SQL or SPARQL database, you should also provide a `query` argument, e.g.

```
databases=[
   {'label': 'default', 'query': 'SELECT * FROM my_table'}
]
```

Similarly, you can define a `sheet_name` for Excel databases if you want to read data from a specific worksheet. Check `help(client.task.create)` for more information.

Creating a task that runs the partial algorithm

You might be interested to know output of the partial algorithm (in this example: the averages for the 'age' column for each node). In that case, you can run only the partial algorithm, omitting the aggregation that the central part of the algorithm will normally do:

```
input_ = {
    'method': 'partial_average',
    'kwargs': {'column_name': 'age'},
}

average_task = client.task.create(collaboration=1,
                                  organizations=[2,3],
                                  name="an-awesome-task",
                                  image="harbor2.vantage6.ai/demo/average",
                                  description='',
                                  input_=input_)
```

Inspecting the results

Of course, it will take a little while to run your algorithm. You can use the following code snippet to run a loop that checks the server every 3 seconds to see if the task has been completed:

```
print("Waiting for results")
task_id = average_task['id']
result = client.wait_for_results(task_id)
```

You can also check the status of the task using:

```
task_info = client.task.get(task_id, include_results=True)
```

and then retrieve the results

```
result_info = client.result.from_task(task_id=task_id)
```

The number of results may be different depending on what you run, but for the central average algorithm in this example, the results would be:

```
>>> result_info
[{'average': 53.25}]
```

while for the partial algorithms, dispatched to two nodes, the results would be:

```
>>> result_info
[{'sum': 253, 'count': 4}, {'sum': 173, 'count': 4}]
```

4.3.3 R client

Warning: We discourage the use of the R client. It is not actively maintained and is not fully implemented. It can not (yet) be used to manage resources, such as creating and deleting users and organizations.

4.3.3.1 Install

You can install the R client by running:

```
devtools::install_github('IKNL/vtg')
```

4.3.3.2 Use

The R client can only create tasks and retrieve their results.

Initialization of the R client can be done by:

```
setup.client <- function() {
  # Username/password should be provided by the administrator of
  # the server.
  username <- "username@example.com"
  password <- "password"

  host <- 'https://cotopaxi.vantage6.ai:443'
  api_path <- ''

  # Create the client & authenticate
  client <- vtg::Client$new(host, api_path=api_path)
  client$authenticate(username, password)

  return(client)
}

# Create a client
client <- setup.client()
```

Then, this client can be used for the different algorithms. Refer to the README in the repository on how to call the algorithm. Usually this includes installing some additional client-side packages for the specific algorithm you are using.

Example

This example shows how to run the vantage6 implementation of a federated Cox Proportional Hazard regression model. First you need to install the client side of the algorithm by:

```
devtools::install_github('iknl/vtg.coxph', subdir="src")
```

This is the code to run the coxph:

```
print( client$getCollaborations() )

# Should output something like this:
#   id     name
# 1  1 ZEPPELIN
# 2  2 PIPELINE

# Select a collaboration
client$setCollaborationId(1)

# Define explanatory variables, time column and censor column
expl_vars <- c("Age","Race2","Race3","Mar2","Mar3","Mar4","Mar5","Mar9",
               "Hist8520","hist8522","hist8480","hist8501","hist8201",
               "hist8211","grade","ts","nne","npn","er2","er4")
time_col <- "Time"
censor_col <- "Censor"

# vtg.coxph contains the function `dcoxph`.
result <- vtg.coxph::dcoxph(client, expl_vars, time_col, censor_col)
```

4.3.4 API

The API can be called via HTTP requests from a programming language of your choice. You can explore how to use the server API on `https://<serverdomain>/apidocs` (e.g. https://cotopaxi.vantage6.ai/apidocs for our Cotopaxi server). This page will show you which API endpoints exist and how you can use them.

4.4 Node admin guide

This section shows you how you can set up your own vantage6 node. First, we discuss the requirements for your node machine, then guide you through the installation process. Finally, we explain how to configure and start your node.

4.4.1 Introduction

The vantage6 node is the software that runs on the machine of the data owner. It is responsible for the execution of the federated learning tasks and the communication with the central server.

Each organization that is involved in a federated learning collaboration has its own node in that collaboration. They should therefore install the node software on a virtual machine hosted in their own infrastructure. The node should have access to the data that is used in the federated learning collaboration.

The following pages explain how to install and run the node software.

4.4.2 Requirements

Note: This section is almost the same as the *server requirements* section - their requirements are very similar.

Below are the minimal requirements for vantage6 infrastructure components. Note that these are recommendations: it may also work on other hardware, operating systems, versions of Python etc. (but they are not tested as much).

Hardware

- x86 CPU architecture + virtualization enabled
- 1 GB memory
- 50GB+ storage
- Stable and fast (1 Mbps+ internet connection)

Software

- Operating system: Ubuntu 18.04+ , MacOS Big Sur+, Windows 10+
- Python
- Docker

Note: For the server, Ubuntu is highly recommended. It is possible to run a development server (using *v6 server start*) on Windows or MacOS, but for production purposes we recommend using Ubuntu.

Warning: The hardware requirements of the node also depend on the algorithms that the node will run. For example, you need much less compute power for a descriptive statistical algorithm than for a machine learning model.

4.4.2.1 Python

Installation of any of the vantage6 packages requires Python 3.10. For installation instructions, see python.org, anaconda.com or use the package manager native to your OS and/or distribution.

Note: We recommend you install vantage6 in a new, clean Python (Conda) environment.

Higher versions of Python (3.11+) will most likely also work, as might lower versions (3.8 or 3.9). However, we develop and test vantage6 on version 3.10, so that is the safest choice.

Warning: Note that Python 3.10 is only used in vantage6 versions 3.8.0 and higher. In lower versions, Python 3.7 is required.

4.4.2.2 Docker

Docker facilitates encapsulation of applications and their dependencies in packages that can be easily distributed to diverse systems. Algorithms are stored in Docker images which nodes can download and execute. Besides the algorithms, both the node and server are also running from a docker container themselves.

Please refer to this page on how to install Docker. To verify that Docker is installed and running you can run the `hello-world` example from Docker.

```
docker run hello-world
```

Warning: Note that for **Linux**, some post-installation steps may be required. Vantage6 needs to be able to run docker without `sudo`, and these steps ensure just that.

For Windows, if you are using Docker Desktop, it may be preferable to limit the amount of memory Docker can use - in some cases it may otherwise consume much memory and slow down the system. This may be achieved as described here.

Note:

- Always make sure that Docker is running while using vantage6!
- We recommend to always use the latest version of Docker.

4.4.3 Install

To install the **vantage6 node** make sure you have met the *requirements*. Then, we provide a command-line interface (CLI) with which you can manage your node. The CLI is a Python package that can be installed using pip. We always recommend to install the CLI in a virtual environment or a conda environment.

Run this command to install the CLI in your environment:

```
pip install vantage6
```

Or if you want to install a specific version:

```
pip install vantage6==x.y.z
```

You can verify that the CLI has been installed by running the command `v6 node --help`. If that prints a list of commands, the installation is completed.

The next pages will explain to configure, start and stop the node. The node software itself will be downloaded when you start the node for the first time.

4.4.4 Use

This section explains which commands are available to manage your node.

4.4.4.1 Quick start

To create a new node, run the command below. A menu will be started that allows you to set up a node configuration file. For more details, check out the *Configure* section.

```
v6 node new
```

To run a node, execute the command below. The `--attach` flag will cause log output to be printed to the console.

```
v6 node start --name <your_node> --attach
```

Finally, a node can be stopped again with:

```
v6 node stop --name <your_node>
```

Note: Before the node is started, it is attempted to obtain the server version. For a server of version `x.y.z`, a node of version `x.y.<latest>` is started - this is the latest available node version for the server version. If no server version can be obtained, the latest node of the same major version as the command-line interface installation is started.

4.4.4.2 Available commands

Below is a list of all commands you can run for your node(s). To see all available options per command use the `--help` flag, i.e. `v6 node start --help` .

Command	Description
`v6 node new`	Create a new node configuration file
`v6 node start`	Start a node
`v6 node stop`	Stop a nodes
`v6 node files`	List the files of a node (e.g. config and log files)
`v6 node attach`	Print the node logs to the console
`v6 node list`	List all existing nodes
`v6 node create-private-key`	Create and upload a new public key for your organization
`v6 node set-api-key`	Update the API key in your node configuration file

Local test setup

Check the section on *Local test setup* of the server if you want to run both the node and server on the same machine.

4.4.5 Configure

The vantage6-node requires a configuration file to run. This is a `yaml` file with a specific format.

The next sections describes how to configure the node. It first provides a few quick answers on setting up your node, then shows an example of all configuration file options, and finally explains where your vantage6 configuration files are stored.

4.4.5.1 How to create a configuration file

The easiest way to create an initial configuration file is via: `v6 node new`. This allows you to configure the basic settings. For more advanced configuration options, which are listed below, you can view the *example configuration file*.

4.4.5.2 Where is my configuration file?

To see where your configuration file is located, you can use the following command

```
v6 node files
```

Warning: This command will not work if you have put your configuration file in a custom location. Also, you may need to specify the `--system` flag if you put your configuration file in the *system folder*.

4.4.5.3 All configuration options

The following configuration file is an example that intends to list all possible configuration options.

You can download this file here: `node_config.yaml`

```yaml
# API key used to authenticate at the server.
api_key: ***

# URL of the vantage6 server
server_url: https://cotopaxi.vantage6.ai

# port the server listens to
port: 443

# API path prefix that the server uses. Usually '/api' or an empty string
api_path: ''

# subnet of the VPN server
vpn_subnet: 10.76.0.0/16

# set the devices the algorithm container is allowed to request.
algorithm_device_requests:
  gpu: false

# Add additional environment variables to the algorithm containers. In case
# you want to supply database specific environment (e.g. usernames and
```

(continues on next page)

(continued from previous page)

```
# passwords) you should use `env` key in the `database` section of this
# configuration file.
# OPTIONAL
algorithm_env:

  # in this example the environment variable 'player' has
  # the value 'Alice' inside the algorithm container
  player: Alice

# Add additional environment variables to the node container. This can be useful
# if you need to modify the configuration of certain python libraries that the
# node uses. For example, if you want to use a custom CA bundle for the requests
# library you can specify it here.
node_extra_env:
  REQUESTS_CA_BUNDLE: /etc/ssl/certs/ca-certificates.crt

# Add additional volumes to the node container. This can be useful if you need
# to mount a custom CA bundle for the requests library for example.
node_extra_mounts:
  - /etc/ssl/certs/ca-certificates.crt:/etc/ssl/certs/ca-certificates.crt:ro

node_extra_hosts:
  # In Linux (no Docker Desktop) you can use this (special) mapping to access
  # the host from the node.
  # See: https://docs.docker.com/reference/cli/docker/container/run/#add-host
  host.docker.internal: host-gateway
  # For testing purposes, it can also be used to map a public domain to a
  # private IP address, allowing you to avoid breaking TLS hostname verification
  v6server.example.com: 192.168.1.10

# specify custom Docker images to use for starting the different
# components.
# OPTIONAL
images:
  node: harbor2.vantage6.ai/infrastructure/node:cotopaxi
  alpine: harbor2.vantage6.ai/infrastructure/alpine
  vpn_client: harbor2.vantage6.ai/infrastructure/vpn_client
  network_config: harbor2.vantage6.ai/infrastructure/vpn_network
  ssh_tunnel: harbor2.vantage6.ai/infrastructure/ssh_tunnel
  squid: harbor2.vantage6.ai/infrastructure/squid

# path or endpoint to the local data source. The client can request a
# certain database by using its label. The type is used by the
# auto_wrapper method used by algorithms. This way the algorithm wrapper
# knows how to read the data from the source. The auto_wrapper currently
# supports: 'csv', 'parquet', 'sql', 'sparql', 'excel', 'omop'. If your
# algorithm does not use the wrapper and you have a different type of
# data source you can specify 'other'.
databases:
  - label: default
    uri: C:\data\datafile.csv
    type: csv
```

(continues on next page)

(continued from previous page)

```
  - label: omop
    uri: jdbc:postgresql://host.docker.internal:5454/postgres
    type: omop
    # additional environment variables that are passed to the algorithm
    # containers (or their wrapper). This can be used to for usernames
    # and passwords for example. Note that these environment variables are
    # only passed to the algorithm container when the user requests that
    # database. In case you want to pass some environment variable to all
    # algorithms regard less of the data source the user specifies you can
    # use the `algorithm_env` setting.
    env:
      user: admin@admin.com
      password: admin
      dbms: postgresql
      cdm_database: postgres
      cdm_schema: public
      results_schema: results


# end-to-end encryption settings
encryption:

  # whenever encryption is enabled or not. This should be the same
  # as the `encrypted` setting of the collaboration to which this
  # node belongs.
  enabled: false

  # location to the private key file
  private_key: /path/to/private_key.pem

# Define who is allowed to run which algorithms on this node.
policies:
  # Control which algorithm images are allowed to run on this node. This is
  # expected to be a valid regular expression.
  allowed_algorithms:
    - ^harbor2\.vantage6\.ai/[a-zA-Z]+/[a-zA-Z]+
    - ^myalgorithm\.ai/some-algorithm
  # Define which users are allowed to run algorithms on your node by their ID
  allowed_users:
    - 2
  # Define which organizations are allowed to run images on your node by
  # their ID or name
  allowed_organizations:
    - 6
    - root

  # The basics algorithm (harbor2.vantage5.ai/algorithms/basics) is whitelisted
  # by default. It is used to collect column names in the User Interface to
  # facilitate task creation. Set to false to disable this.
  allow_basics_algorithm: true
```

(continues on next page)

(continued from previous page)

```
# credentials used to login to private Docker registries
docker_registries:
  - registry: docker-registry.org
    username: docker-registry-user
    password: docker-registry-password

# Create SSH Tunnel to connect algorithms to external data sources. The
# `hostname` and `tunnel:bind:port` can be used by the algorithm
# container to connect to the external data source. This is the address
# you need to use in the `databases` section of the configuration file!
ssh-tunnels:

  # Hostname to be used within the internal network. I.e. this is the
  # hostname that the algorithm uses to connect to the data source. Make
  # sure this is unique and the same as what you specified in the
  # `databases` section of the configuration file.
  - hostname: my-data-source

    # SSH configuration of the remote machine
    ssh:

      # Hostname or ip of the remote machine, in case it is the docker
      # host you can use `host.docker.internal` for Windows and MacOS.
      # In the case of Linux you can use `172.17.0.1` (the ip of the
      # docker bridge on the host)
      host: host.docker.internal
      port: 22

      # fingerprint of the remote machine. This is used to verify the
      # authenticity of the remote machine.
      fingerprint: "ssh-rsa ..."

      # Username and private key to use for authentication on the remote
      # machine
      identity:
        username: username
        key: /path/to/private_key.pem

    # Once the SSH connection is established, a tunnel is created to
    # forward traffic from the local machine to the remote machine.
    tunnel:

      # The port and ip on the tunnel container. The ip is always
      # 0.0.0.0 as we want the algorithm container to be able to
      # connect.
      bind:
        ip: 0.0.0.0
        port: 8000

      # The port and ip on the remote machine. If the data source runs
      # on this machine, the ip most likely is 127.0.0.1.
      dest:
```

(continues on next page)

(continued from previous page)

```
        ip: 127.0.0.1
        port: 8000

# Whitelist URLs and/or IP addresses that the algorithm containers are
# allowed to reach using the http protocol.
whitelist:
  domains:
    - google.com
    - github.com
    - host.docker.internal # docker host ip (windows/mac)
  ips:
    - 172.17.0.1 # docker bridge ip (linux)
    - 8.8.8.8
  ports:
    - 443

# Containers that are defined here are linked to the algorithm containers and
# can therefore be accessed when by the algorithm when it is running. Note that
# for using this option, the container with 'container_name' should already be
# started before the node is started.
docker_services:
    container_label: container_name

# Settings for the logger
logging:
  # Controls the logging output level. Could be one of the following
  # levels: CRITICAL, ERROR, WARNING, INFO, DEBUG, NOTSET
  level:        DEBUG

  # whenever the output needs to be shown in the console
  use_console:  true

  # The number of log files that are kept, used by RotatingFileHandler
  backup_count: 5

  # Size kb of a single log file, used by RotatingFileHandler
  max_size:     1024

  # Format: input for logging.Formatter,
  format:       "%(asctime)s - %(name)-14s - %(levelname)-8s - %(message)s"
  datefmt:      "%Y-%m-%d %H:%M:%S"

  # (optional) set the individual log levels per logger name, for example
  # mute some loggers that are too verbose.
  loggers:
    - name: urllib3
      level: warning
    - name: requests
      level: warning
    - name: engineio.client
      level: warning
    - name: docker.utils.config
```

(continues on next page)

(continued from previous page)

```
      level: warning
    - name: docker.auth
      level: warning

# Additional debug flags
debug:

  # Set to `true` to enable the Flask/socketio into debug mode.
  socketio: false

  # Set to `true` to set the Flask app used for the LOCAL proxy service
  # into debug mode
  proxy_server: false


# directory where local task files (input/output) are stored
task_dir: C:\Users\<your-user>\AppData\Local\vantage6\node\mydir

# Whether or not your node shares some configuration (e.g. which images are
# allowed to run on your node) with the central server. This can be useful
# for other organizations in your collaboration to understand why a task
# is not completed. Obviously, no sensitive data is shared. Default true
share_config: true
```

4.4.5.4 Configuration file location

The directory where the configuration file is stored depends on your operating system (OS). It is possible to store the configuration file at **system** or at **user** level. By default, node configuration files are stored at **user** level, which makes this configuration available only for your user.

The default directories per OS are as follows:

Operating System	System-folder	User-folder
Windows	`C:\ProgramData\vantage\node\`	`C:\Users\<user>\AppData\Local\vantage\node\`
MacOS	`/Library/Application/Support/vantage6/node/`	`/Users/<user>/Library/Application Support/vantage6/node/`
Linux	`/etc/vantage6/node/`	`/home/<user>/.config/vantage6/node/`

Note: The command `v6 node` looks in these directories by default. However, it is possible to use any directory and specify the location with the `--config` flag. But note that doing that requires you to specify the `--config` flag every time you execute a `v6 node` command!

Similarly, you can put your node configuration file in the system folder by using the `--system` flag. Note that in that case, you have to specify the `--system` flag for all `v6 node` commands.

4.4.5.5 Security

As a data owner it is important that you take the necessary steps to protect your data. Vantage6 allows algorithms to run on your data and share the results with other parties. It is important that you review the algorithms before allowing them to run on your data.

Once you approved the algorithm, it is important that you can verify that the approved algorithm is the algorithm that runs on your data. There are two important steps to be taken to accomplish this:

- Set the (optional) `allowed_algorithms` option in the `policies` section of the node-configuration file. You can specify a list of regex expressions here. Some examples of what you could define:
  1. `^harbor2\.vantage6\.ai/[a-zA-Z]+/[a-zA-Z]+`: allow all images from the vantage6 registry
  2. `^harbor2\.vantage6\.ai/algorithms/glm$`: only allow the GLM image, but all builds of this image
  3. `^harbor2\.vantage6\.ai/algorithms/glm@sha256:82becede498899ec668628e7cb0ad87b6e1c371cb8a1e597d83a47fac21d6af3$`: allows only this specific build from the GLM image to run on your data
- Enable `DOCKER_CONTENT_TRUST` to verify the origin of the image. For more details see the documentation from Docker.

> **Warning:** By enabling `DOCKER_CONTENT_TRUST` you might not be able to use certain algorithms. You can check this by verifying that the images you want to be used are signed.

4.4.5.6 Logging

To configure the logger, look at the logging section in the example configuration file in *All configuration options*.

Useful commands:

1. `v6 node files`: shows you where the log file is stored
2. `v6 node attach`: shows live logs of a running server in your current console. This can also be achieved when starting the node with `v6 node start --attach`

4.5 Server admin guide

This section shows you how you can set up your own vantage6 server. First, we discuss the requirements for your server machine, then guide you through the installation process. Finally, we explain how to configure and start your server.

4.5.1 Introduction

The vantage6 server is the central component of the vantage6 platform. It is responsible for managing the different organizations and their nodes, authenticating and authorizing the users and nodes, and managing the communication of task requests and results between the nodes and the users.

All communication in vantage6 is managed through the server's RESTful API and socketIO server. There are also a couple of other services that you may want to run alongside the server, such as a user interface and a message broker.

The following pages explain how to install and configure the server, and how to run a test server or deploy a production server. It also explains which optional services you may want to run alongside the server, and how to configure them.

4.5.2 Requirements

Note: This section is almost the same as the *node requirements* section - their requirements are very similar.

Below are the minimal requirements for vantage6 infrastructure components. Note that these are recommendations: it may also work on other hardware, operating systems, versions of Python etc. (but they are not tested as much).

Hardware

- x86 CPU architecture + virtualization enabled
- 1 GB memory
- 50GB+ storage
- Stable and fast (1 Mbps+ internet connection)

Note that the server's IP address should also be reachable by all users and nodes. This will usually be a public IP address.

Software

- Operating system: Ubuntu 18.04+
- Python
- Docker

Note: For the server, Ubuntu is highly recommended. It is possible to run a development server (using *v6 server start*) on Windows or MacOS, but for production purposes we recommend using Ubuntu.

Warning: The hardware requirements of the node also depend on the algorithms that the node will run. For example, you need much less compute power for a descriptive statistical algorithm than for a machine learning model.

4.5.2.1 Python

Installation of any of the vantage6 packages requires Python 3.10. For installation instructions, see python.org, anaconda.com or use the package manager native to your OS and/or distribution.

Note: We recommend you install vantage6 in a new, clean Python (Conda) environment.

Higher versions of Python (3.11+) will most likely also work, as might lower versions (3.8 or 3.9). However, we develop and test vantage6 on version 3.10, so that is the safest choice.

Warning: Note that Python 3.10 is only used in vantage6 versions 3.8.0 and higher. In lower versions, Python 3.7 is required.

4.5.2.2 Docker

Docker facilitates encapsulation of applications and their dependencies in packages that can be easily distributed to diverse systems. Algorithms are stored in Docker images which nodes can download and execute. Besides the algorithms, both the node and server are also running from a docker container themselves.

Please refer to this page on how to install Docker. To verify that Docker is installed and running you can run the `hello-world` example from Docker.

```
docker run hello-world
```

Warning: Note that for **Linux**, some post-installation steps may be required. Vantage6 needs to be able to run docker without `sudo`, and these steps ensure just that.

For Windows, if you are using Docker Desktop, it may be preferable to limit the amount of memory Docker can use - in some cases it may otherwise consume much memory and slow down the system. This may be achieved as described here.

Note:

- Always make sure that Docker is running while using vantage6!
- We recommend to always use the latest version of Docker.

4.5.3 Install

4.5.3.1 Local (test) Installation

To install the vantage6 server, make sure you have met the *requirements*. Then, we provide a command-line interface (CLI) with which you can manage your server. The CLI is a Python package that can be installed using pip. We always recommend to install the CLI in a virtual environment or a conda environment.

Run this command to install the CLI in your environment:

```
pip install vantage6
```

Or if you want to install a specific version:

```
pip install vantage6==x.y.z
```

You can verify that the CLI has been installed by running the command `v6 --help`. If that prints a list of commands, the installation is completed.

The server software itself will be downloaded when you start the server for the first time.

4.5.3.2 Hosting your server

To host your server, we recommend to use the Docker image we provide: `harbor2.vantage6.ai/infrastructure/server`. Running this docker image will start the server. Check the *deployment* section for deployment examples.

Note: We recommend to use the latest version. Should you have reasons to deploy an older `VERSION`, use the image `harbor2.vantage6.ai/infrastructure/server:<VERSION>`.

4.5.4 Deploy

The vantage6 server is a Flask application, that uses python-socketio for socketIO connections. The server runs as a standalone process (listening on its own ip address/port).

There are many deployment options. We simply provide a few examples.

- *NGINX*
- *Docker compose*
- ...

Note: Because the server uses socketIO to exchange messages with the nodes and users, it is not trivial to horizontally scale the server. To prevent that socket messages get lost, you should deploy a RabbitMQ service and configure the server to use it. *This section* explains how to do so.

4.5.4.1 NGINX

A basic setup is shown below. Note that SSL is not configured in this example.

```
server {

    # Public port
    listen 80;
    server_name _;

    # vantage6-server. In the case you use a sub-path here, make sure
    # to foward also it to the proxy_pass
    location /subpath {
        include proxy_params;

        # internal ip and port
        proxy_pass http://127.0.0.1:5000/subpath;
    }

    # Allow the websocket traffic
    location /socket.io {
        include proxy_params;
        proxy_http_version 1.1;
        proxy_buffering off;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "Upgrade";
```

(continues on next page)

(continued from previous page)

```
        proxy_pass http://127.0.0.1:5000/socket.io;
    }
}
```

Note: When you *Configure* the server, make sure to include the `/subpath` that has been set in the NGINX configuration into the `api_path` setting (e.g. `api_path: /subpath/api`)

4.5.4.2 Docker compose

An alternative to `v6 server start` is to use docker-compose. Below is an example of a `docker-compose.yml` file that may be used to start the server. Obviously, you may want to change this to your own situtation. For example, you may want to use a different image tag, or you may want to use a different port.

```
services:
  vantage6-server:
    image: harbor2.vantage6.ai/infrastructure/server:cotopaxi
    ports:
    - "8000:5000"
    volumes:
    - /path/to/my/server.yaml:/mnt/config.yaml
    command: ["/bin/bash", "-c", "/vantage6/vantage6-server/server.sh"]
```

4.5.5 Install optional components

There are several optional components that you can set up apart from the vantage6 server itself.

User Interface
: An application that will allow your server's users to interact more easily with your vantage6 server.

Docker registry
: A (private) Docker registry can be used to store algorithms but it is also possible to use the (public) Docker hub to upload your Docker images. For production scenarios, we recommend using a private registry.

EduVPN
: If you want to enable algorithm containers that are running on different nodes, to directly communicate with one another, you require a VPN server.

RabbitMQ
: If you have a server with a high workload whose performance you want to improve, you may want to set up a RabbitMQ service which enables horizontal scaling of the Vantage6 server.

SMTP server
: If you want to send emails to your users, e.g. to help them reset their password, you need to set up an SMTP server.

Below, we explain how to install and deploy these components.

4.5.5.1 User Interface

The User Interface (UI) is a web application that will make it easier for your users to interact with the server. It allows you to manage all your resources (such as creating collaborations, editing users, or viewing tasks), except for creating new tasks. We aim to incorporate this functionality in the near future.

To run the UI, we also provide a Docker image. Below is an example of how you may deploy a UI using Docker compose; obviously, you may need to adjust the configuration to your own environment.

```
name: run-ui
services:
  ui:
    image: harbor2.vantage6.ai/infrastructure/ui:cotopaxi
    ports:
      - "8000:80"
    environment:
      - SERVER_URL=https://<url_to_my_server>
      - API_PATH=/api
```

Alternatively, you can also run the UI locally with Angular. In that case, follow the instructions on the UI Github page

The UI is not compatible with older versions (<3.3) of vantage6.

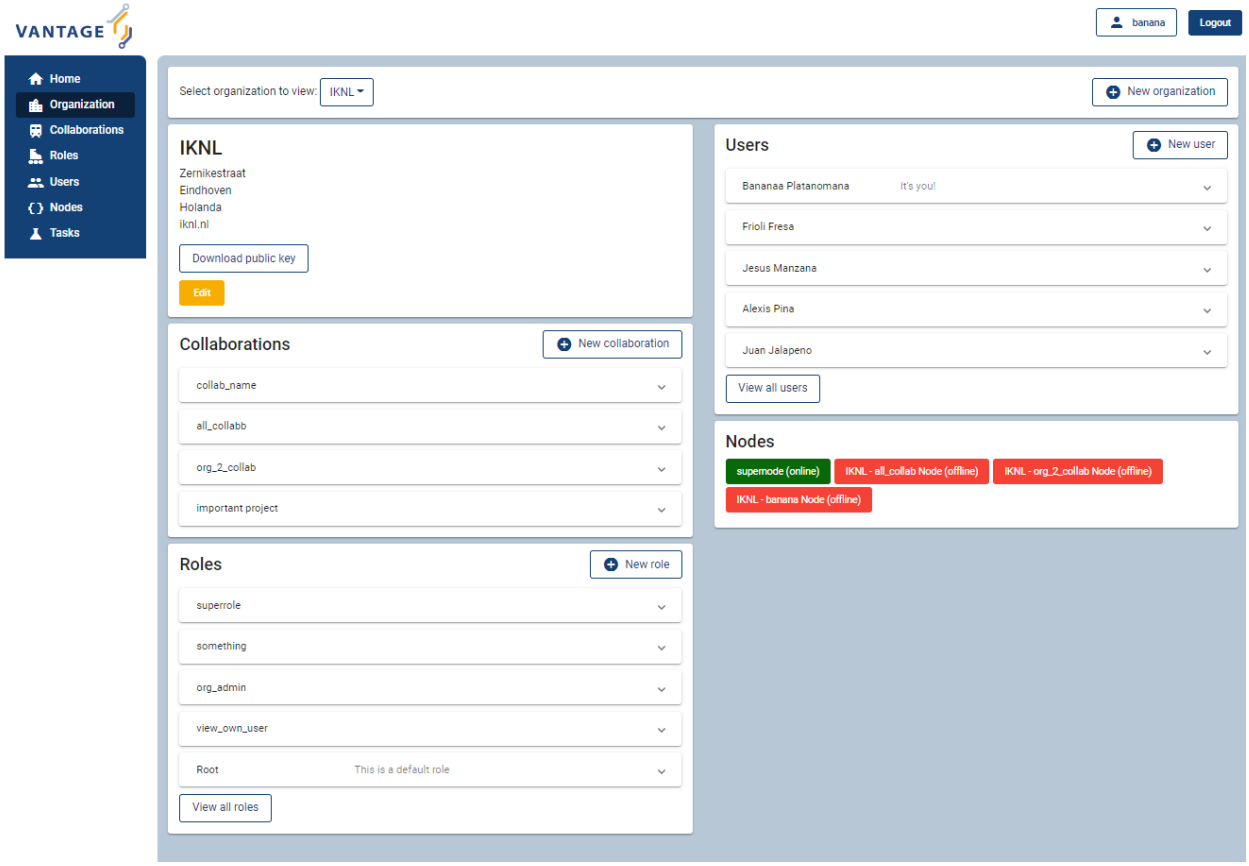


Fig. 4.4: Screenshot of the vantage6 UI

4.5.5.2 Docker registry

A Docker registry or repository provides storage and versioning for Docker images. Installing a private Docker registry is useful if you don't want to share your algorithms. Also, a private registry may have security benefits, for example, you can scan your images for vulnerabilities and you can limit the range of IP addresses that the node may access to its server and the private registry.

Harbor

Our preferred solution for hosting a Docker registry is Harbor. Harbor provides access control, a user interface and automated scanning on vulnerabilities.

Docker Hub

Docker itself provides a registry as a turn-key solution on Docker Hub. Instructions for setting it up can be found here: https://hub.docker.com/_/registry.

Note that some features of vantage6, such as timestamp based retrieval of the newest image, or multi-arch images, are not supported by the Docker Hub registry.

4.5.5.3 EduVPN

EduVPN is an optional server component that enables the use of algorithms that require node-to-node communication.

EduVPN provides an API for the OpenVPN server, which is required for automated certificate retrieval by the nodes. Like vantage6, it is an open source platform.

The following documentation shows you how to install EduVPN:

- Debian
- Centos
- Fedora

After the installation is done, you need to configure the server to:

1. Enable client-to-client communication. This can be achieved in the configuration file by the `clientToClient` setting (see here).
2. Do not block LAN communication (set `blockLan` to `false`). This allows your docker subnetworks to continue to communicate, which is required for vantage6 to function normally.
3. Enable port sharing (Optional). This may be useful if the nodes are behind a strict firewall. Port sharing allows nodes to connect to the VPN server only using outgoing `tcp/443`. Be aware that TCP meltdown can occur when using the TCP protocol for VPN.
4. Create an application account.

> **Warning:** EduVPN enables listening to multiple protocols (UDP/TCP) and ports at the same time. Be aware that all nodes need to be connected using the same protocol and port in order to communicate with each other.

Warning: The EduVPN server should usually be available to the public internet to allow all nodes to find it. Therefore, it should be properly secured, for example by closing all public ports (except http/https).

Additionally, you may want to explicitly allow *only* VPN traffic between nodes, and not between a node and the VPN server. You can achieve that by updating the firewall rules on your machine.

On Debian machines, these rules can be found in */etc/iptables/rules.v4* and */etc/iptables/rules.v6*, on CentOS, Red Hat Enterprise Linux and Fedora they can be found in */etc/sysconfig/iptables* and */etc/sysconfig/ip6tables*. You will have to do the following:

```
# In the firewall rules, below INPUT in the #SSH section, add this line
# to drop all VPN traffic with the VPN server as final destination:
-I INPUT -i tun+ -j DROP

# We only want to allow nodes to reach other nodes, and not other
# network interfaces available in the VPN.
# To achieve, replace the following rules:
-A FORWARD -i tun+ ! -o tun+ -j ACCEPT
-A FORWARD ! -i tun+ -o tun+ -j ACCEPT
# with:
-A FORWARD -i tun+ -o tun+ -j ACCEPT
-A FORWARD -i tun+ -j DROP
```

Example configuration

The following configuration makes a server listens to `TCP/443` only. Make sure you set `clientToClient` to `true` and `blockLan` to `false`. The `range` needs to be supplied to the node configuration files. Also note that the server configured below uses port-sharing.

```
// /etc/vpn-server-api/config.php
<?php

return [
    // List of VPN profiles
    'vpnProfiles' => [
        'internet' => [
            // The number of this profile, every profile per instance has a
            // unique number
            // REQUIRED
            'profileNumber' => 1,

            // The name of the profile as shown in the user and admin portals
            // REQUIRED
            'displayName' => 'vantage6 :: vpn service',

            // The IPv4 range of the network that will be assigned to clients
            // REQUIRED
            'range' => '10.76.0.0/16',

            // The IPv6 range of the network that will be assigned to clients
            // REQUIRED
            'range6' => 'fd58:63db:3245:d20d::/64',

            // The hostname the VPN client(s) will connect to
```

(continues on next page)

(continued from previous page)

```
            // REQUIRED
            'hostName' => 'eduvpn.vantage6.ai',

            // The address the OpenVPN processes will listen on
            // DEFAULT = '::'
            'listen' => '::',

            // The IP address to use for connecting to OpenVPN processes
            // DEFAULT = '127.0.0.1'
            'managementIp' => '127.0.0.1',

            // Whether or not to route all traffic from the client over the VPN
            // DEFAULT = false
            'defaultGateway' => true,

            // Block access to local LAN when VPN is active
            // DEFAULT = false
            'blockLan' => false,

            // IPv4 and IPv6 routes to push to the client, only used when
            // defaultGateway is false
            // DEFAULT = []
            'routes' => [],

            // IPv4 and IPv6 address of DNS server(s) to push to the client
            // DEFAULT  = []
            // Quad9 (https://www.quad9.net)
            'dns' => ['9.9.9.9', '2620:fe::fe'],

            // Whether or not to allow client-to-client traffic
            // DEFAULT = false
            'clientToClient' => true,

            // Whether or not to enable OpenVPN logging
            // DEFAULT = false
            'enableLog' => false,

            // Whether or not to enable ACLs for controlling who can connect
            // DEFAULT = false
            'enableAcl' => false,

            // The list of permissions to allow access, requires enableAcl to
            // be true
            // DEFAULT  = []
            'aclPermissionList' => [],

            // The protocols and ports the OpenVPN processes should use, MUST
            // be either 1, 2, 4, 8 or 16 proto/port combinations
            // DEFAULT = ['udp/1194', 'tcp/1194']
            'vpnProtoPorts' => [
                'tcp/1195',
            ],
```

(continues on next page)

(continued from previous page)

```
            // List the protocols and ports exposed to the VPN clients. Useful
            // for OpenVPN port sharing. When empty (or missing), uses list
            // from vpnProtoPorts
            // DEFAULT = []
            'exposedVpnProtoPorts' => [
                'tcp/443',
            ],

            // Hide the profile from the user portal, i.e. do not allow the
            // user to choose it
            // DEFAULT = false
            'hideProfile' => false,

            // Protect to TLS control channel with PSK
            // DEFAULT = tls-crypt
            'tlsProtection' => 'tls-crypt',
            //'tlsProtection' => false,
        ],
    ],

    // API consumers & credentials
    'apiConsumers' => [
        'vpn-user-portal' => '***',
        'vpn-server-node' => '***',
    ],
];
```

The following configuration snippet can be used to add an API user. The username and the `client_secret` have to be added to the vantage6-server configuration file.

```
...
'Api' => [
  'consumerList' => [
    'vantage6-user' => [
      'redirect_uri_list' => [
        'http://localhost',
      ],
      'display_name' => 'vantage6',
      'require_approval' => false,
      'client_secret' => '***'
    ]
  ]
...
```

4.5.5.4 RabbitMQ

RabbitMQ is an optional component that enables the server to handle more requests at the same time. This is important if a server has a high workload.

There are several options to host your own RabbitMQ server. You can run RabbitMQ in Docker or host RabbitMQ on Azure. When you have set up your RabbitMQ service, you can connect the server to it by adding the following to the server configuration:

```
rabbitmq_uri: amqp://<username>:<password>@<hostname>:5672/<vhost>
```

Be sure to create the user and vhost that you specify exist! Otherwise, you can add them via the RabbitMQ management console.

4.5.5.5 SMTP server

Some features of the server require an SMTP server to send emails. For example, the server can send an email to a user when they lost their password. There are many ways to set up an SMTP server, and we will not go into detail here. Just remember that you need to configure the server to use your SMTP server (see *All configuration options*).

4.5.6 Use

This section explains which commands are available to manage your server. It also explains how to set up a test server locally and how to manage resources via an IPython shell.

4.5.6.1 Quick start

To create a new server, run the command below. A menu will be started that allows you to set up a server configuration file.

```
v6 server new
```

For more details, check out the *Configure* section.

To run a server, execute the command below. The `--attach` flag will copy log output to the console.

```
v6 server start --name <your_server> --attach
```

Warning: When the server is run for the first time, the following user is created:

- username: root
- password: root

It is recommended to change this password immediately.

Finally, a server can be stopped again with:

```
v6 server stop --name <your_server>
```

4.5.6.2 Available commands

The following commands are available in your environment. To see all the options that are available per command use the `--help` flag, e.g. `v6 server start --help`.

Command	Description
`v6 server new`	Create a new server configuration file
`v6 server start`	Start a server
`v6 server stop`	Stop a server
`v6 server files`	List the files that a server is using
`v6 server attach`	Show a server's logs in the current terminal
`v6 server list`	List the available server instances
`v6 server shell`	Run a server instance python shell
`v6 server import`	Import server entities as a batch
`v6 server version`	Shows the versions of all the components of the running server

4.5.6.3 Local test setup

If the nodes and the server run at the same machine, you have to make sure that the node can reach the server.

Windows and MacOS

Setting the server IP to `0.0.0.0` makes the server reachable at your localhost (this is also the case when the dockerized version is used). In order for the node to reach this server, set the `server_url` setting to `host.docker.internal`.

> **Warning:** On the **M1** mac the local server might not be reachable from your nodes as `host.docker.internal` does not seem to refer to the host machine. Reach out to us on Discourse for a solution if you need this!

Linux

You should bind the server to `0.0.0.0`. In the node configuration files, you can then use `http://172.17.0.1`, assuming you use the default docker network settings.

4.5.6.4 Batch import

You can easily create a set of test users, organizations and collaborations by using a batch import. To do this, use the `v6 server import /path/to/file.yaml` command. An example `yaml` file is provided below.

You can download this file `here`.

```yaml
organizations:

  - name:       IKNL
    domain:     iknl.nl
    address1:   Godebaldkwartier 419
    address2:
    zipcode:    3511DT
    country:    Netherlands
    users:
      - username: admin
        firstname: admin
        lastname: robot
```

(continues on next page)

(continued from previous page)

```
        password: password
      - username: frank@iknl.nl
        firstname: Frank
        lastname: Martin
        password: password
      - username: melle@iknl.nl
        firstname: Melle
        lastname: Sieswerda
        password: password
    public_key:␣
↪LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUlJQ0lqQU5CZ2txaGtpRzl3MEJBUUVGQUFPQ0FnOEFNSUlDQ2dLQ0FnRUF2eU4wWW

  - name:       Small Organization
    domain:     small-organization.example
    address1:   Big Ambitions Drive 4
    address2:
    zipcode:    1234AB
    country:    Nowhereland
    users:
      - username: admin@small-organization.example
        firstname: admin
        lastname: robot
        password: password
      - username: stan
        firstname: Stan
        lastname: the man
        password: password
    public_key:␣
↪LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUlJQ0lqQU5CZ2txaGtpRzl3MEJBUUVGQUFPQ0FnOEFNSUlDQ2dLQ0FnRUF2eU4wWW

  - name:       Big Organization
    domain:     big-organization.example
    address1:   Offshore Accounting Drive 19
    address2:
    zipcode:    54331
    country:    Nowhereland
    users:
      - username: admin@big-organization.example
        firstname: admin
        lastname: robot
        password: password
    public_key:␣
↪LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUlJQ0lqQU5CZ2txaGtpRzl3MEJBUUVGQUFPQ0FnOEFNSUlDQ2dLQ0FnRUF2eU4wWW

collaborations:

  - name: ZEPPELIN
    participants:
      - name: IKNL
        api_key: 123e4567-e89b-12d3-a456-426614174001
      - name: Small Organization
        api_key: 123e4567-e89b-12d3-a456-426614174002
```

(continues on next page)

(continued from previous page)

```
    - name: Big Organization
      api_key: 123e4567-e89b-12d3-a456-426614174003
  tasks: ["hello-world"]
  encrypted: false

- name: PIPELINE
  participants:
    - name: IKNL
      api_key: 123e4567-e89b-12d3-a456-426614174004
    - name: Big Organization
      api_key: 123e4567-e89b-12d3-a456-426614174005
  tasks: ["hello-world"]
  encrypted: false

- name: SLIPPERS
  participants:
    - name: Small Organization
      api_key: 123e4567-e89b-12d3-a456-426614174006
    - name: Big Organization
      api_key: 123e4567-e89b-12d3-a456-426614174007
  tasks: ["hello-world", "hello-world"]
  encrypted: false
```

Warning: All users that are imported using `v6 server import` receive the superuser role. We are looking into ways to also be able to import roles. For more background info refer to this issue.

4.5.6.5 Testing

You can test the infrastructure via the `v6 dev` and `v6 test` commands. The purpose of `v6 dev` is to easily setup and run a test server accompanied by N nodes locally. For example, if you have $N = 10$ datasets to test a particular algorithm on, then you can spawn a server and 10 nodes with a single command.

The `v6 test` command is used to run the v6-diagnostics algorithm. You can run it on an existing network or create a `v6 dev` network and run the test on that in one go.

You can view all available commands in the table below, or alternatively, use `v6 dev --help`. By using `--help` with the individual commands (e.g. `v6 dev start-demo-network --help`), you can view more details on how to execute them.

Command	Description
`v6 dev create-demo-network`	Create a new network with server and nodes
`v6 dev start-demo-network`	Start the network
`v6 dev stop-demo-network`	Stop the network
`v6 dev remove-demo-network`	Remove the network completely
`v6 test feature-test`	Run the feature-tester algorithm on an existing network
`v6 test integration-test`	Create a dev network and run feature-tester on that network

An overview of the tests that the v6-diagnostics algorithm runs is given below.

- **Environment variables**: Reports the environment variables that are set in the algorithm container by the node instance. For example the location of the input, token and output files.

- **Input file**: Reports the contents of the input file. You can verify that the input set by the client is actually received by the algorithm.
- **Output file**: Writes 'test' to the output file and reads it back.
- **Token file**: Prints the contents of the token file. It should contain a JWT that you can decode and verify the payload. The payload contains information like the organization and collaboration ids.
- **Temporary directory**: Creates a file in the temporary directory. The temporary directory is a directory that is shared between all containers that share the same run id. This checks that the temporary directory is writable.
- **Local proxy**: Sends a request to the local proxy. The local proxy is used to reach the central server from the algorithm container. This is needed as parent containers need to be able to create child containers (=subtasks). The local proxy also handles encryption/decryption of the input and results as the algorithm container is not allowed to know the private key.
- **Subtask creation**: Creates a subtask (using the local proxy) and waits for the result.
- **Isolation test**: Checks if the algorithm container is isolated such that it can not reach the internet. It tests this by trying to reach google.nl, so make sure this is not a whitelisted domain when testing.
- **External port test**: Check that the algorithm can find its own ports. Algorithms can request a dedicated port for communication with other algorithm containers. The port that they require is stored in the Dockerfile using the `EXPORT` and `LABEL` keywords. For example:

```
LABEL p8888="port8"
EXPOSE 8888
```

  It however does not check that the application is actually listening on the port.
- **Database readable**: Check if the file-based database is readable.
- **VPN connection**: Check if an algorithm container on the node can reach other algorithm containers on other nodes *and* on the same node over the VPN network. This test will not succeed if the VPN connection is not set up - it can also be disabled with `v6 test feature-test --no-vpn`.

4.5.7 Configure

The vantage6-server requires a configuration file to run. This is a `yaml` file with a specific format.

The next sections describes how to configure the server. It first provides a few quick answers on setting up your server, then explains where your vantage6 configuration files are stored, and finally shows an example of all configuration file options.

4.5.7.1 How to create a configuration file

The easiest way to create an initial configuration file is via: `v6 server new`. This allows you to configure the basic settings. For more advanced configuration options, which are listed below, you can view the *example configuration file*.

4.5.7.2 Where is my configuration file?

To see where your configuration file is located, you can use the following command

```
v6 server files
```

> **Warning:** This command will only work for if the server has been deployed using the `v6` commands.
>
> Also, note that on local deployments you may need to specify the `--user` flag if you put your configuration file in the *user folder*.

You can also create and edit this file manually.

4.5.7.3 All configuration options

The following configuration file is an example that intends to list all possible configuration options.

You can download this file here: `server_config.yaml`

```yaml
# Human readable description of the server instance. This is to help
# your peers to identify the server
description: Test

# Human readable name of the server instance. This can help users identify
# the server quickly. It's used for example in the TOTP issuer for 2FA.
server_name: demo

# IP adress to which the server binds. In case you specify 0.0.0.0
# the server listens on all interfaces
ip: 0.0.0.0

# Port to which the server binds
port: 5000

# API path prefix. (i.e. https://yourdomain.org/api_path/<endpoint>). In the
# case you use a referse proxy and use a subpath, make sure to include it
# here also.
api_path: /api

# Let the server know where it is hosted. This is used as a setting to
# communicate back and forth with other vantage6 components such as the
# algorithm store.
# Example for the cotopaxi server
server_url: https://cotopaxi.vantage6.ai
# Example for running the server locally with default settings:
# server_url: http://localhost:5000

# The URI to the server database. This should be a valid SQLAlchemy URI,
# e.g. for an Sqlite database: sqlite:///database-name.sqlite,
# or Postgres: postgresql://username:password@172.17.0.1/database).
uri: sqlite:///test.sqlite
```

(continues on next page)

(continued from previous page)

```
# This should be set to false in production as this allows to completely
# wipe the database in a single command. Useful to set to true when
# testing/developing.
allow_drop_all: True

# Enable or disable two-factor authentication. If enabled, users will be
# presented with a QR-code to scan with their phone the first time they log in.
two_factor_auth: true

# The secret key used to generate JWT authorization tokens. This should
# be kept secret as others are able to generate access tokens if they
# know this secret. This parameter is optional. In case it is not
# provided in the configuration it is generated each time the server
# starts. Thereby invalidating all previous distributed keys.
# OPTIONAL
jwt_secret_key: super-secret-key! # recommended but optional

# Settings for the logger
logging:
  # Controls the logging output level. Could be one of the following
  # levels: CRITICAL, ERROR, WARNING, INFO, DEBUG, NOTSET
  level: DEBUG

  # Filename of the log-file, used by RotatingFileHandler
  file: test.log

  # Whether the output is shown in the console or not
  use_console: True

  # The number of log files that are kept, used by RotatingFileHandler
  backup_count: 5

  # Size in kB of a single log file, used by RotatingFileHandler
  max_size: 1024

  # format: input for logging.Formatter,
  format: "%(asctime)s - %(name)-14s - %(levelname)-8s - %(message)s"
  datefmt: "%Y-%m-%d %H:%M:%S"

  # (optional) set the individual log levels per logger name, for example
  # mute some loggers that are too verbose.
  loggers:
    - name: urllib3
      level: warning
    - name: socketIO-client
      level: warning
    - name: engineio.server
      level: warning
    - name: socketio.server
      level: warning
    - name: sqlalchemy.engine
      level: warning
```

(continues on next page)

(continued from previous page)

```
    - name: requests_oauthlib.oauth2_session
      level: warning

# Additional debug flags
debug:
  # Set to `true` to enable debug mode for the socketio server
  socketio: false

  # Set to `true` to enable debug mode in the Flask app
  flask: false

# Configure a smtp mail server for the server to use for administrative
# purposes. e.g. allowing users to reset their password.
# OPTIONAL
smtp:
  port: 587
  server: smtp.yourmailserver.example.com
  # credentials for authenticating with the SMTP server
  username: your-username
  password: super-secret-password
  # email address to send emails from (header)
  # (defaults to noreply@vantage6.ai)
  email_from: noreply@example.com

# Set an email address you want to direct your users to for support
# (defaults to support@vantage6.ai)
support_email: your-support@example.com

# set how long reset token provided via email are valid (default 1 hour)
email_token_validity_minutes: 60

# set how long tokens and refresh tokens are valid (default 6 and 48
# hours, respectively)
token_expires_hours: 6
refresh_token_expires_hours: 48

# If you have a server with a high workload, it is recommended to use
# multiple server instances (horizontal scaling). If you do so, you also
# need to set up a RabbitMQ message service to ensure that the communication
# between the server and the nodes is handled properly. Then, fill out the
# RabbitMQ connection URI below to connect the server to it. Also, set the
# start_with_server flag to true to start RabbitMQ when you start the server.
rabbitmq:
  uri: amqp://myuser:mypassword@myhostname:5672/myvhost
  start_with_server: false

# If algorithm containers need direct communication between each other
# the server also requires a VPN server. (!) This must be a EduVPN
# instance as vantage6 makes use of their API (!)
# OPTIONAL
vpn_server:
  # the URL of your VPN server
```

(continues on next page)

(continued from previous page)

```
  url: https://your-vpn-server.ext

  # OATH2 settings, make sure these are the same as in the
  # configuration file of your EduVPN instance
  redirect_url: http://localhost
  client_id: your_VPN_client_user_name
  client_secret: your_VPN_client_user_password

  # Username and password to acccess the EduVPN portal
  portal_username: your_eduvpn_portal_user_name
  portal_userpass: your_eduvpn_portal_user_password

# specify custom Docker images to use for starting the different
# components.
# OPTIONAL
images:
  server: harbor2.vantage6.ai/infrastructure/server:cotopaxi
  ui: harbor2.vantage6.ai/infrastructure/ui:cotopaxi

# options for starting the User Interface when starting the server
ui:
  # set this to true to start the UI when starting the server with
  # `v6 server start`
  enabled: true

  # port at which the UI will be available on your local machine
  port: 3456

# set password policies for the server
password_policy:
  # maximum number of failed login attempts before the user is locked out for
  # a certain amount of time. Default is 5.
  max_failed_attempts: 5

  # number of minutes the user is locked out after the maximum number of failed
  # login attempts is reached. Default is 15.
  inactivation_minutes: 15

  # number of minutes to wait between emails sent to the user for each of the following␣
↪events:
  #  - their account has been blocked (max login attempts exceeded)
  #  - a password reset request via email
  #  - a 2FA reset request via email
  # (these events have an independent timer). Default is 60.
  between_user_emails_minutes: 60

# set up with which origins the server should allow CORS requests. The default
# is to allow all origins. If you want to restrict this, you can specify a list
# of origins here. Below are examples to allow requests from the Cotopaxi UI, and
#  port 3456 on localhost
cors_allowed_origins:
  - https://portal.cotopaxi.vantage6.ai
```

(continues on next page)

(continued from previous page)

```
  - http://localhost:3456

# set up which algorithm stores are available for all collaborations in the server.
# In the example below, the vantage6 community algorithm store is made available to
# this server. Note that the 'server_url' *has* to be set in the configuration file
# for this to work (it is required to communicate with the algorithm store what the
# server's address is).
algorithm_stores:
  # Each store should have a name and a URL.
  - name: Community store
    url: https://store.cotopaxi.vantage6.ai

# development mode settings. Only use when running both the server and the algorithm
# store that it communicates with locally
dev:
  # Specify the URI to the host. This is used to generate the correct URIs to
  # communicate with the algorithm store. On Windows and Mac, you can use the special
  # hostname `host.docker.internal` to refer to the host machine. On Linux, you
  # should normally use http://172.17.0.1.
  host_uri: http://host.docker.internal
```

4.5.7.4 Configuration file location

The directory where to store the configuration file depends on your operating system (OS). It is possible to store the configuration file at **system** or at **user** level. At the user level, configuration files are only available for your user. By default, server configuration files are stored at **system** level.

The default directories per OS are as follows:

OS	System	User
Windows	`C:\ProgramData\vantage\server`	`C:\Users\<user>\AppData\Local\vantage\server`
MacOS	`/Library/Application/Support/vantage6/server`	`/Users/<user>/Library/Application Support/vantage6/server`
Linux	`/etc/xdg/vantage6/server/`	`/home/<user>/.config/vantage6/server/`

Warning: The command `v6 server` looks in certain directories by default. It is possible to use any directory and specify the location with the `--config` flag. However, note that using a different directory requires you to specify the `--config` flag every time!

Similarly, you can put your server configuration file in the user folder by using the `--user` flag. Note that in that case, you have to specify the `--user` flag for all `v6 server` commands.

4.5.7.5 Logging

Logging is enabled by default. To configure the logger, look at the `logging` section in the example configuration in *All configuration options*.

Useful commands:

1. `v6 server files`: shows you where the log file is stored
2. `v6 server attach`: show live logs of a running server in your current console. This can also be achieved when starting the server with `v6 server start --attach`

4.5.8 Permission management

Almost everything in the vantage6 server is under role-based access control: not everyone is allowed to access everything.

4.5.8.1 Authentication types

There are three types of entities that can attempt to use the vantage6 server: users, nodes and algorithm containers. Not every resource is available to all three entities. In the vantage6 server code, this is ensured by using so-called decorators that are placed on the API endpoints. These decorators check if the entity that is trying to access the endpoint is allowed to do so. For example, you may see the following decorators on an endpoint:

- `@only_for(('user', 'container'))`: only accessible to users and algorithm containers
- `@with_user`: only accessible to users

These decorators ensure that only authenticated entities of the right type can enter the endpoint. For example, only users can create new users or organizations, and only nodes are allowed to update the results of a task (the algorithm itself cannot do this as it exits when it finishes, and users are not allowed to meddle with results).

4.5.8.2 Permission rules

The fact that users are allowed to create new organizations, does not mean that all users are allowed to do so. There are permission rules that determine what every user is allowed to do. These rules are assigned to a user by another user. A user that creates a new user is never allowed to give the new user more permissions than they have themselves.

Nodes and algorithm containers all have the same permissions, but for specific situations there are specific checks. For instance, nodes are only allowed to update their own results, and not those of other nodes.

The following rules are defined:

The rules have an operation, a scope, and a resource that they work on. For instance, a rule with operation 'View', scope 'Organization' and resource 'Task', will allow a user to view all tasks of their own organization.

There are six operations (view, edit, create, delete, send and receive). The first four correspond to GET, PATCH, CREATE and DELETE requests, respectively. The last two allow users to send and receive data via socket events. For example, sending events would allow them to kill tasks that are running on a node.

The scopes are:

- Global: all resources of all organizations
- Organization: resources of the user's own organization
- Collaboration: resources of all organizations that the user's organization is in a collaboration with

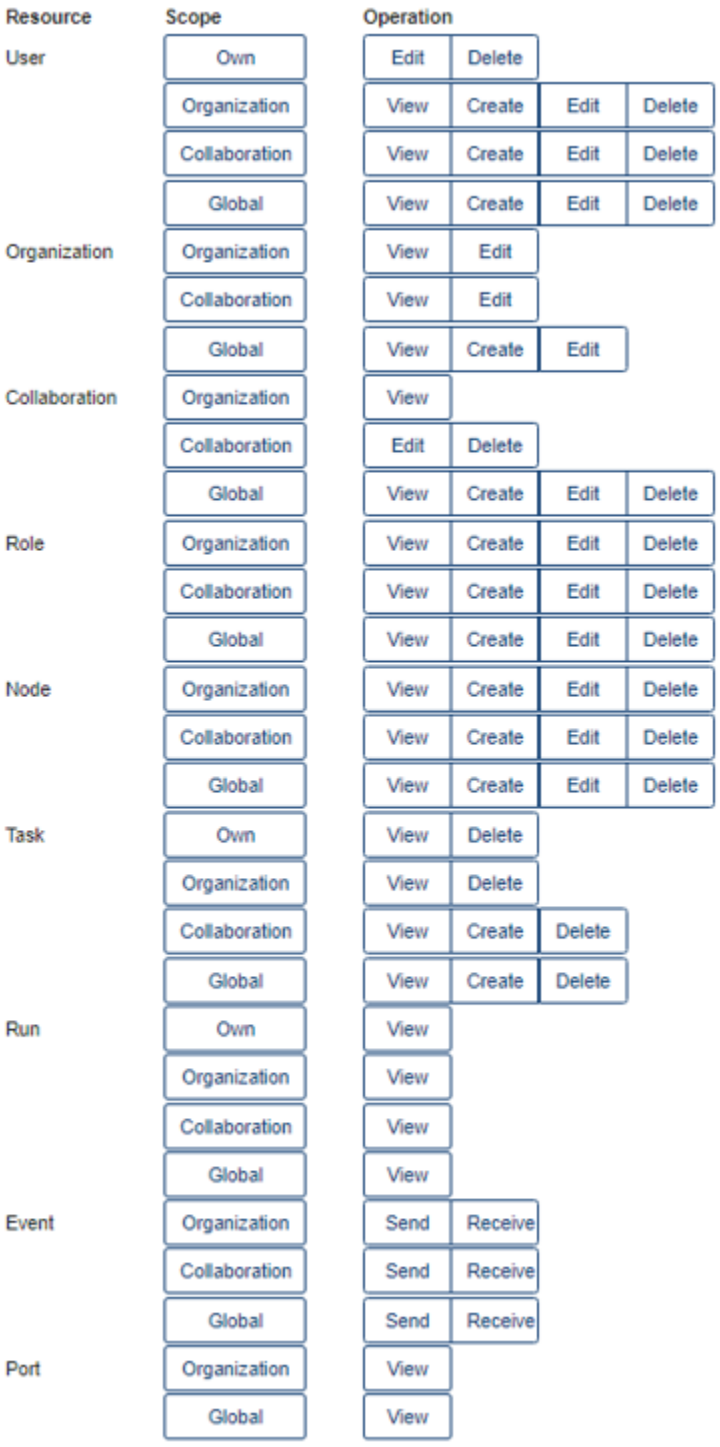


Fig. 4.5: The rules that are available per resource, scope, and operation. For example, the first rule with resource 'User', scope 'Own' and operation 'View' will allow a user to view their own user details.

- Own: these are specific to the user endpoint. Permits a user to see/edit their own user, but not others within the organization.

A user may be assigned anywhere between zero and all of the rules.

Note: When you create a new server, the first time it is started, a new user 'root' is created that has all permissions. This user is meant to be used to create the first users and organizations.

4.5.8.3 Roles

To make it easier to assign permissions to users, there are roles. A role is simply a set of rules. When a user is assigned a role, they are assigned all the rules that are part of that role.

The permission structure of vantage6 allows for a lot of flexibility. However, especially for beginning users, it can be a bit daunting to set up all the permissions. Therefore, there are some default roles that can be used to quickly set up a server. These roles are, in descending order of permissions:

- Root: all permissions
- Collaboration Admin: can do almost everything for all organizations in collaborations that they are a member of, e.g. create new users but not delete the entire collaboration
- Organization Admin: can do everything for their own organization
- Researcher: can view the organization's resources and create tasks
- Viewer: can only view the organization's resources

We do recommend that you review the permissions of these roles before using them in your own project.

4.5.9 Shell

Warning: Using the server shell is not recommended. The shell is outdated and superseded by other tools. The shell offers a server admin the ability to manage the server entities, but does not offer any validation of the input. Therefore, it is easy to break the server by using the shell.

Instead, we recommend using the *user interface*, the *Python client* or the *API*.

The commands in this page have not been updated to match version 4.0.

The shell allows a server admin to manage all server entities. To start the shell, use `v6 server shell [options]`.

In the next sections the different database models that are available are explained. You can retrieve any record and edit any property of it. Every `db.` object has a `help()` method which prints some info on what data is stored in it (e.g. `db.Organization.help()`).

Note: Don't forget to call `.save()` once you are done editing an object.

4.5.9.1 Organizations

Note: Organizations have a public key that is used for end-to-end encryption. This key is automatically created and/or uploaded by the node the first time it runs.

To store an organization you can use the `db.Organization` model:

```
# create new organiztion
organization = db.Organization(
    name="IKNL",
    domain="iknl.nl",
    address1="Zernikestraat 29",
    address2="Eindhoven",
    zipcode="5612HZ",
    country="Netherlands"
)

# store organization in the database
organization.save()
```

Retrieving organizations from the database:

```
# get all organizations in the database
organizations = db.Organization.get()

# get organization by its unique id
organization = db.Organization.get(1)

# get organization by its name
organization = db.Organization.get_by_name("IKNL")
```

A lot of entities (e.g. users) at the server are connected to an organization. E.g. you can see which (computation) tasks are issued by the organization or see which collaborations it is participating in.

```
# retrieve organization from which we want to know more
organization = db.Organization.get_by_name("IKNL")

# get all collaborations in which the organization participates
collaborations = organization.collaborations

# get all users from the organization
users = organization.users

# get all created tasks (from all users)
tasks = organization.created_tasks

# get the runs of all these tasks
runs = organization.runs

# get all nodes of this organization (for each collaboration
# an organization participates in, it needs a node)
nodes = organization.nodes
```

4.5.9.2 Roles and Rules

A user can have multiple roles and rules assigned to them. These are used to determine if the user has permission to view, edit, create or delete certain resources using the API. A role is a collection of rules.

```
# display all available rules
db.Rule.get()

# display rule 1
db.Rule.get(1)

# display all available roles
db.Role.get()

# display role 3
db.Role.get(3)

# show all rules that belong to role 3
db.Role.get(3).rules

# retrieve a certain rule from the DB
rule = db.Rule.get_by_("node", Scope, Operation)

# create a new role
role = db.Role(name="role-name", rules=[rule])
role.save()

# or assign the rule directly to the user
user = db.User.get_by_username("some-existing-username")
user.rules.append(rule)
user.save()
```

4.5.9.3 Users

Users belong to an organization. So if you have not created any *Organizations* yet, then you should do that first. To create a user you can use the `db.User` model:

```
# first obtain the organization to which the new user belongs
org = db.Organization.get_by_name("IKNL")

# obtain role 3 to assign to the new user
role_3 = db.Role.get(3)

# create the new users, see section Roles and Rules on how to
# deal with permissions
new_user = db.User(
    username="root",
    password="super-secret",
    firstname="John",
    lastname="Doe",
    roles=[role_3],
    rules=[],
```

(continues on next page)

(continued from previous page)

```
    organization=org
)

# store the user in the database
new_user.save()
```

You can retrieve users in the following ways:

```
# get all users
db.User.get()

# get user 1
db.User.get(1)

# get user by username
db.User.get_by_username("root")

# get all users from the organization IKNL
db.Organization.get_by_name("IKNL").users
```

To modify a user, simply adjust the properties and save the object.

```
user = db.User.get_by_username("some-existing-username")

# update the firstname
user.firstname = "Brandnew"

# update the password; it is automatically hashed.
user.password = "something-new"

# store the updated user in the database
user.save()
```

4.5.9.4 Collaborations

A collaboration consists of one or more organizations. To create a collaboration you need at least one but preferably multiple *Organizations* in your database. To create a collaboration you can use the `db.Collaboration` model:

```
# create a second organization to collaborate with
other_organization = db.Organization(
    name="IKNL",
    domain="iknl.nl",
    address1="Zernikestraat 29",
    address2="Eindhoven",
    zipcode="5612HZ",
    country="Netherlands"
)
other_organization.save()

# get organization we have created earlier
iknl = db.Organization.get_by_name("IKNL")
```

(continues on next page)

(continued from previous page)

```
# create the collaboration
collaboration = db.Collaboration(
    name="collaboration-name",
    encrypted=False,
    organizations=[iknl, other_organization]
)

# store the collaboration in the database
collaboration.save()
```

Tasks, nodes and organizations are directly related to collaborations. We can obtain these by:

```
# obtain a collaboration which we like to inspect
collaboration = db.Collaboration.get(1)

# get all nodes
collaboration.nodes

# get all tasks issued for this collaboration
collaboration.tasks

# get all organizations
collaboration.organizations
```

Warning: Setting the encryption to False at the server does not mean that the nodes will send encrypted results. This is only the case if the nodes also agree on this setting. If they don't, you will receive an error message.

4.5.9.5 Nodes

Before nodes can login, they need to exist in the server's database. A new node can be created as follows:

```
# we'll use a uuid as the API-key, but you can use anything as
# API key
from uuid import uuid4

# nodes always belong to an organization *and* a collaboration,
# this combination needs to be unique!
iknl = db.Organization.get_by_name("IKNL")
collab = iknl.collaborations[0]

# generate and save
api_key = str(uuid4())
print(api_key)

node = db.Node(
    name = f"IKNL Node - Collaboration {collab.name}",
    organization = iknl,
    collaboration = collab,
    api_key = api_key
```

(continues on next page)

(continued from previous page)

```
)

# save the new node to the database
node.save()
```

Note: API keys are hashed before stored in the database. Therefore you need to save the API key immediately. If you lose it, you can reset the API key later via the shell, API, client or UI.

4.5.9.6 Tasks and Results

Warning: Tasks(/results) created from the shell are not picked up by nodes that are already running. The signal to notify them of a new task cannot be emitted this way. We therefore recommend sending tasks via the Python client.

A task is intended for one or more organizations. For each organization the task is intended for, a corresponding (initially empty) run should be created. Each task can have multiple runs, for example a run from each organization.

```
# obtain organization from which this task is posted
iknl = db.Organization.get_by_name("IKNL")

# obtain collaboration for which we want to create a task
collaboration = db.Collaboration.get(1)

# obtain the next job_id. Tasks sharing the same job_id
# can share the temporary volumes at the nodes. Usually this
# job_id is assigned through the API (as the user is not allowed
# to do so). All tasks from a master-container share the
# same job_id
job_id = db.Task.next_job_id()

task = db.Task(
    name="some-name",
    description="some human readable description",
    image="docker-registry.org/image-name",
    collaboration=collaboration,
    job_id=job_id,
    database="default",
    init_org=iknl,
)
task.save()

# input the algorithm container (docker-registry.org/image-name)
# expects
input_ = {
}

import datetime

# now create a Run model for each organization within the
```

(continues on next page)

(continued from previous page)

```
# collaboration. This could also be a subset
for org in collaboration.organizations:
    res = db.Run(
        input=input_,
        organization=org,
        task=task,
        assigned_at=datetime.datetime.now()
    )
    res.save()
```

Tasks can have a child/parent relationship. Note that the `job_id` is for parent and child tasks the same.

```
# get a task to which we want to create some
# child tasks
parent_task = db.Task.get(1)

child_task = db.Task(
    name="some-name",
    description="some human readable description",
    image="docker-registry.org/image-name",
    collaboration=collaboration,
    job_id=parent_task.job_id,
    database="default",
    init_org=iknl,
    parent=parent_task
)
child_task.save()
```

Note: Tasks that share a `job_id` have access to the same temporary folder at the node. This allows for multi-stage algorithms.

Obtaining algorithm Runs:

```
# obtain all Runs
db.Run.get()

# obtain only completed runs
[run for run in db.Run.get() if run.complete]

# obtain run by its unique id
db.Run.get(1)
```

4.6 Algorithm store admin guide

This section shows you how you can set up your own vantage6 algorithm store. First, we discuss the requirements for hosting this server, then guide you through the installation process. Finally, we explain how to configure and start your algorithm store.

4.6.1 Introduction

4.6.1.1 What is an algorithm store?

When using vantage6, it is important to know which algorithms are available to you. This is why vantage6 has algorithm stores. An algorithm store contains metadata about the algorithms so that you can easily find the algorithm you need, and know how to use it.

There is a community algorithm store, which is by default available to all collaborations. This store is maintained by the vantage6 community. You can also create your own algorithm store. This allows you to create a private algorithm store, which is only available to your own collaborations. .. # TODO add link to creating algorithm store .. TODO add links to an architectural page where algorithm store is explained

4.6.1.2 Linking algorithm stores

Algorithm stores can be linked to a vantage6 server or to a specific collaboration on a server. If an algorithm store is linked to a server, the algorithms in the store are available to all collaborations on that server. If an algorithm store is linked to a collaboration, the algorithms in the store are only available to that collaboration.

Users can link algorithm stores to a collaboration if they have permission to modify that collaboration. Algorithm stores can only be linked to a server by users that have permission to modify all collaborations on the server.

4.6.2 Requirements

Note: This section is almost the same as the *node* and *server* sections - their requirements are very similar.

Below are the minimal requirements for vantage6 infrastructure components. Note that these are recommendations: it may also work on other hardware, operating systems, versions of Python etc. (but they are not tested as much).

Hardware

- x86 CPU architecture + virtualization enabled
- 1 GB memory
- 50GB+ storage
- Stable and fast (1 Mbps+ internet connection)

Note that the algorithm store's IP address should also be reachable by users and the central server. This will usually be a public IP address.

Software

- Operating system: Ubuntu 18.04+
- Python
- Docker

Note: For the server, Ubuntu is highly recommended. It is possible to run a development server (using *v6 server start*) on Windows or MacOS, but for production purposes we recommend using Ubuntu.

Warning: The hardware requirements of the node also depend on the algorithms that the node will run. For example, you need much less compute power for a descriptive statistical algorithm than for a machine learning model.

4.6.2.1 Python

Installation of any of the vantage6 packages requires Python 3.10. For installation instructions, see python.org, anaconda.com or use the package manager native to your OS and/or distribution.

Note: We recommend you install vantage6 in a new, clean Python (Conda) environment.

Higher versions of Python (3.11+) will most likely also work, as might lower versions (3.8 or 3.9). However, we develop and test vantage6 on version 3.10, so that is the safest choice.

Warning: Note that Python 3.10 is only used in vantage6 versions 3.8.0 and higher. In lower versions, Python 3.7 is required.

4.6.2.2 Docker

Docker facilitates encapsulation of applications and their dependencies in packages that can be easily distributed to diverse systems. Algorithms are stored in Docker images which nodes can download and execute. Besides the algorithms, both the node and server are also running from a docker container themselves.

Please refer to this page on how to install Docker. To verify that Docker is installed and running you can run the `hello-world` example from Docker.

```
docker run hello-world
```

Warning: Note that for **Linux**, some post-installation steps may be required. Vantage6 needs to be able to run docker without `sudo`, and these steps ensure just that.

For Windows, if you are using Docker Desktop, it may be preferable to limit the amount of memory Docker can use - in some cases it may otherwise consume much memory and slow down the system. This may be achieved as described here.

Note:

- Always make sure that Docker is running while using vantage6!
- We recommend to always use the latest version of Docker.

4.6.3 Install

4.6.3.1 Local (test) Installation

To install the vantage6 algorithm store, make sure you have met the *requirements*. Then, we provide a command-line interface (CLI) with which you can manage your algorithm store. The CLI is a Python package that can be installed using pip. We always recommend to install the CLI in a virtual environment or a conda environment.

Run this command to install the CLI in your environment:

```
pip install vantage6
```

Or if you want to install a specific version:

```
pip install vantage6==x.y.z
```

You can verify that the CLI has been installed by running the command `v6 --help`. If that prints a list of commands, the installation is completed.

The algorithm store software itself will be downloaded when you start the algorithm store for the first time.

4.6.3.2 Hosting your algorithm store

To host your algorithm store, we recommend to use the Docker image we provide: `harbor2.vantage6.ai/infrastructure/algorithm-store`. Running this docker image will start the server. Check the *deployment* section for deployment examples.

Note: We recommend to use the latest version. Should you have reasons to deploy an older `VERSION`, use the image `harbor2.vantage6.ai/infrastructure/algorithm-store:<VERSION>`.

4.6.4 Deploy

The deployment of the algorithm store is highly similar to the deployment of the vantage6 server. Both are Flask applications that are structured very similarly.

The algorithm store's deployment is a bit simpler because it does not use socketIO. This means that you don't have to take into account that the websocket channels should be open, and makes it easier to horizontally scale the application.

4.6.4.1 NGINX

The algorithm store can be deployed with a similar nginx script as detailed for the *server*.

One note is that for the algorithm store, the subpath is fixed at */api*, so be sure to set that in the subpath block.

4.6.4.2 Docker compose

The algorithm store can be started with `v6 algorithm-store start`, but in most deployment scenarios a docker-compose file is used. Below is an example of a docker-compose file that can be used to deploy the algorithm store.

```
services:
  vantage6-algorithm-store:
    image: harbor2.vantage6.ai/infrastructure/algorithm-store:cotopaxi
    ports:
    - "8000:5000"
    volumes:
    - /path/to/my/server.yaml:/mnt/config.yaml
    command: ["/bin/bash", "-c", "/vantage6/vantage6-algorithm-store/server.sh"]
```

4.6.5 Use

This section explains which commands are available to manage your algorithm store. These can be used to set up a test server locally. To deploy a server, see the *deployment* section.

4.6.5.1 Quick start

To create a new algorithm store, run the command below. A menu will be started that allows you to set up an algorithm store configuration file.

```
v6 algorithm-store new
```

For more details, check out the *Configure* section.

To run an algorithm store, execute the command below. The `--attach` flag will copy log output to the console.

```
v6 algorithm-store start --name <your_store> --attach
```

Finally, a server can be stopped again with:

```
v6 algorithm-store stop --name <your_store>
```

4.6.5.2 Available commands

The following commands are available in your environment. To see all the options that are available per command use the `--help` flag, e.g. `v6 server start --help`.

Table 4.1: Available commands

Command	Description
`v6 algorithm-store new`	Create a new algorithm store configuration file
`v6 algorithm-store start`	Start an algorithm store
`v6 algorithm-store stop`	Stop an algorithm store
`v6 algorithm-store files`	List the files that an algorithm store is using
`v6 algorithm-store attach`	Show an algorithm store's logs in the current terminal
`v6 algorithm-store list`	List the available algorithm store instances

4.6.6 Configure

The algorithm store requires a configuration file to run. This is a `yaml` file with a specific format.

The next sections describes how to configure the algorithm store. It first provides a few quick answers on setting up your store, then shows an example of all configuration file options, and finally explains where your configuration files are stored.

4.6.6.1 How to create a configuration file

The easiest way to create an initial configuration file is via: `v6 algorithm-store new`. This allows you to configure the basic settings. For more advanced configuration options, which are listed below, you can view the *example configuration file*.

4.6.6.2 Where is my configuration file?

To see where your configuration file is located, you can use the following command

```
v6 algorithm-store files
```

> **Warning:** This command will only work for if the algorithm store has been deployed using the `v6` commands.
>
> Also, note that on local deployments you may need to specify the `--user` flag if you put your configuration file in the *user folder*.

You can also create and edit this file manually.

4.6.6.3 All configuration options

The following configuration file is an example that intends to list all possible configuration options.

You can download this file here: `algorithm_store_config.yaml`

```yaml
# Human readable description of the algorithm store instance. This is to help
# your peers to identify the store
description: Test

# IP adress to which the algorithm store server binds. In case you specify 0.0.0.0
# the server listens on all interfaces
ip: 0.0.0.0

# Port to which the algorithm store binds
port: 5000

# The URI to the algorithm store database. This should be a valid SQLAlchemy URI,
# e.g. for an Sqlite database: sqlite:///database-name.sqlite,
# or Postgres: postgresql://username:password@172.17.0.1/database).
uri: sqlite:///test.sqlite

# This should be set to false in production as this allows to completely
# wipe the database in a single command. Useful to set to true when
```

(continues on next page)

(continued from previous page)

```
# testing/developing.
allow_drop_all: True

# Settings for the logger
logging:
  # Controls the logging output level. Could be one of the following
  # levels: CRITICAL, ERROR, WARNING, INFO, DEBUG, NOTSET
  level: DEBUG

  # Filename of the log-file, used by RotatingFileHandler
  file: test.log

  # Whether the output is shown in the console or not
  use_console: True

  # The number of log files that are kept, used by RotatingFileHandler
  backup_count: 5

  # Size in kB of a single log file, used by RotatingFileHandler
  max_size: 1024

  # format: input for logging.Formatter,
  format: "%(asctime)s - %(name)-14s - %(levelname)-8s - %(message)s"
  datefmt: "%Y-%m-%d %H:%M:%S"

  # (optional) set the individual log levels per logger name, for example
  # mute some loggers that are too verbose.
  loggers:
    - name: urllib3
      level: warning
    - name: sqlalchemy.engine
      level: warning

# Additional debug flags
debug:
  # Set to `true` to enable debug mode in the Flask app
  flask: false

# Settings for the algorithm store's policies
policies:
  # Set to `true` to allow anyone to view and run the algorithms in the store.
  algorithms_open: true
  # Set to `true` to allow any user from whitelisted vantage6 servers to view and run
  # the algorithms in the store. Superfluous if `algorithms_open` is set to `true`.
  # If this is set to `false`, only users with specific roles in the algorithm store
  # can view the algorithms in the store.
  algorithms_open_to_whitelisted: true

# development mode settings. Only use when running both the algorithm store and
# the server that it communicates with locally
dev:
  # Specify the URI to the host. This is used to generate the correct URIs to
```

(continues on next page)

(continued from previous page)

```
  # communicate with the server. On Windows and Mac, you can use the special
  # hostname `host.docker.internal` to refer to the host machine. On Linux, you
  # should normally use http://172.17.0.1.
  host_uri: http://host.docker.internal

# Provide an initial root user for the algorithm store. This user will be created
# when the store is started for the first time. The root user has full access to
# the store and can create other users. The root user should be a reference to an
# existing user in a vantage6 server.
root_user:
  # URI to the vantage6 server
  v6_server_uri: http://localhost:5000/api
  # username of the vantage6 server's user you want to make root in the algorithm store
  username: root
```

4.6.6.4 Configuration file location

The directory where to store the configuration file depends on your operating system (OS). It is possible to store the configuration file at **system** or at **user** level. At the user level, configuration files are only available for your user. By default, algorithm store configuration files are stored at **system** level.

The default directories per OS are as follows:

OS	System	User
Windows	`C:\ProgramData\vantage\algorithm-store`	`C:\Users\<user>\AppData\Local\vantage\algorithm-store`
MacOS	`/Library/Application/Support/vantage6/algorithm-store`	`/Users/<user>/Library/Application Support/vantage6/algorithm-store`
Linux	`/etc/xdg/vantage6/algorithm-store/`	`/home/<user>/.config/vantage6/algorithm-store/`

Warning: The command `v6 algorithm-store` looks in certain directories by default. It is possible to use any directory and specify the location with the `--config` flag. However, note that using a different directory requires you to specify the `--config` flag every time!

Similarly, you can put your algorithm store configuration file in the user folder by using the `--user` flag. Note that in that case, you have to specify the `--user` flag for all `v6 algorithm-store` commands.

4.6.6.5 Logging

Logging is enabled by default. To configure the logger, look at the `logging` section in the example configuration in *All configuration options*.

Useful commands:

1. `v6 algorithm-store files`: shows you where the log file is stored
2. `v6 algorithm-store attach`: show live logs of a running store in your current console. This can also be achieved when starting the store with `v6 algorithm-store start --attach`

4.7 Algorithm Development

This section helps you to develop MPC and FL algorithms that are compatible with vantage6. You are **not** going to find a list of algorithms here or help on how to use them. In the *Introduction*, the basic concepts and interface between node and algorithm are explained. Then, in the *Algorithm development step-by-step guide*, the algorithm development process is explained. Finally, in the *Classic Tutorial*, an FL algorithm is build from scratch.

> **Warning:** Note that the classic tutorial is not outdated, and the commands may be wrong. Nevertheless, the concepts are still valid and the tutorial is still useful to get a grasp of the vantage6 framework.

4.7.1 Algorithm concepts

This page details the concepts used in vantage6 algorithms. Understanding these concepts is helpful when you to create your own algorithms. A guide to develop your own algorithms can be found in the *Algorithm development step-by-step guide*.

Algorithms are executed at the vantage6 node. The node receives a computation task from the vantage6-server. The node will then retrieve the algorithm, execute it and return the results to the server.

Algorithms are shared using Docker images which are stored in a Docker image registry. The node downloads the algorithms from the Docker registry. In the following sections we explain the fundamentals of algorithm containers.

1. *Algorithm structure*: The typical structure of an algorithm
2. *Input & output*: Interface between the node and algorithm container
3. *Wrappers*: Library to simplify and standardized the node-algorithm input and output
4. *Child containers*: Creating subtasks from an algorithm container
5. *Networking*: Communicate with other algorithm containers and the vantage6-server
6. *Cross language*: Cross language data serialization

4.7.1.1 Algorithm structure

Multi-party analyses commonly have a central part and a remote part. The remote part is responsible for the actual analysis and the central part is often responsible for aggregating the partial results of the remote parts. An alternative to aggregating is orchestration, where the central part does not combine the partial results itself, but instead orchestrates the remote parts in a certain way that also leads to a global result. Of course, the central part may also do both aggregation and orchestration.

In vantage6, we refer to the orchestration part as the **central function** and the federated part as the **partial function**.

A common pattern for a central function would be:

1. Request partial models from all participants
2. Obtain the partial models
3. Combine the partial models to a global model
4. (optional) Repeat step 1-3 until the model converges

In vantage6, it is possible to run only the partial parts of the analysis on the nodes and combine them on your own machine, but it is usually preferable to run the central part within vantage6, because:

- You don't have to keep your machine running during the analysis

- The results are stored on the server, so they may also be accessed by other users

Note: Central functions also run at a node and *not* at the server. For more information, see here.

4.7.1.2 Input & output

The algorithm runs in an isolated environment at the data station. It is important to limit the connectivity and accessability of an algorithm run for security reasons. For instance, by default, algorithms cannot access the internet.

In order for the algorithm to do its work, it needs to be provided with several environment variables and file mounts. The exact environment variables that are available to algorithms are described in the *Environment variables* section. The available file mounts are described below.

Note: This section describes the current process. Keep in mind that this is subjected to be changed. For more information, please see this Github issue

File mounts

The algorithm container has access to several file mounts. These files mounts are provided by the vantage6 infrastructure, so the algorithm developer does not need to provide these files themselves. They can access these files using the environment variables described in the *Environment variables* section.

The available file mounts are:

Input
: The input file contains the user defined input. The user specifies this when a task is created.

Output
: The algorithm writes its output to this file. When the docker container exits, the contents of this file will be send back to the vantage6-server.

Token
: The token file contains a JWT token which can be used by the algorithm to communicate with the central server. The token can only be used to create a new task with the same image, and is only valid while the task has not yet been completed.

Temporary directory
: The temporary directory can be used by an algorithm container to share files with other algorithm containers that:

  - run on the same node
  - have the same `job_id`

  Algorithm containers share a `job_id` as long as they originate from the same user-created task. Child containers (see *Child containers*) therefore have the same `job_id` as their parent container.

The paths to these files and directories are stored in the environment variables, which we will explain now.

4.7.1.3 Wrappers

The vantage6 algorithm wrappers simplifies and standardizes the interaction between algorithm and node. The algorithm wrapper does the following:

- read the data from the database(s) and provide it to the algorithm
- read the environment variables and file mounts and supply these to your algorithm.
- select the appropriate algorithm function to run. In more detail, this means that it provides an entrypoint for the Docker container
- write the output of your algorithm to the output file

Using the wrappers allows algorithm developers to write a single algorithm for multiple types of data sources, because the wrapper is responsible for reading the data from the database(s) and providing it to the algorithm. Note however that algorithms cannot be run using databases that are not supported by the wrapper. The wrapper currently supports the following database types listed *here*.

The wrapper is language specific and currently we support Python and R. Extending this to other languages is usually simple.

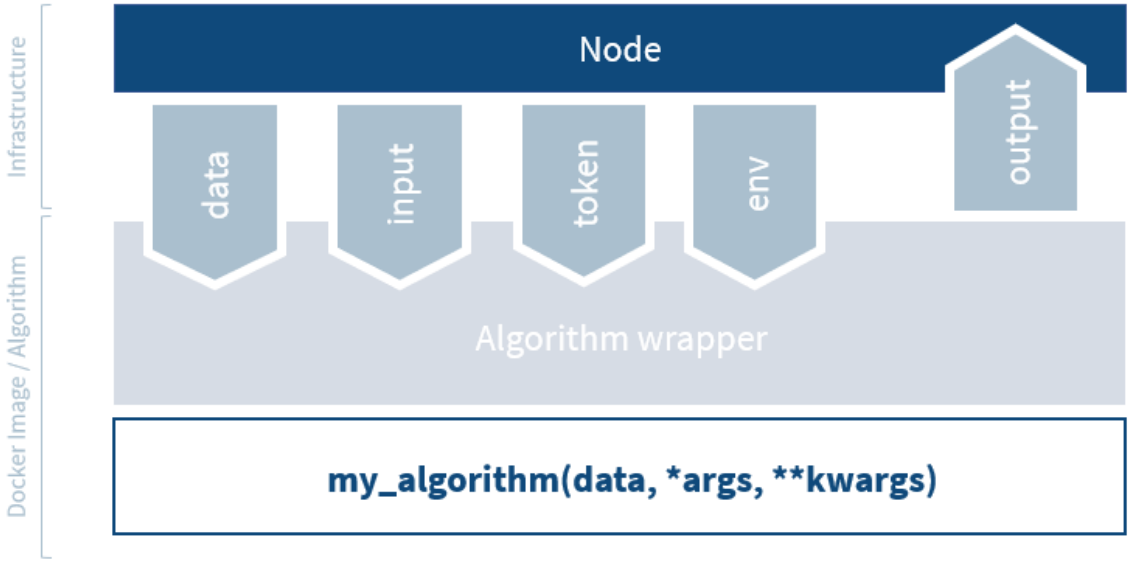


Fig. 4.6: The algorithm wrapper handles algorithm input and output.

4.7.1.4 Child containers

When a user creates a task, one or more nodes spawn an algorithm container. These algorithm containers can create new tasks themselves.

Every algorithm is supplied with a JWT token (see *Input & output*). This token can be used to communicate with the vantage6-server. In case you use an algorithm wrapper, you can supply an `AlgorithmClient` using the *appropriate decorator*.

A child container can be a parent container itself. There is no limit to the amount of task layers that can be created. It is common to have only a single parent container which handles many child containers.

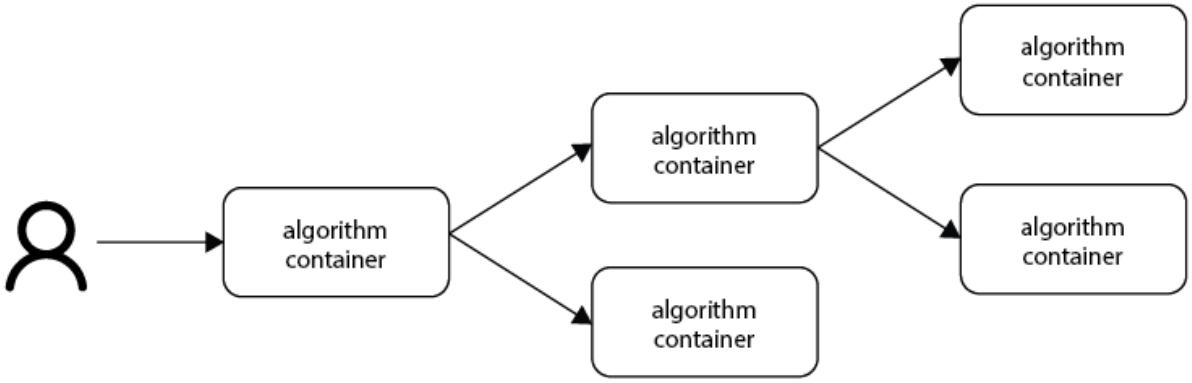


Fig. 4.7: Each container can spawn new containers in the network. Each container is provided with a unique token which they can use to communicate to the vantage6-server.

The token to which the containers have access supplies limited permissions to the container. For example, the token can be used to create additional tasks, but only in the same collaboration, and using the same image.

4.7.1.5 Networking

The algorithm container is deployed in an isolated network to reduce their exposure. Hence, the algorithm it cannot reach the internet. There are two exceptions:

1. When the VPN feature is enabled on the server all algorithm containers are able to reach each other using an `ip` and `port` over VPN.
2. The central server is reachable through a local proxy service. In the algorithm you can use the `HOST`, `POST` and `API_PATH` to find the address of the server.

Note: We are working on a whitelisting feature which allows a node to configure addresses that the algorithm container is able to reach.

VPN connection

Algorithm containers within the same task can communicate directly with each other over a VPN network. More information on that can be found *here* and *this section* describes how to use it in an algorithm.

4.7.1.6 Cross language

Because algorithms are exchanged as Docker images they can be written in any language. This is an advantage as developers can use their preferred language for the problem they need to solve.

Warning: The wrappers are only available for Python and (partially) R, so when you use different language you need to handle the IO yourself. Consult the *Input & Output* section on what the node supplies to your algorithm container.

When data is exchanged between the user and the algorithm they both need to be able to read the data. When the algorithm uses a language specific serialization (e.g. a `pickle` in the case of Python or `RData` in the case of R) the user needs to use the same language to read the results. A better solution would be to use a type of serialization that is not specific to a language. In our wrappers we use JSON for this purpose.

Note: Communication between algorithm containers can use language specific serialization as long as the different parts of the algorithm use the same language.

4.7.2 Algorithm development step-by-step guide

This page offers a step-by-step guide to develop a vantage6 algorithm. We refer to the algorithm concepts section regularly. In that section, we explain the fundamentals of algorithm containers in more detail than in this guide.

Also, note that this guide is mainly aimed at developers who want to develop their algorithm in Python, although we will try to clearly indicate where this differs from algorithms written in other languages.

4.7.2.1 Starting point

When starting to develop a new vantage6 algorithm in Python, the easiest way to start is:

```
v6 algorithm create
```

Running this command will prompt you to answering some questions, which will result in a personalized starting point or 'boilerplate' for your algorithm. After doing so, you will have a new folder with the name of your algorithm, boilerplate code and a checklist in the README.md file that you can follow to complete your algorithm.

Note: There is also a boilerplate for R, but this is not flexible and it is not updated as frequently as the Python boilerplate.

4.7.2.2 Setting up your environment

It is good practice to set up a virtual environment for your algorithm package.

```
# This code is just a suggestion - there are many ways of doing this.

# go to the algorithm directory
cd /path/to/algorithm

# create a Python environment. Be sure to replace <my-algorithm-env> with
# the name of your environment.
conda create -n <my-algorithm-env> python=3.10
conda activate <my-algorithm-env>

# install the algorithm dependencies
pip install -r requirements.txt
```

Also, it is always good to use a version control system such as `git` to keep track of your changes. An initial commit of the boilerplate code could be:

```
cd /path/to/algorithm
git init
git add .
git commit -m "Initial commit"
```

Note that having your code in a git repository is necessary if you want to *update your algorithm*.

4.7.2.3 Implementing your algorithm

Your personalized starting point should make clear to you which functions you need to implement - there are *TODO* comments in the code that indicate where you need to add your own code.

You may wonder why the boilerplate code is structured the way it is. This is explained in the *code structure section*.

4.7.2.4 Environment variables

The algorithms have access to several environment variables. You can also specify additional environment variables via the `algorithm_env` option in the node configuration files (see the *example node configuration file*).

Environment variables provided by the vantage6 infrastructure are used to locate certain files or to add local configuration settings into the container. These are usually used in the Python wrapper and you don't normally need them in your functions. However, you can access them in your functions as follows:

```
def my_function():
    input_file = os.environ["INPUT_FILE"]
    token_file = os.environ["DEFAULT_DATABASE_URI"]

    # do something with the input file and database URI
    pass
```

The environment variables that you specify in the node configuration file can be used in the exact same manner. You can view all environment variables that are available to your algorithm by `print(os.environ)`.

4.7.2.5 Returning results

Returning the results of you algorithm is rather straightforward. At the end of your algorithm function, you can simply return the results as a dictionary:

```
def my_function(column_name: str):
    return {
        "result": 42
    }
```

These results will be returned to the user after the algorithm has finished.

Warning: The results that you return should be JSON serializable. This means that you cannot, for example, return a `pandas.DataFrame` or a `numpy.ndarray` (such objects may not be readable to a non-Python using recipient or may even be insecure to send over the internet). They should be converted to a JSON-serializable format first.

4.7.2.6 Example functions

Just an example of how you can implement your algorithm:

Central function

```
from vantage6.algorithm.tools.decorators import algorithm_client
from vantage6.client.algorithm_client import AlgorithmClient

 @algorithm_client
 def main(client: AlgorithmClient, *args, **kwargs):
    # Run partial function.
    task = client.task.create(
       {
          "method": "my_algorithm",
          "args": args,
          "kwargs": kwargs
       },
       organization_ids=[1, 2]
    )

     # wait for the federated part to complete
     # and return
     results = wait_and_collect(task)

     return results
```

Partial function

```
import pandas as pd
from vantage6.algorithm.tools.decorators import data

@data(1)
def my_partial_function(data: pd.DataFrame, column_name: str):
    # do something with the data
    data[column_name] = data[column_name] + 1

    # return the results
    return {
        "result": sum(data[colum_name].to_list())
    }
```

4.7.2.7 Testing your algorithm

It can be helpful to test your algorithm outside of Docker using the `MockAlgorithmClient`. This may save time as it does not require you to set up a test infrastructure with a vantage6 server and nodes, and allows you to test your algorithm without building a Docker image every time. The algorithm boilerplate code comes with a test file that you can use to test your algorithm using the `MockAlgorithmClient` - you can of course extend that to add more or different tests.

The *MockAlgorithmClient* has the same interface as the `AlgorithmClient`, so it should be easy to switch between the two. An example of how you can use the `MockAlgorithmClient` to test your algorithm is included in the boilerplate code.

4.7.2.8 Writing documentation

It is important that you add documentation of your algorithm so that users know how to use it. In principle, you may choose any format of documentation, and you may choose to host it anywhere you like. However, in our experience it works well to keep your documentation close to your code. We recommend using the `readthedocs` platform to host your documentation. Alternatively, you could use a `README` file in the root of your algorithm directory - if the documentation is not too extensive, this may be sufficient.

Note: We intend to provide a template for the documentation of algorithms in the future. This template will be based on the `readthedocs` platform.

4.7.2.9 Package & distribute

The algorithm boilerplate comes with a `Dockerfile` that is a blueprint for creating a Docker image of your algorithm. This Docker image is the package that you will distribute to the nodes.

If you go to the folder containing your algorithm, you will also find the Dockerfile there, immediately at the top directory. You can then build the project as follows:

```
docker build -t repo/image:tag .
```

The `-t` indicated the name of your image. This name is also used as reference where the image is located on the internet. Once the Docker image is created it needs to be uploaded to a registry so that nodes can retrieve it, which you can do by pushing the image:

```
docker push repo/image:tag
```

Here are a few examples of how to build and upload your image:

```
# Build and upload to Docker Hub. Replace <my-user-name> with your Docker
# Hub username and make sure you are logged in with ``docker login``.
docker build -t my-user-name/algorithm-example:latest .
docker push my-user-name/algorithm-example:latest

# Build and upload to private registry. Here you don't need to provide
# a username but you should write out the full image URL. Also, again you
# need to be logged in with ``docker login``.
docker build -t harbor2.vantage6.ai/PROJECT/algorithm-example:latest .
docker push harbor2.vantage6.ai/PROJECT/algorithm-example:latest
```

Now that your algorithm has been uploaded it is available for nodes to retrieve when they need it.

4.7.2.10 Calling your algorithm from vantage6

If you want to test your algorithm in the context of vantage6, you should set up a vantage6 infrastructure. You should create a server and at least one node (depending on your algorithm you may need more). Follow the instructions in the *Server admin guide* and *Node admin guide* to set up your infrastructure. If you are running them on the same machine, take care to provide the node with the proper address of the server as detailed *here*.

Once your infrastructure is set up, you can create a task for your algorithm. You can do this either via the *UI* or via the *Python client*.

4.7.2.11 Updating your algorithm

At some point, there may be changes in the vantage6 infrastructure that require you to update your algorithm. Such changes are made available via the `v6 algorithm update` command. This command will update your algorithm to the latest version of the vantage6 infrastructure.

You can also use the `v6 algorithm update` command to update your algorithm if you want to modify your answers to the questionnaire. In that case, you should be sure to commit the changes in `git` before running the command.

4.7.3 Algorithm code structure

Note: These guidelines are Python specific.

Here we provide some more information on algorithm code is organized. Most of these structures are generated automatically when you create a *personalized algorithm starting point*. We detail them here so that you understand why the algorithm code is structured as it is, and so that you know how to modify it if necessary.

4.7.3.1 Defining functions

The functions that will be available to the user have to be defined in the `__init__.py` file at the base of your algorithm module. Other than that, you have complete freedom in which functions you implement.

Vantage6 algorithms commonly have an orchestator or aggregator part and a remote part. The orchestrator part is responsible for combining the partial results of the remote parts. The remote part is usually executed at each of the nodes included in the analysis. While this structure is common for vantage6 algorithms, it is not required.

If you do follow this structure however, we recommend the following file structure:

```
my_algorithm/
├── __init__.py
├── central.py
└── partial.py
```

where `__init__.py` contains the following:

```
from .central import my_central_function
from .partial import my_partial_function
```

and where `central.py` and `partial.py` obviously contain the implementation of those functions.

4.7.3.2 Implementing the algorithm functions

Let's say you are implementing a function called `my_function`:

```
def my_function(column_name: str):
    pass
```

You have complete freedom as to what arguments you define in your function; `column_name` is just an example. Note that these arguments have to be provided by the user when the algorithm is called. This is explained *here* for the Python client.

Often, you will want to use the data that is available at the node. This data can be provided to your algorithm function in the following way:

```
import pandas as pd
from vantage6.algorithm.tools.decorators import data

@data(2)
def my_function(df1: pd.DataFrame, df2: pd.DataFrame, column_name: str):
    pass
```

The `@data(2)` decorator indicates that the first two arguments of the function are dataframes that should be provided by the vantage6 infrastructure. In this case, the user would have to specify two databases when calling the algorithm. Note that depending on the type of the database used, the user may also have to specify additional parameters such as a SQL query or the name of a worksheet in an Excel file.

Note that it is also possible to just specify `@data()` without an argument - in that case, a single dataframe is added to the arguments.

For some data sources it's not trivial to construct a dataframe from the data. One of these data sources is the OHDSI OMOP CDM database. For this data source, the `@database_connection` is available:

```
from rpy2.robjects import RS4
from vantage6.algorithm.tools.decorators import (
    database_connection, OHDSIMetaData
)

@database_connection(types=["OMOP"], include_metadata=True)
def my_function(connection: RS4, metadata: OHDSIMetaData,
                <other_arguments>):
    pass
```

This decorator provides the algorithm with a database connection that can be used to interact with the database. For instance, you can use this connection to execute functions from python-ohdsi package. The `include_metadata` argument indicates whether the metadata of the database should also be provided. It is possible to connect to multiple databases at once, but you can also specify a single database by using the `types` argument.

```
from rpy2.robjects import RS4
from vantage6.algorithm.tools.decorators import database_connection

@database_connection(types=["OMOP", "OMOP"], include_metadata=False)
def my_function(connection1: RS4, connection2: Connection,
                <other_arguments>):
    pass
```

Note: The `@database_connection` decorator is current only available for OMOP CDM databases. The connection object `RS4` is an R object, mapped to Python using the rpy2, package. This object can be passed directly on to the functions from *python-ohdsi <https://python-ohdsi.readthedocs.io/>*.

Another useful decorator is the `@algorithm_client` decorator:

```
import pandas as pd
from vantage6.client.algorithm_client import AlgorithmClient
from vantage6.algorithm.tools.decorators import algorithm_client, data

@data()
@algorithm_client
def my_function(client: AlgorithmClient, df1: pd.DataFrame, column_name: str):
    pass
```

This decorator provides the algorithm with a client that can be used to interact with the vantage6 central server. For instance, you can use this client in the central part of an algorithm to create a subtasks for each node with `client.task.create()`. A full list of all commands that are available can be found in the *algorithm client documentation*.

Warning: The decorators `@data` and `@algorithm_client` each have one reserved keyword: `mock_data` for the `@data` decorator and `mock_client` for the `@algorithm_client` decorator. These keywords should not be used as argument names in your algorithm functions.

The reserved keywords are used by the *MockAlgorithmClient* to mock the data and the algorithm client. This is useful for testing your algorithm locally.

4.7.3.3 Algorithm wrappers

The vantage6 *wrappers* are used to simplify the interaction between the algorithm and the node. The wrappers are responsible for reading the input data from the data source and supplying it to the algorithm. They also take care of writing the results back to the data source.

As algorithm developer, you do not have to worry about the wrappers. The main point you have to make sure is that the following line is present at the end of your `Dockerfile`:

```
CMD python -c "from vantage6.algorithm.tools.wrap import wrap_algorithm; wrap_algorithm()
↪"
```

The `wrap_algorithm` function will wrap your algorithm to ensure that the vantage6 algorithm tools are available to it. Note that the `wrap_algorithm` function will also read the `PKG_NAME` environment variable from the `Dockerfile` so make sure that this variable is set correctly.

For R, the command is slightly different:

```
CMD Rscript -e "vtg::docker.wrapper('$PKG_NAME')"
```

Also, note that when using R, this only works for CSV files.

4.7.3.4 VPN

Within vantage6, it is possible to communicate with algorithm instances running on different nodes via the *VPN network feature*. Each of the algorithm instances has their own IP address and port within the VPN network. In your algorithm code, you can use the `AlgorithmClient` to obtain the IP address and port of other algorithm instances. For example:

```
from vantage6.client import AlgorithmClient

def my_function(client: AlgorithmClient, ...):
    # Get the IP address and port of the algorithm instance with id 1
    child_addresses = client.get_child_addresses()
    # returns something like:
    # [
    #     {
    #       'port': 1234,
    #       'ip': 11.22.33.44,
    #       'label': 'some_label',
    #       'organization_id': 22,
    #       'task_id': 333,
    #       'parent_id': 332,
    #     }, ...
    # ]

    # Do something with the IP address and port
```

The function `get_child_addresses()` gets the VPN addresses of all child tasks of the current task. Similarly, the function `get_parent_address()` is available to get the VPN address of the parent task. Finally, there is a client function `get_addresses()` that returns the VPN addresses of all algorithm instances that are part of the same task.

VPN communication is only possible if the docker container exposes ports to the VPN network. In the algorithm boilerplate, one port is exposed by default. If you need to expose more ports (e.g. for sending different information to different parts of your algorithm), you can do so by adding lines to the Dockerfile:

```
# port 8888 is used by the algorithm for communication purposes
EXPOSE 8888
LABEL p8888 = "some-label"

# port 8889 is used by the algorithm for data-exchange
EXPOSE 8889
LABEL p8889 = "some-other-label"
```

The `EXPOSE` command exposes the port to the VPN network. The `LABEL` command adds a label to the port. This label returned with the clients' `get_addresses()` function suite. You may specify as many ports as you need. Note that you *must* specify the label with `p` as prefix followed by the port number. The vantage6 infrastructure relies on this naming convention.

4.7.3.5 Dockerfile structure

Once the algorithm code is written, the algorithm needs to be packaged and made available for retrieval by the nodes. The algorithm is packaged in a Docker image. A Docker image is created from a Dockerfile, which acts as a blue-print.

The Dockerfile is already present in the boilerplate code. Usually, the only line that you need to update is the `PKG_NAME` variable to the name of your algorithm package.

Warning: This classic tutorial was written for vantage6 version 2.x. The commands below have not been updated and therefore might not work anymore. We are leaving this here for reference, as it includes some useful information about concepts that may not be included elsewhere in this documentation.

4.7.4 Classic Tutorial

In this section the basic steps for creating an algorithm for horizontally partitioned data are explained.

Note: The final code of this tutorial is published on Github. The algorithm is also published in our Docker registry: *harbor2.vantage6.ai/demo/average*

It is assumed that it is mathematically possible to create a federated version of the algorithm you want to use. In the following sections we create a federated algorithm to compute the average of a distributed dataset. An overview of the steps that we are going through:

1. Mathematically decompose the model
2. Federated implementation and local testing
3. Vantage6 algorithm wrapper
4. Dockerize and push to a registry

This tutorial shows you how to create a **federated mean** algorithm.

4.7.4.1 Mathematical decomposition

The mean of $Q = [q_1, q_2 ... q_n]$ is computed as:

$$Q_{mean} = \frac{1}{n} \sum_{i=1}^{n} q_i = \frac{q_1 + q_2 + ... + q_n}{n}$$

When dataset Q is **horizontally partitioned** in dataset A and B:

$$A = [a_1, a_2 ... a_j] = [q_1, q_2 ... q_j]$$
$$B = [b_1, b_2 ... b_k] = [q_{j+1}, q_{j+2} ... q_n]$$

We would like to compute Q_{mean} from dataset A and B. This could be computed as:

$$Q_{mean} = \frac{(a_1 + a_2 + ... + a_j) + (b_1 + b_2 + ... + b_k)}{j + k} = \frac{\sum A + \sum B}{j + k}$$

Both the number of samples in each dataset and the total sum of each dataset is needed. Then we can compute the global average of dataset A and B.

Note: We cannot simply compute the average on each node and combine them, as this would be mathematically incorrect. This would only work if dataset **A** and **B** contain the exact same number of samples.

4.7.4.2 Federated implementation

> **Warning:** In this example we use python, however you are free to use any language. The only requirements are: 1) It has to be able to create HTTP-requests, and 2) has to be able to read and write to files.
>
> However, if you use a different language you are not able to use our wrapper. Reach out to us on Discord to discuss how this works.

A federated algorithm consist of two parts:

1. A federated part of the algorithm which is responsible for creating the partial results. In our case this would be computing (1) the sum of the observations, and (2) the number of observations.
2. A central part of the algorithm which is responsible for combining the partial results from the nodes. In the case of the federated mean that would be dividing the total sum of the observations by the total number of observations.

Note: The central part of the algorithm can either be run on the machine of the researcher himself or in a master container which runs on a node. The latter is the preferred method.

In case the researcher runs this part, he/she needs to have a proper setup to do so (i.e. a Python environment with the necessary dependencies). This can be useful when developing new algorithms.

Federated part

The node that runs this part contains a CSV-file with one column (specified by the argument *column_name*) which we want to use to compute the global mean. We assume that this column has no *NaN* values.

```python
import pandas

def federated_part(path, column_name="numbers"):
    """Compute the sum and number of observations of a column"""

    # extract the column numbers from the CSV
    numbers = pandas.read_csv(path)[column_name]

    # compute the sum, and count number of rows
    local_sum = numbers.sum()
    local_count = len(numbers)

    # return the values as a dict
    return {
        "sum": local_sum,
        "count": local_count
    }
```

Central part

The central algorithm receives the sums and counts from all sites and combines these to a global mean. This could be from one or more sites.

```
def central_part(node_outputs):
    """Combine the partial results to a global average"""
    global_sum = 0
    global_count = 0
    for output in node_outputs:
        global_sum += output["sum"]
        global_count += output["count"]

    return {"average": global_sum / global_count}
```

Local testing

To test, simply create two datasets **A** and **B**, both having a numerical column **numbers**. Then run the following:

```
outputs = [
    federated_part("path/to/dataset/A"),
    federated_part("path/to/dataset/B")
]
Q_average = central_part(outputs)["average"]
print(f"global average = {Q_average}.")
```

4.7.4.3 Vantage6 integration

Note: A good starting point would be to use the boilerplate code from our Github. This section outlines the steps needed to get to this boilerplate but also provides some background information.

Note: In this example we use a **csv**-file. It is also possible to use other types of data sources. This tutorial makes use of our algorithm wrapper which is currently only available for **csv**, **SPARQL** and **Parquet** files.

Other wrappers like **SQL**, **OMOP**, etc. are under consideration. Let us now if you want to use one of these or other datasources.

Now that we have a federated implementation of our algorithm we need to make it compatible with the vantage6 infrastructure. The infrastructure handles the communication with the server and provides data access to the algorithm.

The algorithm consumes a file containing the input. This contains both the method name to be triggered as well as the arguments provided to the method. The algorithm also has access to a CSV file (in the future this could also be a database) on which the algorithm can run. When the algorithm is finished, it writes back the output to a different file.

The central part of the algorithm has to be able to create (sub)tasks. These subtasks are responsible for executing the federated part of the algorithm. The central part of the algorithm can either be executed on one of the nodes in the vantage6 network or on the machine of a researcher. In this example we only show the case in which one of the nodes executes the central part of the algorithm. The node provides the algorithm with a JWT token so that the central part of the algorithm has access to the server to post these subtasks.

Algorithm Structure

The algorithm needs to be structured as a Python package. This way the algorithm can be installed within the Docker image. The minimal file-structure would be:

```
project_folder
├── Dockerfile
├── setup.py
└── algorithm_pkg
    └── __init__.py
```

We also recommend adding a `README.md`, `LICENSE` and `requirements.txt` to the *project_folder*.

setup.py

Contains the setup method to create a package from your algorithm code. Here you specify some details about your package and the dependencies it requires.

```
from os import path
from codecs import open
from setuptools import setup, find_packages

# we're using a README.md, if you do not have this in your folder, simply
# replace this with a string.
here = path.abspath(path.dirname(__file__))
with open(path.join(here, 'README.md'), encoding='utf-8') as f:
    long_description = f.read()

# Here you specify the meta-data of your package. The `name` argument is
# needed in some other steps.
setup(
    name='v6-average-py',
    version="1.0.0",
    description='vantage6 average',
    long_description=long_description,
    long_description_content_type='text/markdown',
    url='https://github.com/IKNL/v6-average-py',
    packages=find_packages(),
    python_requires='>=3.10',
    install_requires=[
        'vantage6-client',
        # list your dependencies here:
        # pandas, ...
    ]
)
```

Note: The `setup.py` above is sufficient in most cases. However if you want to do more advanced stuff (like adding static data, or a CLI) you can use the extra arguments from `setup`.

Dockerfile

The Dockerfile contains the recipe for building the Docker image. Typically you only have to change the argument `PKG_NAME` to the name of you package. This name should be the same as as the name you specified in the `setup.py`. In our case that would be `v6-average-py`.

```
# This specifies our base image. This base image contains some commonly used
# dependancies and an install from all vantage6 packages. You can specify a
# different image here (e.g. python:3). In that case it is important that
# `vantage6-client` is a dependancy of you project as this contains the wrapper
# we are using in this example.
FROM harbor2.vantage6.ai/algorithms/algorithm-base

# Change this to the package name of your project. This needs to be the same
# as what you specified for the name in the `setup.py`.
ARG PKG_NAME="v6-average-py"

# This will install your algorithm into this image.
COPY . /app
RUN pip install /app

# This will run your algorithm when the Docker container is started. The
# wrapper takes care of the IO handling (communication between node and
# algorithm). You dont need to change anything here.
ENV PKG_NAME=${PKG_NAME}
CMD python -c "from vantage6.tools.docker_wrapper import docker_wrapper; docker_wrapper('
↪${PKG_NAME}')"
```

`__init__.py`

This contains the code for your algorithm. It is possible to split this into multiple files, however the methods that should be available to the researcher should be in this file. You can do that by simply importing them into this file (e.g. `from .average import my_nested_method`)

We can distinguish two types of methods that a user can trigger:

name	description	prefix	arguments
master	Central part of the algorithm. Receives a `client` as argument which provides an interface to the central server. This way the master can create tasks and collect their results.		`(client, data, *args, **kwargs)`
Remote procedure call	Consumes the data at the node to compute the partial.	*RPC_*	`(data, *args, **kwargs)`

Warning: Everything that is returned by the`return` statement is sent back to the central vantage6-server. This should never contain any privacy-sensitive information.

Warning: The `client` the master method receives is an `AlgorithmClient` (or a `ContainerClient` if you are using an older version), which is different than the client you use as a user.

For our average algorithm the implementation will look as follows:

```
import time

from vantage6.tools.util import info

def master(client, data, column_name):
    """Combine partials to global model

    First we collect the parties that participate in the collaboration.
    Then we send a task to all the parties to compute their partial (the
    row count and the column sum). Then we wait for the results to be
    ready. Finally when the results are ready, we combine them to a
    global average.

    Note that the master method also receives the (local) data of the
    node. In most usecases this data argument is not used.

    The client, provided in the first argument, gives an interface to
    the central server. This is needed to create tasks (for the partial
    results) and collect their results later on. Note that this client
    is a different client than the client you use as a user.
    """

    # Info messages can help you when an algorithm crashes. These info
    # messages are stored in a log file which is send to the server when
    # either a task finished or crashes.
    info('Collecting participating organizations')

    # Collect all organization that participate in this collaboration.
    # These organizations will receive the task to compute the partial.
    organizations = client.get_organizations_in_my_collaboration()
    ids = [organization.get("id") for organization in organizations]

    # Request all participating parties to compute their partial. This
    # will create a new task at the central server for them to pick up.
    # We've used a kwarg but is is also possible to use `args`. Although
    # we prefer kwargs as it is clearer.
    info('Requesting partial computation')
    task = client.create_new_task(
        input_={
            'method': 'average_partial',
            'kwargs': {
                'column_name': column_name
            }
        },
        organization_ids=ids
    )
```

(continues on next page)

(continued from previous page)

```
    # Now we need to wait untill all organizations(/nodes) finished
    # their partial. We do this by polling the server for results. It is
    # also possible to subscribe to a websocket channel to get status
    # updates.
    info("Waiting for results")
    results = client.wait_for_results(task_id=task.get("id"))

    # Now we can combine the partials to a global average.
    global_sum = 0
    global_count = 0
    for result in results:
        global_sum += result["sum"]
        global_count += result["count"]

    return {"average": global_sum / global_count}

def RPC_average_partial(data, column_name):
    """Compute the average partial

    The data argument contains a pandas-dataframe containing the local
    data from the node.
    """

    # extract the column_name from the dataframe.
    info(f'Extracting column {column_name}')
    numbers = data[column_name]

    # compute the sum, and count number of rows
    info('Computing partials')
    local_sum = numbers.sum()
    local_count = len(numbers)

    # return the values as a dict
    return {
        "sum": local_sum,
        "count": local_count
    }
```

Local testing

Now that we have a vantage6 implementation of the algorithm it is time to test it. Before we run it in a vantage6 setup we can test it locally by using the `ClientMockProtocol` which simulates the communication with the central server.

Before we can locally test it we need to (editable) install the algorithm package so that the Mock client can use it. Simply go to the root directory of your algorithm package (with the `setup.py` file) and run the following:

```
pip install -e .
```

Then create a script to test the algorithm:

```python
from vantage6.tools.mock_client import ClientMockProtocol

# Initialize the mock server. The datasets simulate the local datasets from
# the node. In this case we have two parties having two different datasets:
# a.csv and b.csv. The module name needs to be the name of your algorithm
# package. This is the name you specified in `setup.py`, in our case that
# would be v6-average-py.
client = ClientMockProtocol(
    datasets=["local/a.csv", "local/b.csv"],
    module="v6-average-py"
)

# to inspect which organization are in your mock client, you can run the
# following
organizations = client.get_organizations_in_my_collaboration()
org_ids = ids = [organization["id"] for organization in organizations]

# we can either test a RPC method or the master method (which will trigger the
# RPC methods also). Lets start by triggering an RPC method and see if that
# works. Note that we do *not* specify the RPC_ prefix for the method! In this
# example we assume that both a.csv and b.csv contain a numerical column `age`.
average_partial_task = client.create_new_task(
    input_={
        'method':'average_partial',
        'kwargs': {
            'column_name': 'age'
        }
    },
    organization_ids=org_ids
)

# You can directly obtain the result (we dont have to wait for nodes to
# complete the tasks)
results = client.result.from_task(average_partial_task.get("id"))
print(results)

# To trigger the master method you also need to supply the `master`-flag
# to the input. Also note that we only supply the task to a single organization
# as we only want to execute the central part of the algorithm once. The master
# task takes care of the distribution to the other parties.
average_task = client.create_new_task(
    input_={
        'master': 1,
        'method':'master',
        'kwargs': {
            'column_name': 'age'
        }
    },
    organization_ids=[org_ids[0]]
)
results = client.result.from_task(average_task.get("id"))
print(results)
```

Building and Distributing

Now that we have a fully tested algorithm for the vantage6 infrastructure. We need to package it so that it can be distributed to the data-stations/nodes. Algorithms are delivered in Docker images. So that's where we need the `Dockerfile` for. To build an image from our algorithm (make sure you have docker installed and it's running) you can run the following command from the root directory of your algorithm project.

```
docker build -t harbor2.vantage6.ai/demo/average .
```

The option `-t` specifies the (unique) identifier used by the researcher to use this algorithm. Usually this includes the registry address (harbor2.vantage6.ai) and the project name (demo).

Note: In case you are using docker hub as registry, you do not have to specify the registry or project as these are set by default to the Docker hub and your docker hub username.

```
docker push harbor2.vantage6.ai/demo/average
```

Note: Reach out to us on Discord if you want to use our registries (harbor2.vantage6.ai and harbor2.vantage6.ai).

4.8 Feature descriptions

Under construction

The vantage6 platform contains many features - some of which are optional, some which are always active. This section aims to give an overview of the features and how they may be used.

Each component has its own set of features. The features are described in the following sections, as well as a section on inter-component features.

4.8.1 Server features

The following pages each describe one feature of the vantage6 server.

4.8.1.1 Two-factor authentication

Available since version 3.5.0

The vantage6 infrastructure includes the option to use two-factor authentication (2FA). This option is set at the server level: the server administrator decides if it is either enabled or disabled for everyone. Users cannot set this themselves. Server administrators can choose to require 2FA when prompted in `v6 server new`, or by adding the option `two_factor_auth: true` to the configuration file (see *Configure*).

Currently, the only 2FA option is to use Time-based one-time passwords (TOTP) With this form of 2FA, you use your phone to scan a QR code using an authenticator app like LastPass authenticator or Google authenticator. When you scan the QR code, your vantage6 account is added to the authenticator app and will show you a 6-digit code that changes every 30 seconds.

Setting up 2FA for a user

If a new user logs in, or if a user logs in for the first time after a server administrator has enabled 2FA, they will be required to set it up. The endpoint `/token/user` will first verify that their password is correct, and then set up 2FA. It does so by generating a random TOTP secret for the user, which is stored in the database. From this secret, a URI is generated that can be used to visualize the QR code.

If the user is logging in via the vantage6 user interface, this QR code will be visualized to allow the user to scan it. Also, users that login via the Python client will be shown a QR code. In both cases, they also have the option to manually enter the TOTP secret into their authenticator app, in case scanning the QR code is not possible.

Users that log in via the R client or directly via the API will have to visualize the QR code themselves, or manually enter the TOTP secret into their authenticator app.

Using 2FA

If a user has already setup 2FA tries to login, the endpoint `/token/user` will require that they provide their 6-digit TOTP code via the `mfa_code` argument. This code will be checked using the TOTP secret stored in the database, and if it is valid, the user will be logged in.

To prevent users with a slow connection from having difficulty logging in, valid codes from the 30s period directly prior to the current period will also be logged in.

Resetting 2FA

When a user loses access to their 2FA, they may reset it via their email. They should use the endpoint `/recover/2fa/lost` to get an email with a reset token and then use the reset token in `/recover/2fa/reset` to reset 2FA. This endpoint will give them a new QR code that they can visualize just like the initial QR code.

4.8.1.2 Horizontal scaling

By horizontal scaling, we mean that you can run multiple instances of the vantage6 server simultaneously to handle a high workload. This is useful when a single machine running the server is no longer sufficient to handle all requests.

How it works

Horizontal scaling with vantage6 can be done using a RabbitMQ server. RabbitMQ is a widely used message broker. Below, we will first explain how we use RabbitMQ, and then discuss the implementation.

The websocket connection between server and nodes is used to process various changes in the network's state. For example, a node can create a new (sub)task for the other nodes in the collaboration. The server then communicates these tasks via the socket connection. Now, if we use multiple instances of the central server, different nodes in the same collaboration may connect to different instances, and then, the server would not be able to deliver the new task properly. This is where RabbitMQ comes in.

When RabbitMQ is enabled, the websocket messages are directed over the RabbitMQ message queue, and delivered to the nodes regardless of which server instance they are connected to. The RabbitMQ service thus helps to ensure that all websocket events are still communicated properly to all involved parties.

How to use

If you use multiple server instances, you should always connect them to the same RabbitMQ instance. You can achieve this by adding your RabbitMQ server when you create a new server with `v6 server new`, or you can add it later to your server configuration file as follows:

```
rabbitmq_uri: amqp://$user:$password@$host:$port/$vhost
```

Where `$user` is the username, `$password` is the password, `$host` is the URL where your RabbitMQ service is running, `$port` is the queue's port (which is 5672 if you are using the RabbitMQ Docker image), and `$vhost` is the name of your virtual host (you could e.g. run one instance group per vhost).

Deploy

If you are running a test server with `v6 server start`, a RabbitMQ docker container will be started automatically for you. This docker container contains a management interface which will be available on port 15672.

For deploying a production server, there are several options to run RabbitMQ. For instance, you can install RabbitMQ on Azure.

4.8.1.3 API response structure

Each API endpoint returns a JSON response. All responses are structured in the same way, loosely following HATEOAS rules. An example is detailed below:

```
>>> client.task.get(task_id)
{
    "id": 1,
    "name": "test",
    "results": "/api/result?task_id=1",
    "image": "harbor2.vantage6.ai/testing/v6-test-py",
    ...
}
```

The response for this task includes a link to the results that are attached to this task. More detail on the results are provided when collecting the response for that link.

4.8.2 Node features

The following pages each describe one feature of the vantage6 node.

** Under construction **

4.8.2.1 Whitelisting

Available since version 3.9.0

Vantage6 algorithms are normally disconnected from the internet, and are therefore unable to connect to access data that is not connected to the node on node startup. Via this feature it is possible to whitelist certain domains, ips and ports to allow the algorithm to connect to these resources. It is important to note that only the http protocol is supported. If you require a different protocol, please look at *SSH Tunnel*.

> **Warning:** As a node owner you are responsible for the security of your node. Make sure you understand the implications of whitelisting before enabling this feature.
>
> Be aware that when a port is whitelisted it is whitelisted for all domains and ips.

Setting up whitelisting

Add block `whitelist` to the node configuration file:

```yaml
whitelist:
    domains:
        - .google.com
        - github.com
        - host.docker.internal # docker host ip (windows/mac)
    ips:
        - 172.17.0.1 # docker bridge ip (linux)
        - 8.8.8.8
    ports:
        - 443
```

> **Note:** This feature makes use of Squid, which is a proxy server. For every domain, ip and port a *acl* directive is created. See their documentation for more details on what valid values are.

Implementation details / Notes

The algorithm container is provided with the environment variables `http_proxy`, `HTTP_PROXY`, `https_proxy`, `HTTPS_PROXY`, `no_proxy` and `NO_PROXY`. Unfortunately, there is no standard for handling these variables. Therefore, whether this works will depend on the application you are using. See this post for more details.

In case the algorithm tries to connect to a domain that is not whitelisted, a http 403 error will be returned by the squid instance.

> **Warning:** Make sure the requests from the algorithm are using the environment variables. Some libraries will ignore these variables and use their own configuration.
>
> - The `requests` library will work for all cases.
> - The `curl` command will not work for vantage6 VPN addresses as the format of `no_proxy` variable is not supported. You can fix this by using the `--noproxy` option when requesting a VPN address.

Note: VPN addresses in `no_proxy` have the same format as in the node configuration file, by default `10.76.0.0/16`. Make sure the request library understands this format when connecting to a VPN address.

4.8.2.2 SSH Tunnel

Available since version 3.7.0

Vantage6 algorithms are normally disconnected from the internet, and are therefore unable to connect to access data that is not connected to the node on node startup. Via this feature, however, it is possible to connect to a remote server through a secure SSH connection. This allows you to connect to a dataset that is hosted on another machine than your node, as long as you have SSH access to that machine.

An alternative solution would be to create a whitelist of domains, ports and IP addresses that are allowed to be accessed by the algorithm.

Setting up SSH tunneling

1. Create a new SSH key pair

Create a new key pair *without* a password on your node machine. To do this, enter the command below in your terminal, and leave the password empty when prompted.

```
ssh-keygen -t rsa
```

You are required not to use a password for the private key, as vantage6 will set up the SSH tunnel without user intervention and you will therefore not be able to enter the password in that process.

2. Add the public key to the remote server

Copy the contents of the public key file (`your_key.pub`) to the remote server, so that your node will be allowed to connect to it. In the most common case, this means adding your public key to the `~/.ssh/authorized_keys` file on the remote server.

3. Add the SSH tunnel to your node configuration

An example of the SSH tunnel configuration can be found below. See *here* for a full example of a node configuration file.

```
databases:
  httpserver: http://my_http:8888
ssh-tunnels:
  - hostname: my_http
    ssh:
      host: my-remote-machine.net
      port: 22
      fingerprint: "ssh-rsa AAAAE2V....wwef987vD0="
      identity:
        username: bob
```

(continues on next page)

(continued from previous page)

```
      key: /path/to/your/private/key
  tunnel:
    bind:
      ip: 0.0.0.0
      port: 8888
    dest:
      ip: 127.0.0.1
      port: 9999
```

There are a few things to note about the SSH tunnel configuration:

1. You can provide multiple SSH tunnels in the `ssh-tunnels` list, by simply extending the list.
2. The hostname of each tunnel should come back in one of the databases, so that they may be accessible to the algorithms.
3. The `host` is the address at which the remote server can be reached. This is usually an IP address or a domain name. Note that you are able to specify IP addresses in the local network. Specifying non-local IP addresses is not recommended, as you might be exposing your node if the IP address is spoofed.
4. The `fingerprint` is the fingerprint of the remote server. You can usually find it in */etc/ssh/ssh_host_rsa_key.pub* on the remote server.
5. The `identity` section contains the username and path to the private key your node is using. The username is the username you use to log in to the remote server, in the case above it would be `ssh bob@my-remote-machine.net`.
6. The `tunnel` section specifies the port on which the SSH tunnel will be listening, and the port on which the remote server is listening. In the example above, on the remote machine, there would be a service listening on port 9999 on the machine itself (which is why the IP is 127.0.0.1 a.k.a. localhost). The tunnel will be bound to port 8888 on the node machine, and you should therefore take care to include the correct port in your database path.

Using the SSH tunnel

How you should use the SSH tunnel depends on the service that you are running on the other side. In the example above, we are running a HTTP server and therefore we should obtain data via HTTP requests. In the case of a SQL service, one would need to send SQL queries to the remote server instead.

Note: We aim to extend this section later with an example of an algorithm that is using this feature.

4.8.2.3 Linked docker containers

Available since version 3.2.0

You may have a service running in a Docker container that you would like to make available to your algorithm. This may be useful, for example, if you are running a (test) SQL database in a container and want to make it available to your algorithm without having to set up whitelisting or SSH tunnels.

You can define the container that you want to make available to the algorithm in the *docker_services* section of your node configuration file:

where *container_name* is the name of your Docker container. This container will be made available in the Docker network where the algorithm containers are running, so your algorithm will be able to access it via *http://localhost*. The *container_label* will be used as alias for the container in the isolated Docker network.

Note that this option only works if your container with *container_name* is already running when you start the node. If it is not, the node will not be able to link the container to the isolated docker network and will print a warning.

4.8.3 Algorithm features

The following pages each describe one feature of vantage6 algorithms.

4.8.3.1 Algorithm wrappers

Algorithm wrappers are used in algorithms to make it easier for algorithms to handle input and output.

- list the available wrappers
- links to their docstrings

4.8.3.2 Algorithm container isolation

The algorithms run in vantage6 have access to the sensitive data that we want to protect. Also, the algorithms may be built improperly, or may be outdated, which might make it vulnerable to attacks. Therefore, one of the important security measures that vantage6 implements is that all algorithms run in a container that is not connected to the internet. The isolation from the internet is achieved by starting the algorithm container is a Docker network that has no internet access.

While the algorithm is thus isolated from the internet, it still has to be able to access several different resources, such as the vantage6 server if it needs to spawn other containers for subtasks. Such communication all takes place over interfaces that are an integral part of vantage6, and are thus considered safe. Below is a list of interfaces that are available to the algorithm container.

- The vantage6 server is available to the algorithm container via a proxy server running on the node.
- The VPN network is available to the algorithm container via the VPN client container.
- The SSH tunnel is available to the algorithm container via the SSH tunnel container.
- The whitelisted addresses are available to the algorithm container via the Squid proxy container.

Note that all of these connections are initiated from the algorithm container. Vantage6 does not support incoming connections to the algorithm container.

4.8.4 Communication between components

The following pages each describe one way that is used to communicate between different vantage6 components.

4.8.4.1 SocketIO connection

A SocketIO connection is a bidirectional, persistent, event-based communication line. In vantage6, it is used for example to send status updates from the server to the nodes or to send a signal to a node that it should kill a task.

Each socketIO connection consists of a server and one or more clients. The clients can only send a message to the server and not to each other. The server can send messages to all clients or to a specific client. In vantage6, the central server is the socketIO server; the clients can be nodes or users.

Note: The vantage6 user interface automatically establishes a socketIO connection with the server when the user logs in. The user can then view the updates they are allowed to see.

Permissions

The socketIO connection is split into different rooms. The vantage6 server decides which rooms a client is allowed to join; they will only be able to read messages from that room.

Nodes always join the room of their own collaboration, and a room of all nodes. Users only join the room of collaborations whose events they are allowed to view which is checked via event view rules.

Usage in vantage6

The server sends the following events to the clients:

- Notify nodes a new task is available
- Letting nodes and users know if a node in their collaboration comes online or goes offline
- Instructing nodes to renew their token if it is expired
- Letting nodes and users know if a task changed state on a node (e.g. started, finished, failed). This is especially important for nodes to know in case an algorithm they are running depends on the output of another node.
- Instruct nodes to kill one or more tasks
- Checking if nodes are still alive

The nodes send the following events to the server:

- Alert the server of task state changes (e.g. started, finished, failed)
- Share information about the node configuration (e.g. which algorithms are allowed to run on the node)

In theory, users could use their socketIO connection to send events, but none of the events they send will lead to action on the server.

4.8.4.2 End to end encryption

Encryption in vantage6 is handled at organization level. Whether encryption is used or not, is set at collaboration level. All the nodes in the collaboration need to agree on this setting. You can enable or disable encryption in the node configuration file, see the example in *All configuration options*.

The encryption module encrypts data so that the server is unable to read communication between users and nodes. The only messages that go from one organization to another through the server are computation requests and their results. Only the algorithm input and output are encrypted. Other metadata (e.g. time started, finished, etc), can be read by the server.

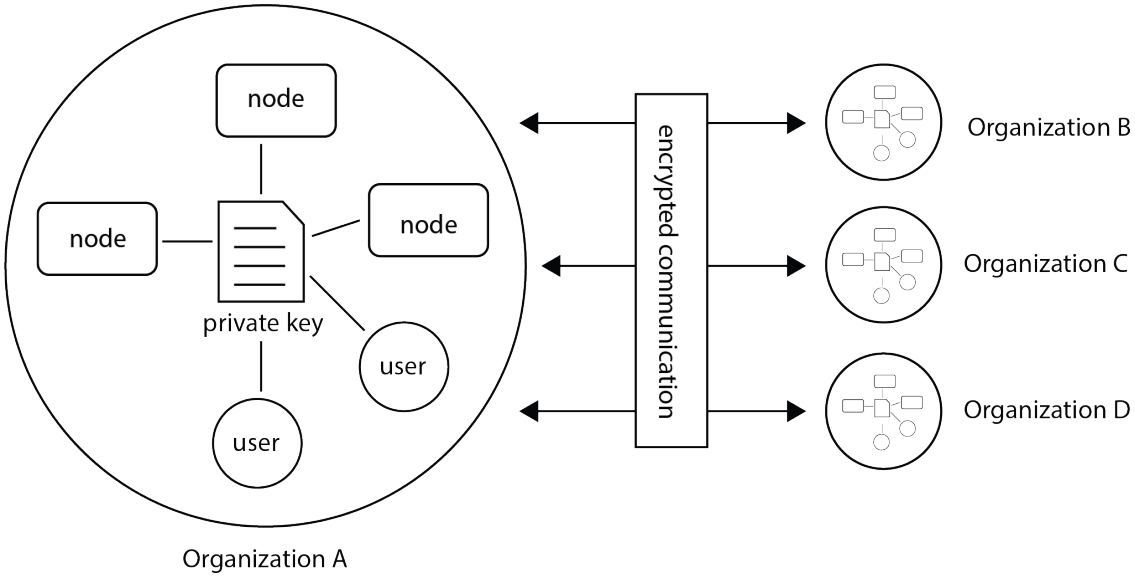


Fig. 4.8: Encryption takes place between organizations therefore all nodes and users from the a single organization should use the same private key.

The encryption module uses RSA keys. The public key is uploaded to the vantage6 server. Tasks and other users can use this public key (this is automatically handled by the python-client and R-client) to send messages to the other parties.

Note: The RSA key is used to create a shared secret which is used for encryption and decryption of the payload.

When the node starts, it checks that the public key stored at the server is derived from the local private key. If this is not the case, the node will replace the public key at the server.

Warning: If an organization has multiple nodes and/or users, they must use the same private key.

In case you want to generate a new private key, you can use the command `v6 node create-private-key`. If a key already exists at the local system, the existing key is reused (unless you use the `--force` flag). This way, it is easy to configure multiple nodes to use the same key.

It is also possible to generate the key yourself and upload it by using the endpoint `https://SERVER[/api_path]/organization/<ID>`.

4.8.4.3 Algorithm-to-algorithm VPN comunication

Since version 3.0.0

Originally, all communication in the vantage6 infrastructure occurs via the central server. Algorithms and nodes could not directly communicate with one another. Since version 3.0.0, algorithms can communicate with one another directly, without the need to go through the central server. This is achieved by connecting the nodes to a VPN network.

The implementation of algorithm-to-algorithm communication in vantage6 is discussed at length in this paper.

When to use

Some algorithms require a lot of communication between algorithm containers before a solution is achieved. For example, there are algorithms that uses iterative methods to optimize a solution, or algorithms that share partial machine learning models with one another in the learning process.

For such algorithms, using the default communication method (via the central server) can be very inefficient. Also, some popular libraries assume that direct communication between algorithm containers is possible. These libraries would have to be adapted specifically for the vantage6 infrastructure, which is not always feasible. In such cases, it is better to setup a VPN connection to allow algorithm containers to communicate directly with one another.

Another reason to use a VPN connection is that for some algorithms, routing all partial results through the central server can be undeseriable. For example, with many algorithms using an MPC protocol, it may be possible for the central party to reconstruct the original data if they have access to all partial results.

How to use

In order to use a VPN connection, a VPN server must be set up, and the vantage6 server and nodes must be configured to use this VPN server. Below we detail How this can be done.

Installing a VPN server

To use algorithm-to-algorithm communication, a VPN server must be set up by the server administrator. The installation instructions for the VPN server are *here*.

Configuring the vantage6 server

The vantage6 server must be configured to use the VPN server. This is done by adding the following configuration snippet to the configuration file.

```
vpn_server:
    # the URL of your VPN server
    url: https://your-vpn-server.ext

    # OATH2 settings, make sure these are the same as in the
    # configuration file of your EduVPN instance
    redirect_url: http://localhost
    client_id: your_VPN_client_user_name
    client_secret: your_VPN_client_user_password

    # Username and password to acccess the EduVPN portal
```

(continues on next page)

(continued from previous page)

```
    portal_username: your_eduvpn_portal_user_name
    portal_userpass: your_eduvpn_portal_user_password
```

Note that the vantage6 server does not connect to the VPN server itself. It uses the configuration above to provide nodes with a VPN configuration file when they want to connect to VPN.

Configuring the vantage6 node

A node administrator has to configure the node to use the VPN server. This is done by adding the following configuration snippet to the configuration file.

```
vpn_subnet: '10.76.0.0/16'
```

This snippet should include the subnet on which the node will connect to the VPN network, and should be part of the subnet range of the VPN server. Node administrators should consult the VPN server administrator to determine which subnet range to use.

If all configuration is properly set up, the node will automatically connect to the VPN network on startup.

> **Warning:** If the node fails to connect to the VPN network, the node will not stop. It will print a warning message and continue to run.

Note: Nodes that connect to a vantage6 server with VPN do not necessarily have to connect to the VPN server themselves: they may be involved in a collaboration that does not require VPN.

How to test the VPN connection

This algorithm can be used to test the VPN connection. The script *test_on_v6.py* in this repository can be used to send a test task which will print whether echoes over the VPN network are working.

Use VPN in your algorithm

If you are using the Python algorithm client, you can call the following function:

```
client.vpn.get_addresses()
```

which will return a dictionary containing the VPN IP address and port of each of the algorithms running that task.

> **Warning:** If you are using the old algorithm client `ContainerClient` (which is the default in vantage6 3.x), you should use `client.get_algorithm_addresses()` instead.

If you are not using the algorithm client, you can send a request to to the endpoint `/vpn/algorithm/addresses` on the vantage6 server (via the node proxy server), which will return a dictionary containing the VPN IP address and port of each of the algorithms running that task.

How does it work?

As mentioned before, the implementation of algorithm-to-algorithm communication is discussed at length in this paper. Below, we will give a brief overview of the implementation.

On startup, the node requests a VPN configuration file from the vantage6 server. The node first checks if it already has a VPN configuration file and if so, it will try to use that. If connecting with the existing configuration file fails, it will try to renew the configuration file's keypair by calling `/vpn/update`. If that fails, or if no configuration file is present yet (e.g. on first startup of a node), the node will request a new configuration file by calling `/vpn`.

The VPN configuration file is an `.ovpn` file that is passed to a VPN client container that establishes the VPN connection. This VPN client container keeps running in the background for as long as the node is running.

When the VPN client container is started, a few network rules are changed on the host machine to forward the incoming traffic on the VPN subnet to the VPN client container. This is necessary because the VPN traffic will otherwise never reach the vantage6 containers. The VPN client container is configured to drop any traffic that does not originate from the VPN connection.

When a task is started, the vantage6 node determines how many ports that particular algorithm requires on the local Docker network. It determines which ports are available and then assigns those ports to the algorithm. The node then stores the VPN IP address and the assigned ports in the database. Also, it configures the local Docker network such that the VPN client container forwards all incoming traffic for algorithm containers to the right port on the right algorithm container. *Vice versa*, the VPN client container is configured to forward outgoing traffic over the VPN network to the right addresses.

Only when the all this configuration is completed, is the algorithm container started.

4.9 Developer community

As an open-source platform, we welcome anyone who would like to contribute to the vantage6 code and/or documentation. The following sections are meant to clarify our processes in development, documentation and releasing.

4.9.1 Contribute

4.9.1.1 Support questions

If you have questions, you can use

- Github discussions
- Ask us on Discord

We prefer that you ask questions via these routes rather than creating Github issues. The issue tracker is intended to address bugs, feature requests, and code changes.

4.9.1.2 Reporting issues

Issues can be posted at our Github issue page.

We distinguish between the following types of issues:

- Bug report: you encountered broken code
- Feature request: you want something to be added
- Change request: there is a something you would like to be different but it is not considered a new feature nor is something broken
- Security vulnerabilities: you found a security issue

Each issue type has its own template. Using these templates makes it easier for us to manage them.

> **Warning:** Security vulnerabilities should not be reported in the Github issue tracker as they should not be publically visible. To see how we deal with security vulnerabilities read our policy.
>
> See the *Security vulnerabilities* section when you want to release a security patch yourself.

We distibute the open issues in sprints and hotfixes. You can check out these boards here:

- Sprints
- Hotfixes

When a high impact bug is reported, we will put it on the hotfix board and create a patch release as soon as possible.

The sprint board tracks which issues we plan to fix in which upcoming release. Low-impact bugs, new features and changes will be scheduled into a sprint periodically. We automatically assign the label 'new' to all newly reported issues to track which issues should still be scheduled.

If you would like to fix an existing bug or create a new feature, check *Submitting patches* for more details on e.g. how to set up a local development environment and how the release process works. We prefer that you let us know you what are working on so we prevent duplicate work.

4.9.1.3 Security vulnerabilities

If you are a member of the Vantage6 Github organization, you can create an security advisory in the Security tab. See Table 4.2 on what to fill in.

If you are not a member, please reach out directly to Frank Martin and/or Bart van Beusekom, or any other project member. They can then create a security advisory for you.

Table 4.2: Advisory details

Name	Details
Ecosystem	Set to `pip`
Package name	Set to `vantage6`
Affected versions	Specify the versions (or set of verions) that are affected
Patched version	Version where the issue is addessed, you can fill this in later when the patch is released.
Severity	Determine severity score using this tool. Then use table Table 4.3 to determine the level from this score.
Common weakness enumerator (CWE)	Find the CWE (or multiple) on this website.

Table 4.3: Severity

Score	Level
0.1-3.9	Low
4.0-6.9	Medium
7.0-8.9	High
9.0-10.0	Critical

Once the advisory has been created it is possible to create a private fork from there (Look for the button `Start a temporary private fork`). This private fork should be used to solve the issue.

From the same page you should request a CVE number so we can alert dependent software projects. Github will review the request. We are not sure what this entails, but so far they approved all advisories.

4.9.1.4 Community Meetings

We host bi-monthly community meetings intended for aligning development efforts. Anyone is welcome to join although they are mainly intended for infrastructure and algorithm developers. There is an opportunity to present what your team is working on an find collaboration partners.

Community meetings are usually held on the third Thursday of the month at 11:00 AM CET on Microsoft Teams. Reach out on Discord if you want to join the community meeting.

For more information and slides from previous meetings, check our website.

4.9.1.5 Submitting patches

If there is not an open issue for what you want to submit, please open one for discussion before submitting the PR. We encourage you to reach out to us on Discord, so that we can work together to ensure your contribution is added to the repository.

The workflow below is specific to the vantage6 infrastructure repository. However, the concepts for our other repositories are the same. Then, modify the links below and ignore steps that may be irrelevant to that particular repository.

Setup your environment

- Make sure you have a Github account
- Install and configure `git` and `make`
- (Optional) install and configure Miniconda
- Clone the main repository locally:

```
git clone https://github.com/vantage6/vantage6
cd vantage6
```

- Add your fork as a remote to push your work to. Replace `{username}` with your username.

```
git remote add fork https://github.com/{username}/vantage6
```

- Create a virtual environment to work in. If you are using miniconda:

```
conda create -n vantage6 python=3.10
conda activate vantage6
```

It is also possible to use `virtualenv` if you do not have a conda installation.

- Update pip and setuptools

```
python -m pip install --upgrade pip setuptools
```

- Install vantage6 as development environment:

```
make install-dev
```

Coding

First, create a branch you can work on. Make sure you branch of the latest `main` branch:

```
git fetch origin
git checkout -b your-branch-name origin/main
```

Then you can create your bugfix, change or feature. Make sure to commit frequently. Preferably include tests that cover your changes.

Finally, push your commits to your fork on Github and create a pull request.

```
git push --set-upstream fork your-branch-name
```

Code style

We use black to format our code. It is important that you use this style so make sure that your contribution will be easily incorporated into the code base.

Black is automatically installed into your python environment when you run `make install-dev`. To automatically enable black, we recommend that you install the *Black Formatter* extension from Microsoft in the VSCode marketplace. By enabling the option 'format on save' you can then automatically format your code in the proper style when you save a file.

Alternatively, or additionally, you may install a pre-commit hook that will automatically format your code when you commit it. To do so, run the following command:

```
pre-commit install
```

You may need to run `pre-commit autoupdate` to update the pre-commit hook.

Unit tests & coverage

You can execute unit tests using the `test` command in the Makefile:

```
make test
```

If you want to execute a specific unit test (e.g. the one you just created or one that is failing), you can use a command like:

```
python -m unittest tests_folder.test_filename.TestClassName.test_name
```

This command assumes you are in the directory above `tests_folder`. If you are inside the `tests_folder`, then you should remove that part.

Verifying local code changes

While working on a new feature, it can be useful to run a server and/or nodes locally with your code changes to verify that it does what you expect it to do. This can be done by using the commands `v6 server` and `v6 node` in combination with the options `--mount-src` and optionally `--image`.

- The `--mount-src /path/to/vantage6` option will overwrite the code that the server/node runs with your local code when running the docker image. The provided path should point towards the root folder of the vantage6 repository - where you have your local changes.
- The `--image <url_to_docker_image>` can be used to point towards a custom Docker image for the node or server. This is mostly useful when your code update includes dependency upgrades. Then, you need to build a custom infrastructure image as the 'old' image does not contain the new depencey and the `--mount-src` option will only overwrite the source code and not re-install dependencies.

Note: If you are using Docker Desktop (which is usually the case if you are on Windows or MacOS) and want to setup a test environment, you should use `http://host.docker.interal` for the server address in the node configuration file. You should not use `http://localhost` in that case as that points to the localhost within the docker container instead of the system-wide localhost.

Pull Request

Please consider first which branch you want to merge your contribution into. **Patches** are usually directly merged into `main`, but **features** are usually merged into a release branch (e.g. `release/4.1` for version 4.1.0) before being merged into the `main` branch.

Before the PR is merged, it should pass the following requirements:

- At least one approved review of a code owner
- All unit tests should complete
- CodeQL (vulnerability scanning) should pass
- Codacy - Code quality checks - should be OK
- Coveralls - Code coverage analysis - should not decrease

Documentation

Depending on the changes you made, you may need to add a little (or a lot) of documentation. For more information on how and where to edit the documentation, see the section *Documentation*.

Consider which documentation you need to update:

- **User documentation.** Update it if your change led to a different expierence for the end-user
- **Technical documentation.** Update it if you added new functionality. Check if your function docstrings have also been added (see last bullet below).
- **OAS (Open API Specification).** If you changed input/output for any of the API endpoints, make sure to add it to the docstrings. See *API Documenation with OAS3+* for more details.
- **Function docstrings** These should always be documented using the numpy format. Such docstrings can then be used to automatically generate parts of the technical documentation space.

4.9.1.6 Roles in the vantage6 community

As an open-source community, vantage6 is open to constructive development efforts from anyone. Developers that contribute regularly may at some point become official members and as such can get more permissions. This section outlines the rules that we follow as a community to govern this process.

Community access tiers

A few levels of access are discerned within the vantage6 community:

- **Contributors**: people that have opened pull requests which have been merged
- **Members**: members of the vantage6 Github organization
- **Administrators**: administrators of the vantage6 Github organization

Contributor access is available to anyone that wants to contribute to vantage6. They can create their own forks of the vantage6 repository and create pull requests from there.

Membership gives developers more extensive access, for instance to create branches within the official repository and view private repositories within the vantage6 Github organization. Membership may be given to anyone that requests it and will be granted if the majority of the vantage6 members approves of this. There are no hard requirements for membership: usually, making several contributions helps in receiving membership, but someone may also attain membership if they are, for instance, an employee of a trusted organization that plans to invest in vantage6.

Administrator level access gives developers access to merge pull requests into the main branch and execute other sensitive actions within the repositories. This level of access will only be granted to a small number of developers that have demonstrated their knowledge of vantage6 extensively. Administrator access will only be given if all administrators agree unanimously that it should be granted. In rare cases, administrator access may also be revoked if the other administrators unanimously agree that it should be revoked.

Voting for membership and administrator access may be done in the community meetings, but can also be done asynchronously via email.

4.9.2 Documentation

The vantage6 framework is documented on this website. Additionally, there is *API Documenation with OAS3+*. This documentation is shipped directly with the server instance. All of these documentation pages are described in more detail below.

4.9.2.1 How this documentation is created

The source of the documentation you are currently reading is located here, in the `docs` folder of the *vantage6* repository itself.

To build the documentation locally, there are two options. To build a static version, you can do `make html` when you are in the `docs` directory. If you want to automatically refresh the documentation whenever you make a change, you can use sphinx-autobuild. Assuming you are in the main directory of the repository, run the following commands:

```
pip install -r docs/requirements.txt
sphinx-autobuild docs docs/_build/html --watch .
```

Of course, you only have to install the requirements if you had not done so before.

Note: This documentation also includes some UML diagrams which are generated using PlantUML. To generate these diagrams, you need to install Java. PlantUML itself is included in the Python requirements, so you do not have to install it separately.

Then you can access the documentation on `http://127.0.0.1:8000`. The `--watch` option makes sure that if you make changes to either the documentation text or the docstrings, the documentation pages will also be reloaded.

This documentation is automatically built and published on a commit (on certain branches, including `main`). Both Frank and Bart have access to the vantage6 project when logged into readthedocs. Here they can manage which branches are to be synced, manage the webhook used to trigger a build, and some other -less important- settings.

The files in this documentation use the `rst` format, to see the syntax view this cheatsheet.

4.9.2.2 API Documenation with OAS3+

The API documentation is hosted at the server at the `/apidocs` endpoint. This documentation is generated from the docstrings using Flasgger. The source of this documentation can be found in the docstrings of the API functions.

If you are unfamiliar with OAS3+, note that it was formerly known as Swagger.

An example of such a docsting:

```
"""Summary of the endpoint
   ---
   description: >-
       Short description on what the endpoint does, and which users have
       access or which permissions are required.

   parameters:
       - in: path
         name: id
         schema:
           type: integer
         description: some identifier
         required: true

   responses:
       200:
           description: Ok
       401:
           description: Unauthorized or missing permission

   security:
       - bearerAuth: []

   tags: ["Group"]
"""
```

4.9.3 Release

This page is intended to provide information about our release process. First, we discuss the version formatting, after which we discuss the actual creation and distribution of a release.

4.9.3.1 Version format

Semantic versioning is used: `Major.Minor.Patch.Pre[N].Post<n>`.

Major releases update the first digit, e.g. `1.2.3` is updated to `2.0.0`. This is used for releasing breaking changes: server and nodes of version 2.x.y are unlikely to be able to run an algorithm written for version 1.x.y. Also, the responses of the central server API may change in a way that changes the response to client requests.

Minor releases update the second digit, e.g. `1.2.3` to `1.3.0`. This is used for releasing new features (e.g. a new endpoint), enhancements and other changes that are compatible with all other components. Algorithms written for version``1.x.y`` should run on any server of version `1.z.a`. Also, the central server API should be compatible with other minor versions - the same fields present before will be present in the new version, although new fields may be added. However, nodes and servers of different minor versions may not be able to communicate properly.

Patch releases update the third digit, e.g. `1.2.3` to `1.2.4`. This is used for bugfixes and other minor changes. Different patch releases should be compatible with each other, so a node of version `1.2.3` should be able to communicate with a server of version `1.2.4`.

Pre[N] is used for alpha (a), beta (b) and release candidates (rc) releases and the build number is appended (e.g. `2.0.1b1` indicates the first beta-build of version `2.0.1`). These releases are used for testing before the actual release is made.

Post[N] is used for a rebuild where no code changes have been made, but where, for example, a dependency has been updated and a rebuild is required. In vantage6, this is only used to version the Docker images that are updated in these cases.

4.9.3.2 Testing a release

Before a release is made, it is tested by the development team. They go through the following steps to test a release:

1. *Create a release candidate*. This process is the same as creating the *actual release*, except that the candidate has a 'pre' tag (e.g. `1.2.3rc1` for release candidate number 1 of version 1.2.3). Note that for an RC release, no notifications are sent to Discord.
2. *Install the release*. The release should be tested from a clean conda environment.

```
conda create -n <name> python=3.10
conda activate <name>
pip install vantage6==<version>
```

3. *Start server and node*. Start the server and node for the release candidate:

```
v6 server start --name <server name>
    --image harbor2.vantage6.ai/infrastructure/server:<version>
    --attach
v6 node start --name <node name>
    --image harbor2.vantage6.ai/infrastructure/node:<version>
    --attach
```

4. *Test code changes*. Go through all issues that are part of the new release and test their functionality.

5. *Run test algorithms*. The algorithm *v6-feature-tester* is run and checked. This algorithm checks several features to see if they are performing as expected. Additionaly, the *v6-node-to-node-diagnostics* algorithm is run to check the VPN functionality.
6. *Check swagger*. Check if the swagger docs run without error. They should be on http://localhost:5000/apidocs/ when running server locally.
7. *Test the UI*. Also make a release candidate there.

After these steps, the release is ready. It is executed for both the main infrastructure and the UI. The release process is described below.

Note: We are working on further automating the testing and release process.

4.9.3.3 Create a release

To create a new release, one should go through the following steps:

- Check out the correct branch of the vantage6 repository and pull the latest version:

```
git checkout main
git pull
```

  Make sure the branch is up-to-date. **Patches** are usually directly merged into main, but for **minor** or **major** releases you usually need to execute a pull request from a development branch.

- Create a tag for the release. See *Version format* for more details on version names:

```
git tag version/x.y.z
```

- Push the tag to the remote. This will trigger the release pipeline on Github:

```
git push origin version/x.y.z
```

Note: The release process is protected and can only be executed by certain people. Reach out if you have any questions regarding this.

4.9.3.4 The release pipeline

The release pipeline executes the following steps:

1. It checks if the tag contains a valid version specification. If it does not, the process it stopped.
2. Update the version in the repository code to the version specified in the tag and commit this back to the main branch.
3. Install the dependencies and build the Python package.
4. Upload the package to PyPi.
5. Build and push the Docker image to harbor2.vantage6.ai.
6. Post a message in Discord to alert the community of the new release. This is not done if the version is a pre-release (e.g. version/x.y.0rc1).

Note: If you specify a tag with a version that already exists, the build pipeline will fail as the upload to PyPi is rejected.

The release pipeline uses a number of environment variables to, for instance, authenticate to PyPi and Discord. These variables are listed and explained in the table below.

Table 4.4: Environment variables

Secret	Description
COMMIT_PAT	Github Personal Access Token with commit privileges. This is linked to an individual user with admin right as the commit on the `main` needs to bypass the protections. There is unfortunately not -yet- a good solution for this.
ADD_TO_PROJECT_PAT	Github Personal Access Token with project management privileges. This token is used to add new issues to project boards.
COVERALLS_TOKEN	Token from coveralls to post the test coverage stats.
DOCKER_TOKEN	Token used together DOCKER_USERNAME to upload the container images to our https://harbor2.vantage6.ai.
DOCKER_USERNAME	See DOCKER_TOKEN.
PYPI_TOKEN	Token used to upload the Python packages to PyPi.
DISCORD_RELEASE_TOKEN	Token to post a message to the Discord community when a new release is published.

4.9.3.5 Distribute release

Nodes and servers that are already running will automatically be upgraded to the latest version of their major release when they are restarted. This happens by pulling the newly released docker image. Note that the major release is never automatically updated: for example, a node running version 2.1.0 will update to 2.1.1 or 2.2.0, but never to 3.0.0. Depending on the version of Vantage6 that is being used, there is a reserved Docker image tag for distributing the upgrades. These are the following:

Tag	Description
cotopaxi	`4.x.x` release
petronas	`3.x.x` release
harukas	`2.x.x` release
troltunga	`1.x.x` release

Docker images can be pulled manually with e.g.

```
docker pull harbor2.vantage6.ai/infrastructure/server:cotopaxi
docker pull harbor2.vantage6.ai/infrastructure/node:3.1.0
```

4.9.3.6 User Interface release

The release process for the user interface (UI) is very similar to the release of the infrastructure detailed above. The same versioning format is used, and when you push a version tag, the automated release process is triggered.

We have semi-synchronized the version of the UI with that of the infrastructure. That is, we try to release major and minor versions at the same time. For example, if we are currently at version 3.5 and release version 3.6, we release it both for the infrastructure and for the UI. However, there may be different patch versions for both: the latest version for the infrastructure may then be 3.6.2 while the UI may still be at 3.6.

The release pipeline for the UI executes the following steps:

1. Version tag is verified (same as infrastructure).
2. Version is updated in the code (same as infrastructure).
3. Application is built.
4. Docker images are built and released to harbor2.
5. Application is pushed to our UI deployment slot (an Azure app service).

4.9.3.7 Post-release checks

After a release, there are a few checks that are performed. Most of these are only relevant if you are hosting a server yourself that is being automatically updated upon new releases, as is for instance the case for the Cotopaxi server.

For Cotopaxi, the following checks are done:

- Check that harbor2.vantage6.ai has updated images `server:cotopaxi`, `server:cotopaxi-live` and `node:cotopaxi`.
- Check if the (live) server version is updated. Go to: https://cotopaxi.vantage6.ai/version. Check logs if it is not updated.
- Release any documentation that may not yet have been released.
- Upgrade issue status to 'Done' in any relevant issue tracker.
- Check if nodes are online, and restart them to update to the latest version if desired.

4.10 Function documentation

This part of the documentation documents the code of the vantage6 infrastructure. It lists all the functions and classes and describes what they do and how they may be used. It is ordered by package: each of the subsections below represents a distinct PyPi package.

4.10.1 Node

A node in its simplest would retrieve a task from the central server by an API call, run this task and finally return the results to the central server again.

The node application runs four threads:

Main thread
: Checks the task queue and run the next task if there is one available.

Listening thread
: Listens for incoming websocket messages. Among other functionality, it adds new tasks to the task queue.

Speaking thread
: Waits for tasks to finish. When they do, return the results to the central server.

Proxy server thread
: Algorithm containers are isolated from the internet for security reasons. The local proxy server provides an interface to the central server for algorithm containers to create subtasks and retrieve their results.

The node connects to the server using a websocket connection. This connection is mainly used for sharing status updates. This avoids the need for polling to see if there are new tasks available.

Below you will find the structure of the classes and functions that comprise the node. A few that we would like to highlight:

- *Node*: the main class in a vantage6 node.
- *NodeContext* and *DockerNodeContext*: classes that handle the node configuration. The latter inherits from the former and adds some properties for when the node runs in a docker container.
- *DockerManager*: Manages the docker containers and networks of the vantage6 node.
- *DockerTaskManager*: Start a docker container that runs an algorithm and manage its lifecycle.
- *VPNManager*: Sets up the VPN connection (if it is configured) and manages it.
- *vnode-local commands*: commands to run non-dockerized (development) instances of your nodes.

4.10.1.1 vantage6.node.Node

class Node(*ctx*)

Authenticates to the central server, setup encryption, a websocket connection, retrieving task that were posted while offline, preparing dataset for usage and finally setup a local proxy server..

Parameters
ctx (NodeContext | DockerNodeContext) – Application context object.

__listening_worker()

Listen for incoming (websocket) messages from the server.

Runs in a separate thread. Received events are handled by the appropriate action handler.

Return type
None

__proxy_server_worker()

Proxy algorithm container communcation.

A proxy for communication between algorithms and central server.

Return type
None

__speaking_worker()

Sending messages to central server.

Routine that is in a seperate thread sending results to the server when they come available.

Return type
None

__start_task(*task_incl_run*)

Start the docker image and notify the server that the task has been started.

Parameters
task_incl_run (*dict*) – A dictionary with information required to run the algorithm

Return type
None

`authenticate()`

Authenticate with the server using the api-key from the configuration file. If the server rejects for any reason -other than a wrong API key- serveral attempts are taken to retry.

Return type
None

`connect_to_socket()`

Create long-lasting websocket connection with the server. The connection is used to receive status updates, such as new tasks.

Return type
None

`get_task_and_add_to_queue(task_id)`

Fetches (open) task with task_id from the server. The *task_id* is delivered by the websocket-connection.

Parameters
task_id (*int*) – Task identifier

Return type
None

`initialize()`

Initialization of the node

Return type
None

`kill_containers(kill_info)`

Kill containers on instruction from socket event

Parameters
kill_info (*dict*) – Dictionary received over websocket with instructions for which tasks to kill

Returns
List of dictionaries with information on killed task (keys: run_id, task_id and parent_id)

Return type
list[dict]

`private_key_filename()`

Get the path to the private key.

Return type
Path

`run_forever()`

Keep checking queue for incoming tasks (and execute them).

Return type
None

`setup_encryption()`

Setup encryption if the node is part of encrypted collaboration

Return type
None

setup_squid_proxy(*isolated_network_mgr*)

Initiates a Squid proxy if configured in the config.yml

Expects the configuration in the following format:

```
whitelist:
    domains:
        - domain1
        - domain2
    ips:
        - ip1
        - ip2
    ports:
        - port1
        - port2
```

Parameters
isolated_network_mgr (NetworkManager) – Network manager for isolated network

Returns
Squid proxy instance

Return type
Squid

setup_ssh_tunnels(*isolated_network_mgr*)

Create a SSH tunnels when they are defined in the configuration file. For each tunnel a new container is created. The image used can be specified in the configuration file as *ssh-tunnel* in the *images* section, else the default image is used.

Parameters
isolated_network_mgr (NetworkManager) – Manager for the isolated network

Return type
list[SSHTunnel]

setup_vpn_connection(*isolated_network_mgr*, *ctx*)

Setup container which has a VPN connection

Parameters

- **isolated_network_mgr** (NetworkManager) – Manager for the isolated Docker network
- **ctx** (DockerNodeContext | NodeContext) – Context object for the node

Returns
Manages the VPN connection

Return type
VPNManager

share_node_details()

Share part of the node's configuration with the server.

This helps the other parties in a collaboration to see e.g. which algorithms they are allowed to run on this node.

Return type
None

sync_task_queue_with_server()

Get all unprocessed tasks from the server for this node.

Return type
None

class DockerNodeContext(**args*, ***kwargs*)

Node context for the dockerized version of the node.

static instance_folders(*instance_type*, *instance_name*, *system_folders*)

Log, data and config folders are always mounted. The node manager should take care of this.

set_folders(*instance_type*, *instance_name*, *system_folders*)

In case of the dockerized version we do not want to use user specified directories within the container.

4.10.1.2 vantage6.node.docker.docker_base

class DockerBaseManager(*isolated_network_mgr*, *docker_client=None*)

Base class for docker-using classes. Contains simple methods that are used by multiple derived classes

get_isolated_netw_ip(*container*)

Get address of a container in the isolated network

Parameters
container (*Container*) – Docker container whose IP address should be obtained

Returns
IP address of a container in isolated network

Return type
str

4.10.1.3 vantage6.node.docker.docker_manager

class DockerManager(*ctx*, *isolated_network_mgr*, *vpn_manager*, *tasks_dir*, *client*, *proxy=None*)

Bases: *DockerBaseManager*

Wrapper for the docker-py module.

This class manages tasks related to Docker, such as logging in to docker registries, managing input/output files, logs etc. Results can be retrieved through *get_result()* which returns the first available algorithm result.

cleanup()

Stop all active tasks and delete the isolated network

Note: the temporary docker volumes are kept as they may still be used by a parent container

Return type
None

cleanup_tasks()

Stop all active tasks

Returns
List of information on tasks that have been killed

Return type
list[KilledRun]

`create_volume`(*volume_name*)

Create a temporary volume for a single run.

A single run can consist of multiple algorithm containers. It is important to note that all algorithm containers having the same job_id have access to this container.

Parameters
`volume_name` (*str*) – Name of the volume to be created

Return type
None

`get_column_names`(*label*, *type_*)

Get column names from a node database

Parameters

- `label` (*str*) – Label of the database
- `type` (*str*) – Type of the database

Returns
List of column names

Return type
list[str]

`get_result`()

Returns the oldest (FIFO) finished docker container.

This is a blocking method until a finished container shows up. Once the container is obtained and the results are read, the container is removed from the docker environment.

Returns
result of the docker image

Return type
Result

`is_docker_image_allowed`(*docker_image_name*, *task_info*)

Checks the docker image name.

Against a list of regular expressions as defined in the configuration file. If no expressions are defined, all docker images are accepted.

Parameters

- `docker_image_name` (*str*) – uri to the docker image
- `task_info` (*dict*) – Dictionary with information about the task

Returns
Whether docker image is allowed or not

Return type
bool

`is_running`(*run_id*)

Check if a container is already running for <run_id>.

Parameters
`run_id` (*int*) – run_id of the algorithm container to be found

Returns
Whether or not algorithm container is running already

Return type
bool

kill_selected_tasks(*org_id*, *kill_list=None*)

Kill tasks specified by a kill list, if they are currently running on this node

Parameters

- **org_id** (*int*) – The organization id of this node
- **kill_list** (*list[ToBeKilled]*) – A list of info about tasks that should be killed.

Returns
List with information on killed tasks

Return type
list[KilledRun]

kill_tasks(*org_id*, *kill_list=None*)

Kill tasks currently running on this node.

Parameters

- **org_id** (*int*) – The organization id of this node
- **kill_list** (*list[ToBeKilled] (optional)*) – A list of info on tasks that should be killed. If the list is not specified, all running algorithm containers will be killed.

Returns
List of dictionaries with information on killed tasks

Return type
list[KilledRun]

link_container_to_network(*container_name*, *config_alias*)

Link a docker container to the isolated docker network

Parameters

- **container_name** (*str*) – Name of the docker container to be linked to the network
- **config_alias** (*str*) – Alias of the docker container defined in the config file

Return type
None

login_to_registries(*registries=[]*)

Login to the docker registries

Parameters
registries (*list*) – list of registries to login to

Return type
None

run(*run_id*, *task_info*, *image*, *docker_input*, *tmp_vol_name*, *token*, *databases_to_use*)

Checks if docker task is running. If not, creates DockerTaskManager to run the task

Parameters

- **run_id** (*int*) – Server run identifier

- **task_info** (*dict*) – Dictionary with task information
- **image** (*str*) – Docker image name
- **docker_input** (*bytes*) – Input that can be read by docker container
- **tmp_vol_name** (*str*) – Name of temporary docker volume assigned to the algorithm
- **token** (*str*) – Bearer token that the container can use
- **databases_to_use** (*list[str]*) – Labels of the databases to use

Returns
Returns a tuple with the status of the task and a description of each port on the VPN client that forwards traffic to the algorithm container (`None` if VPN is not set up).

Return type
TaskStatus, list[dict] | None

class Result(*run_id: int*, *task_id: int*, *logs: str*, *data: str*, *status: str*, *parent_id: int | None*)

Data class to store the result of the docker image.

Variables

- **run_id** (*int*) – ID of the current algorithm run
- **logs** (*str*) – Logs attached to current algorithm run
- **data** (*str*) – Output data of the algorithm
- **status_code** (*int*) – Status code of the algorithm run

4.10.1.4 vantage6.node.docker.task_manager

class DockerTaskManager(*image*, *docker_client*, *vpn_manager*, *node_name*, *run_id*, *task_info*, *tasks_dir*, *isolated_network_mgr*, *databases*, *docker_volume_name*, *alpine_image=None*, *proxy=None*, *device_requests=None*)

Bases: *DockerBaseManager*

Manager for running a vantage6 algorithm container within docker.

Ensures that the environment is properly set up (docker volumes, directories, environment variables, etc). Then runs the algorithm as a docker container. Finally, it monitors the container state and can return it's results when the algorithm finished.

cleanup()

Cleanup the containers generated for this task

Return type
None

get_results()

Read results output file of the algorithm container

Returns
Results of the algorithm container

Return type
bytes

`is_finished()`

Checks if algorithm container is finished

Returns
True if algorithm container is finished

Return type
bool

`pull(local_exists)`

Pull the latest docker image.

Parameters
local_exists (*bool*) – Whether the image already exists locally

Raises
PermanentAlgorithmStartFail – If the image could not be pulled and does not exist locally

Return type
None

`report_status()`

Checks if algorithm has exited successfully. If not, it prints an error message

Returns
logs – Log messages of the algorithm container

Return type
str

`run(docker_input, tmp_vol_name, token, algorithm_env, databases_to_use)`

Runs the docker-image in detached mode.

It will will attach all mounts (input, output and datafile) to the docker image. And will supply some environment variables.

Parameters

- **docker_input** (*bytes*) – Input that can be read by docker container
- **tmp_vol_name** (*str*) – Name of temporary docker volume assigned to the algorithm
- **token** (*str*) – Bearer token that the container can use
- **algorithm_env** (*dict*) – Dictionary with additional environment variables to set
- **databases_to_use** (*list[str]*) – List of labels of databases to use in the task

Returns
Description of each port on the VPN client that forwards traffic to the algo container. None if VPN is not set up.

Return type
list[dict] | None

4.10.1.5 vantage6.node.docker.vpn_manager

class VPNManager(*isolated_network_mgr*, *node_name*, *node_client*, *vpn_volume_name*, *vpn_subnet*, *alpine_image=None*, *vpn_client_image=None*, *network_config_image=None*)

Bases: DockerBaseManager

Setup a VPN client in a Docker container and configure the network so that the VPN container can forward traffic to and from algorithm containers.

connect_vpn()

Start VPN client container and configure network to allow algorithm-to-algoritm communication

Return type
None

exit_vpn(*cleanup_host_rules=True*)

Gracefully shutdown the VPN and clean up

Parameters
cleanup_host_rules (*bool, optional*) – Whether or not to clear host configuration rules. Should be True if they have been created at the time this function runs.

Return type
None

forward_vpn_traffic(*helper_container*, *algo_image_name*)

Setup rules so that traffic is properly forwarded between the VPN container and the algorithm container (and its helper container)

Parameters

- **algo_helper_container** (*Container*) – Helper algorithm container
- **algo_image_name** (*str*) – Name of algorithm image that is run

Returns
Description of each port on the VPN client that forwards traffic to the algo container. None if VPN is not set up.

Return type
list[dict] | None

get_vpn_ip()

Get VPN IP address in VPN server namespace

Returns
IP address assigned to VPN client container by VPN server

Return type
str

has_connection()

Return True if VPN connection is active

Returns
True if VPN connection is active, False otherwise

Return type
bool

static is_isolated_interface(*ip_interface*, *vpn_ip_isolated_netw*)

Return True if a network interface is the isolated network interface. Identify this based on the IP address of the VPN client in the isolated network

Parameters

- **ip_interface** (*dict*) – IP interface obtained by executing *ip –json addr* command
- **vpn_ip_isolated_netw** (*str*) – IP address of VPN container in isolated network

Returns
True if this is the interface describing the isolated network

Return type
bool

send_vpn_ip_to_server()

Send VPN IP address to the server

Return type
None

4.10.1.6 vantage6.node.docker.exceptions

Below are some custom exception types that are raised when algorithms cannot be executed successfully.

exception AlgorithmContainerNotFound

Algorithm container was lost. Potentially running it again would resolve the issue.

exception PermanentAlgorithmStartFail

Algorithm failed to start and should not be attempted to be started again.

exception UnknownAlgorithmStartFail

Algorithm failed to start due to an unknown reason. Potentially running it again would resolve the issue.

4.10.1.7 vantage6.node.proxy_server

This module contains a proxy server implementation that the node uses to communicate with the server. It contains general methods for any routes, and methods to handle tasks and results, including their encryption and decryption.

(!) Not to be confused with the squid proxy that allows algorithm containers to access other places in the network.

decrypt_result(*run*)

Decrypt the *result* from a run dictonary

Parameters
run (*dict*) – Run dict

Returns
Run dict with the *result* decrypted

Return type
dict

get_method(*method*)

Obtain http method based on string identifier

Parameters
method (*str*) – Http method requested

Returns
HTTP method

Return type
function

`get_response_json_and_handle_exceptions`(*response*)

Obtain json content from request response

Parameters
response (*requests.Response*) – Requests response object

Returns
Dict containing the json body

Return type
dict | None

`make_proxied_request`(*endpoint*)

Helper to create proxies requests to the central server.

Parameters
endpoint (*str*) – endpoint to be reached at the vantage6 server

Returns
Response from the vantage6 server

Return type
requests.Response

`make_request`(*method*, *endpoint*, *json=None*, *params=None*, *headers=None*)

Make request to the central server

Parameters

- **method** (*str*) – HTTP method to be used
- **endpoint** (*str*) – endpoint of the vantage6 server
- **json** (*dict, optional*) – JSON body
- **params** (*dict, optional*) – HTTP parameters
- **headers** (*dict, optional*) – HTTP headers

Returns
Response from the vantage6 server

Return type
requests.Response

`proxy`(*central_server_path*)

Generalized http proxy request

Parameters
central_server_path (*str*) – The endpoint on the server to be reached

Returns
Contains the server response

Return type
requests.Response

proxy_result()

Obtain and decrypt all results to belong to a certain task

Parameters
id (*int*) – Task id from which the results need to be obtained

Returns
Reponse from the vantage6 server

Return type
requests.Response

proxy_results(*id_*)

Obtain and decrypt the algorithm result from the vantage6 server to be used by an algorithm container.

Parameters
id (*int*) – Id of the result to be obtained

Returns
Response of the vantage6 server

Return type
requests.Response

proxy_task()

Proxy to create tasks at the vantage6 server

Returns
Response from the vantage6 server

Return type
requests.Response

4.10.1.8 vantage6.node.cli.node

This contains the `vnode-local` commands. These commands are similar to the `v6 node` CLI commands, but they start up the node outside of a Docker container, and are mostly intended for development purposes.

Some commands, such as `vnode-local start`, are used within the Docker container when `v6 node start` is used.

vnode-local

Command *vnode-local*.

```
vnode-local [OPTIONS] COMMAND [ARGS]...
```

files

Print out the paths of important files.

If the specified configuration cannot be found, it exits. Otherwise it returns the absolute path to the output.

```
vnode-local files [OPTIONS]
```

Options

-n, --name <name>
Configuration name

--system
Use configuration from system folders (default)

--user
Use configuration from user folders

list

Lists all nodes in the default configuration directories.

```
vnode-local list [OPTIONS]
```

new

Create a new configation file.

Checks if the configuration already exists. If this is not the case a questionaire is invoked to create a new configuration file.

```
vnode-local new [OPTIONS]
```

Options

-n, --name <name>
Configuration name

--system
Use configuration from system folders (default)

--user
Use configuration from user folders

start

Start the node instance.

If no name or config is specified the default.yaml configuation is used. In case the configuration file not exists, a questionaire is invoked to create one.

```
vnode-local start [OPTIONS]
```

Options

-n, --name <name>
Configuration name

-c, --config <config>
Absolute path to configuration-file; overrides "name"

--system
Use configuration from system folders (default)

--user
Use configuration from user folders

--dockerized, -non-dockerized
Whether to use DockerNodeContext or regular NodeContext (default)

version

Returns current version of vantage6 services installed.

```
vnode-local version [OPTIONS]
```

4.10.2 Server

4.10.2.1 Main server class

vantage6.server.ServerApp

4.10.2.2 Starting the server

vantage6.server.run_server

Warning: Note that the `run_server` function is normally not used directly to start the server, but is used as utility function in places that start the server. The recommended way to start a server is using uWSGI as is done in `v6 server start`.

vantage6.server.run_dev_server

4.10.2.3 Permission management

vantage6.server.model.rule.Scope

vantage6.server.model.rule.Operation

vantage6.server.model.permission.RuleCollection

vantage6.server.permission.PermissionManager

4.10.2.4 Socket functionality

vantage6.server.websockets.DefaultSocketNamespace

4.10.2.5 API endpoints

Warning: The API endpoints are also documented on the `/apidocs` endpoint of the server (e.g. `https://cotopaxi.vantage6.ai/apidocs`). That documentation requires a different format than the one used to create this documentation. We are therefore not including the API documentation here. Instead, we merely list the supporting functions and classes.

vantage6.server.resource

vantage6.server.resource.common.output_schema

vantage6.server.resource.common.auth_helper

vantage6.server.resource.common.swagger_template

This module contains the template for the OAS3 documentation of the API.

4.10.2.6 SQLAlchemy models

vantage6.server.model.base

This module contains a few base classes that are used by the other models.

Database models for the API resources

vantage6.server.model.algorithm_port.AlgorithmPort

vantage6.server.model.authenticatable.Authenticatable

vantage6.server.model.collaboration.Collaboration

vantage6.server.model.node.Node

vantage6.server.model.organization.Organization

vantage6.server.model.run.Run

vantage6.server.model.role.Role

vantage6.server.model.rule.Rule

vantage6.server.model.task.Task

vantage6.server.model.user.User

Database models that link resources together

vantage6.server.model.Member

vantage6.server.model.permission

vantage6.server.model.role_rule_association

4.10.2.7 Database utility functions

vantage6.server.db

4.10.2.8 Mail service

vantage6.server.mail_service

4.10.2.9 Default roles

vantage6.server.default_roles

4.10.2.10 Custom server exceptions

vantage6.server.exceptions

4.10.3 Algorithm store

4.10.3.1 Main class of algorithm store

vantage6.algorithm.store

4.10.3.2 API endpoints

Warning: The API endpoints are documented on the `/apidocs` endpoint of the server (e.g. `https://cotopaxi.vantage6.ai/apidocs`). That documentation requires a different format than the one used to create this documentation. We are therefore not including the API documentation here. Instead, we merely list the supporting functions and classes.

vantage6.algorithm.store.resource

vantage6.algorithm.store.resource.schema.output_schema

vantage6.algorithm.store.resource.schema.input_schema

4.10.3.3 SQLAlchemy models

vantage6.algorithm.store.model.base

This module contains a few base classes that are used by the other models.

Database models for the API resources

vantage6.algorithm.store.model.algorithm

vantage6.algorithm.store.model.argument

vantage6.algorithm.store.model.database

vantage6.algorithm.store.model.function

vantage6.algorithm.store.model.vantage6_server

4.10.4 Command line interface

This page contains the API reference of the functions in the vantage package. This package contains the Command-Line Interface (CLI) of the Vantage6 framework.

4.10.4.1 Node CLI

vantage6.cli.node

v6 node

Manage your vantage6 node instances.

```
v6 node [OPTIONS] COMMAND [ARGS]...
```

cli-node-attach

Show the node logs in the current console.

```
v6 node cli-node-attach [OPTIONS]
```

Options

-n, --name <name>
Configuration name

--system
Search for configuration in system folders rather than user folders

--user
Search for configuration in user folders rather than system folders. This is the default

cli-node-clean

Erase temporary Docker volumes.

```
v6 node cli-node-clean [OPTIONS]
```

cli-node-create-private-key

Create and upload a new private key

Use this command with caution! Uploading a new key has several consequences, e.g. you and other users of your organization will no longer be able to read the results of tasks encrypted with current key.

```
v6 node cli-node-create-private-key [OPTIONS]
```

Options

-n, --name <name>
Configuration name

-c, --config <config>
Absolute path to configuration-file; overrides NAME

--system
Search for configuration in system folders rather than user folders

--user
Search for configuration in user folders rather than system folders. This is the default

--no-upload
Don't upload the public key to the server

-o, --organization-name <organization_name>
Organization name. Used in the filename of the private key so that it can easily be recognized again later

--overwrite
Overwrite existing private key if present

cli-node-files

Prints the location of important node files.

If the specified configuration cannot be found, it exits. Otherwise it returns the absolute path to the output.

```
v6 node cli-node-files [OPTIONS]
```

Options

-n, --name <name>
: Configuration name

--system
: Search for the configuration in the system folders

--user
: Search for the configuration in the user folders. This is the default

cli-node-list

Lists all node configurations.

Note that this command cannot find node configuration files in custom directories.

```
v6 node cli-node-list [OPTIONS]
```

cli-node-new-configuration

Create a new node configuration.

Checks if the configuration already exists. If this is not the case a questionnaire is invoked to create a new configuration file.

```
v6 node cli-node-new-configuration [OPTIONS]
```

Options

-n, --name <name>
: Configuration name

--system
: Store this configuration in the system folders

--user
: Store this configuration in the user folders. This is the default

cli-node-remove

Delete a node permanently.

Remove the configuration file, log file, and docker volumes attached to the node.

```
v6 node cli-node-remove [OPTIONS]
```

Options

-n, --name <name>
Configuration name

--system
Search for configuration in system folders rather than user folders

--user
Search for configuration in user folders rather than system folders. This is the default

-f, --force
Don't ask for confirmation

cli-node-set-api-key

Put a new API key into the node configuration file

```
v6 node cli-node-set-api-key [OPTIONS]
```

Options

-n, --name <name>
Configuration name

--api-key <api_key>
New API key

--system
Search for configuration in system folders rather than user folders

--user
Search for configuration in user folders rather than system folders. This is the default

cli-node-start

Start the node.

```
v6 node cli-node-start [OPTIONS]
```

Options

-i, --image <image>
Node Docker image to use

--keep, --auto-remove
Keep node container after finishing. Useful for debugging

--force-db-mount
Always mount node databases; skip the check if they are existing files.

--attach, --detach
Show node logs on the current console after starting the node

--mount-src <mount_src>
Override vantage6 source code in container with the source code in this path

-n, --name <name>
Name of the configuration.

-c, --config <config>
Path to configuration-file; overrides –name

--system
Use system folders instead of user folders. This is the default

--user
Use user folders instead of system folders

cli-node-stop

Stop one or all running nodes.

```
v6 node cli-node-stop [OPTIONS]
```

Options

-n, --name <name>
Configuration name

--system
Search for configuration in system folders instead of user folders

--user
Search for configuration in the user folders instead of system folders. This is the default.

--all
Stop all running nodes

--force
Kill nodes instantly; don't wait for them to shut down

cli-node-version

Returns current version of a vantage6 node.

```
v6 node cli-node-version [OPTIONS]
```

Options

-n, --name <name>
Configuration name

--system
Search for configuration in system folders rather than user folders

--user
Search for configuration in user folders rather than system folders. This is the default

4.10.4.2 Server CLI

v6 server

Manage your vantage6 server instances.

```
v6 server [OPTIONS] COMMAND [ARGS]...
```

cli-server-attach

Show the server logs in the current console.

```
v6 server cli-server-attach [OPTIONS]
```

Options

-n, --name <name>
configuration name

--system

--user

cli-server-configuration-list

Print the available server configurations.

```
v6 server cli-server-configuration-list [OPTIONS]
```

cli-server-files

List files that belong to a particular server instance.

```
v6 server cli-server-files [OPTIONS]
```

Options

-n, --name <name>
Name of the configuration.

-c, --config <config>
Path to configuration-file; overrides –name

--system
Use system folders instead of user folders. This is the default

--user
Use user folders instead of system folders

cli-server-import

Import vantage6 resources into a server instance.

This allows you to create organizations, collaborations, users, tasks, etc from a yaml file.

The **FILE_** argument should be a path to a yaml file containing the vantage6 formatted data to import.

```
v6 server cli-server-import [OPTIONS] FILE
```

Options

--drop-all
Drop all existing data before importing

-i, --image <image>
Node Docker image to use

--mount-src <mount_src>
Override vantage6 source code in container with the source code in this path

--keep, --auto-remove
Keep image after finishing. Useful for debugging

--wait <wait>
: Wait for the import to finish

-n, --name <name>
: Name of the configuration.

-c, --config <config>
: Path to configuration-file; overrides –name

--system
: Use system folders instead of user folders. This is the default

--user
: Use user folders instead of system folders

Arguments

FILE
: Required argument

cli-server-new

Create a new server configuration.

```
v6 server cli-server-new [OPTIONS]
```

Options

-n, --name <name>
: name of the configuration you want to use.

--system

--user

cli-server-remove

Function to remove a server.

Parameters

ctx
: [ServerContext] Server context object

force
: [bool] Whether to ask for confirmation before removing or not

```
v6 server cli-server-remove [OPTIONS]
```

Options

-n, --name <name>
Name of the configuration.

-c, --config <config>
Path to configuration-file; overrides –name

--system
Use system folders instead of user folders. This is the default

--user
Use user folders instead of system folders

-f, --force

cli-server-shell

Run an iPython shell within a running server. This can be used to modify the database.

NOTE: using the shell is no longer recommended as there is no validation on the changes that you make. It is better to use the Python client or a graphical user interface instead.

```
v6 server cli-server-shell [OPTIONS]
```

Options

-n, --name <name>
Name of the configuration.

-c, --config <config>
Path to configuration-file; overrides –name

--system
Use system folders instead of user folders. This is the default

--user
Use user folders instead of system folders

cli-server-start

Start the server.

```
v6 server cli-server-start [OPTIONS]
```

Options

--ip <ip>
IP address to listen on

-p, --port <port>
Port to listen on

-i, --image <image>
Server Docker image to use

--with-ui
Start the graphical User Interface as well

--ui-port <ui_port>
Port to listen on for the User Interface

--with-rabbitmq
Start RabbitMQ message broker as local container - use in development only

--rabbitmq-image <rabbitmq_image>
RabbitMQ docker image to use

--keep, --auto-remove
Keep image after server has stopped. Useful for debugging

--mount-src <mount_src>
Override vantage6 source code in container with the source code in this path

--attach, --detach
Print server logs to the console after start

-n, --name <name>
Name of the configuration.

-c, --config <config>
Path to configuration-file; overrides –name

--system
Use system folders instead of user folders. This is the default

--user
Use user folders instead of system folders

cli-server-stop

Stop one or all running server(s).

```
v6 server cli-server-stop [OPTIONS]
```

Options

-n, --name <name>
Configuration name

--system

--user

--all
Stop all servers

cli-server-version

Print the version of the vantage6 server.

```
v6 server cli-server-version [OPTIONS]
```

Options

-n, --name <name>
Configuration name

--system

--user

4.10.4.3 Algorithm store CLI

v6 algorithm-store

Manage your vantage6 algorithm store server instances.

```
v6 algorithm-store [OPTIONS] COMMAND [ARGS]...
```

cli-algo-store-attach

Show the server logs in the current console.

```
v6 algorithm-store cli-algo-store-attach [OPTIONS]
```

Options

-n, --name <name>
Name of the configuration.

-c, --config <config>
Path to configuration-file; overrides –name

--system
Use system folders instead of user folders. This is the default

--user
Use user folders instead of system folders

cli-algo-store-configuration-list

Print the available server configurations.

```
v6 algorithm-store cli-algo-store-configuration-list [OPTIONS]
```

cli-algo-store-files

List files that belong to a particular server instance.

```
v6 algorithm-store cli-algo-store-files [OPTIONS]
```

Options

-n, --name <name>
Name of the configuration.

-c, --config <config>
Path to configuration-file; overrides –name

--system
Use system folders instead of user folders. This is the default

--user
Use user folders instead of system folders

cli-algo-store-new

Create a new server configuration.

```
v6 algorithm-store cli-algo-store-new [OPTIONS]
```

Options

-n, --name <name>
name of the configuration you want to use.

--system

--user

cli-algo-store-start

Start the algorithm store server.

```
v6 algorithm-store cli-algo-store-start [OPTIONS]
```

Options

--ip <ip>
IP address to listen on

-p, --port <port>
Port to listen on

-i, --image <image>
Algorithm store Docker image to use

--keep, --auto-remove
Keep image after algorithm store has been stopped. Useful for debugging

--mount-src <mount_src>
Override vantage6 source code in container with the source code in this path

--attach, --detach
Print server logs to the console after start

-n, --name <name>
Name of the configuration.

-c, --config <config>
Path to configuration-file; overrides –name

--system
Use system folders instead of user folders. This is the default

--user
Use user folders instead of system folders

cli-algo-store-stop

Stop one or all running server(s).

```
v6 algorithm-store cli-algo-store-stop [OPTIONS]
```

Options

-n, --name <name>

Name of the configuration.

-c, --config <config>

Path to configuration-file; overrides –name

--system

Use system folders instead of user folders. This is the default

--user

Use user folders instead of system folders

--all

Stop all servers

4.10.4.4 Local test setup CLI

v6 dev

Quickly manage a test network with a server and several nodes.

These commands are helpful for local testing of your vantage6 environment.

```
v6 dev [OPTIONS] COMMAND [ARGS]...
```

create-demo-network

Creates a demo network.

Creates server instance as well as its import configuration file. Server name is set to 'dev_default_server'. Generates *n* node configurations, but by default this is set to 3. Then runs a Batch import of organizations/collaborations/users and tasks.

```
v6 dev create-demo-network [OPTIONS]
```

Options

-n, --name <name>

Name for your development setup

--num-nodes <num_nodes>

Generate this number of nodes in the development network

--server-url <server_url>

Server URL to point to. If you are using Docker Desktop, the default http://host.docker.internal should not be changed.

-p, --server-port <server_port>

Port to run the server on. Default is 5000.

-i, --image <image>

Server docker image to use when setting up resources for the development server

--extra-server-config <extra_server_config>

YAML File with additional server configuration. This will be appended to the server configuration file

--extra-node-config <extra_node_config>

YAML File with additional node configuration. This will be appended to each of the node configuration files

remove-demo-network

Remove all related demo network files and folders.

Select a server configuration to remove that server and the nodes attached to it.

```
v6 dev remove-demo-network [OPTIONS]
```

Options

-n, --name <name>

Name of the configuration.

-c, --config <config>

Path to configuration-file; overrides –name

--system

Use system folders instead of user folders. This is the default

--user

Use user folders instead of system folders

start-demo-network

Starts running a demo-network.

Select a server configuration to run its demo network. You should choose a server configuration that you created earlier for a demo network. If you have not created a demo network, you can run *vdev create-demo-network* to create one.

```
v6 dev start-demo-network [OPTIONS]
```

Options

-n, --name <name>
: Name of the configuration.

-c, --config <config>
: Path to configuration-file; overrides –name

--system
: Use system folders instead of user folders. This is the default

--user
: Use user folders instead of system folders

--server-image <server_image>
: Server Docker image to use

--node-image <node_image>
: Node Docker image to use

stop-demo-network

Stops a demo network's server and nodes.

Select a server configuration to stop that server and the nodes attached to it.

```
v6 dev stop-demo-network [OPTIONS]
```

Options

-n, --name <name>
: Name of the configuration.

-c, --config <config>
: Path to configuration-file; overrides –name

--system
: Use system folders instead of user folders. This is the default

--user
: Use user folders instead of system folders

4.10.4.5 Run test algorithms CLI

v6 test

Execute tests on your vantage6 infrastructure.

```
v6 test [OPTIONS] COMMAND [ARGS]...
```

cli-test-features

Run diagnostic checks on an existing vantage6 network.

This command will create a task in the requested collaboration that will test the functionality of vantage6, and will report back the results.

```
v6 test cli-test-features [OPTIONS]
```

Options

--host <host>
: URL of the server

--port <port>
: Port of the server

--api-path <api_path>
: API path of the server

--username <username>
: Username of vantage6 user account to create the task with

--password <password>
: Password of vantage6 user account to create the task with

--collaboration <collaboration>
: ID of the collaboration to create the task in

-o, --organizations <organizations>
: ID(s) of the organization(s) to create the task for

--all-nodes
: Run the diagnostic test on all nodes in the collaboration

--online-only
: Run the diagnostic test on only nodes that are online

--no-vpn
: Don't execute VPN tests

--private-key <private_key>
: Path to the private key for end-to-end encryption

cli-test-integration

Create dev network and run diagnostic checks on it.

This is a full integration test of the vantage6 network. It will create a test server with some nodes using the *vdev* commands, and then run the v6-diagnostics algorithm to test all functionality.

```
v6 test cli-test-integration [OPTIONS]
```

Options

-n, --name <name>

Name for your development setup

--server-url <server_url>

Server URL to point to. If you are using Docker Desktop, the default http://host.docker.internal should not be changed.

-i, --image <image>

Server Docker image to use

--keep <keep>

Keep the dev network after finishing the test

--extra-server-config <extra_server_config>

YAML File with additional server configuration. This will be appended to the server configuration file

--extra-node-config <extra_node_config>

YAML File with additional node configuration. This will be appended to each of the node configuration files

4.10.4.6 vantage6.cli.context

The context module in the CLI package contains the Context classes of instances started from the CLI, such as nodes and servers. These contexts are related to the host system and therefore part of the CLI package.

All classes are derived from the abstract AppContext class and provide the vantage6 applications with naming conventions, standard file locations, and more.

get_context(*type_*, *name*, *system_folders*)

Load the server context from the configuration file.

Parameters

- **type** (*InstanceType*) – The type of instance to get the context for
- **name** (*str*) – Name of the instance
- **system_folders** (*bool*) – Wether to use system folders or if False, the user folders

Returns

Specialized subclass context of AppContext for the given instance type

Return type

AppContext

`select_context_class`(*type_*)

Select the context class based on the type of instance.

Parameters
type (*InstanceType*) – The type of instance for which the context should be inserted

Returns
Specialized subclass of AppContext for the given instance type

Return type
ServerContext | *NodeContext* | *AlgorithmStoreContext*

Raises
NotImplementedError – If the type_ is not implemented

`class NodeContext`(*instance_name*, *system_folders=False*, *config_file=None*, *print_log_header=True*)

Bases: *AppContext*

Node context object for the host system.

See DockerNodeContext for the node instance mounts when running as a dockerized service.

Parameters

- **instance_name** (*str*) – Name of the configuration instance, corresponds to the filename of the configuration file.
- **system_folders** (*bool, optional*) – _description_, by default N_FOL
- **config_file** (*str, optional*) – _description_, by default None

`INST_CONFIG_MANAGER`

alias of *NodeConfigurationManager*

`classmethod available_configurations`(*system_folders=False*)

Find all available server configurations in the default folders.

Parameters
system_folders (*bool, optional*) – System wide or user configuration, by default N_FOL

Returns
The first list contains validated configuration files, the second list contains invalid configuration files.

Return type
tuple[list, list]

`classmethod config_exists`(*instance_name*, *system_folders=False*)

Check if a configuration file exists.

Parameters

- **instance_name** (*str*) – Name of the configuration instance, corresponds to the filename of the configuration file.
- **system_folders** (*bool, optional*) – System wide or user configuration, by default N_FOL

Returns
Whether the configuration file exists or not

Return type
bool

property databases: dict

Dictionary of local databases that are available for this node.

Returns
dictionary with database names as keys and their corresponding paths as values.

Return type
dict

property docker_container_name: str

Docker container name of the node.

Returns
Node's Docker container name

Return type
str

property docker_network_name: str

Private Docker network name which is unique for this node.

Returns
Docker network name

Return type
str

property docker_squid_volume_name: str

Docker volume in which the SSH configuration is stored.

Returns
Docker volume name

Return type
str

property docker_ssh_volume_name: str

Docker volume in which the SSH configuration is stored.

Returns
Docker volume name

Return type
str

docker_temporary_volume_name(*job_id*)

Docker volume in which temporary data is stored. Temporary data is linked to a specific run. Multiple algorithm containers can have the same run id, and therefore the share same temporary volume.

Parameters
job_id (*int*) – run id provided by the server

Returns
Docker volume name

Return type
str

property docker_volume_name: str

Docker volume in which task data is stored. In case a file based database is used, this volume contains the database file as well.

Returns
Docker volume name

Return type
str

property docker_vpn_volume_name: str

Docker volume in which the VPN configuration is stored.

Returns
Docker volume name

Return type
str

classmethod from_external_config_file(*path*, *system_folders=False*)

Create a node context from an external configuration file. External means that the configuration file is not located in the default folders but its location is specified by the user.

Parameters

- **path** (*str*) – Path of the configuration file
- **system_folders** (*bool, optional*) – System wide or user configuration, by default N_FOL

Returns
Node context object

Return type
NodeContext

get_database_uri(*label='default'*)

Obtain the database URI for a specific database.

Parameters
label (*str, optional*) – Database label, by default "default"

Returns
URI to the database

Return type
str

static type_data_folder(*system_folders=False*)

Obtain OS specific data folder where to store node specific data.

Parameters
system_folders (*bool, optional*) – System wide or user configuration

Returns
Path to the data folder

Return type
Path

class ServerContext(*instance_name*, *system_folders=True*)

Bases: *BaseServerContext*

Server context

Parameters

- **instance_name** (*str*) – Name of the configuration instance, corresponds to the filename of the configuration file.
- **system_folders** (*bool, optional*) – System wide or user configuration, by default S_FOL

INST_CONFIG_MANAGER

alias of *ServerConfigurationManager*

classmethod available_configurations(*system_folders=True*)

Find all available server configurations in the default folders.

Parameters
system_folders (*bool, optional*) – System wide or user configuration, by default S_FOL

Returns
The first list contains validated configuration files, the second list contains invalid configuration files.

Return type
tuple[list, list]

classmethod config_exists(*instance_name*, *system_folders=True*)

Check if a configuration file exists.

Parameters

- **instance_name** (*str*) – Name of the configuration instance, corresponds to the filename of the configuration file.
- **system_folders** (*bool, optional*) – System wide or user configuration, by default S_FOL

Returns
Whether the configuration file exists or not

Return type
bool

property docker_container_name: str

Name of the docker container that the server is running in.

Returns
Server's docker container name

Return type
str

classmethod from_external_config_file(*path*, *system_folders=True*)

Create a server context from an external configuration file. External means that the configuration file is not located in the default folders but its location is specified by the user.

Parameters

- **path** (*str*) – Path of the configuration file
- **system_folders** (*bool, optional*) – System wide or user configuration, by default S_FOL

Returns
Server context object

Return type
ServerContext

`get_database_uri`()

Obtain the database uri from the environment or the configuration. The *VANTAGE6_DB_URI* environment variable is used by the Docker container, but can also be set by the user.

Returns
string representation of the database uri

Return type
str

`class AlgorithmStoreContext`(*instance_name*, *system_folders=True*)

Bases: *BaseServerContext*

A context class for the algorithm store server.

Parameters

- **instance_name** (*str*) – Name of the configuration instance, corresponds to the filename of the configuration file.
- **system_folders** (*bool, optional*) – System wide or user configuration, by default S_FOL

`INST_CONFIG_MANAGER`

alias of *ServerConfigurationManager*

`classmethod available_configurations`(*system_folders=True*)

Find all available server configurations in the default folders.

Parameters
system_folders (*bool, optional*) – System wide or user configuration, by default S_FOL

Returns
The first list contains validated configuration files, the second list contains invalid configuration files.

Return type
tuple[list, list]

`classmethod config_exists`(*instance_name*, *system_folders=True*)

Check if a configuration file exists.

Parameters

- **instance_name** (*str*) – Name of the configuration instance, corresponds to the filename of the configuration file.
- **system_folders** (*bool, optional*) – System wide or user configuration, by default S_FOL

Returns
Whether the configuration file exists or not

Return type
bool

`property docker_container_name: str`

Name of the docker container that the server is running in.

Returns
Server's docker container name

Return type
str

classmethod from_external_config_file(*path*, *system_folders=True*)

Create a server context from an external configuration file. External means that the configuration file is not located in the default folders but its location is specified by the user.

Parameters

- **path** (*str*) – Path of the configuration file
- **system_folders** (*bool, optional*) – System wide or user configuration, by default S_FOL

Returns
Server context object

Return type
AlgorithmStoreContext

get_database_uri()

Obtain the database uri from the environment or the configuration. The *VANTAGE6_DB_URI* environment variable is used by the Docker container, but can also be set by the user.

Returns
string representation of the database uri

Return type
str

class BaseServerContext(*instance_type*, *instance_name*, *system_folders=False*, *config_file=None*, *print_log_header=True*)

Bases: *AppContext*

Base context for a vantage6 server or algorithm store server

Contains functions that the ServerContext and AlgorithmStoreContext have in common.

classmethod from_external_config_file(*path*, *server_type*, *config_name_env_var*, *system_folders=True*)

Create a server context from an external configuration file. External means that the configuration file is not located in the default folders but its location is specified by the user.

Parameters

- **path** (*str*) – Path of the configuration file
- **server_type** (*ServerType*) – Type of server, either 'server' or 'algorithm-store'
- **config_name_env_var** (*str*) – Name of the environment variable that contains the name of the configuration
- **system_folders** (*bool, optional*) – System wide or user configuration, by default S_FOL

Returns
Server context object

Return type
ServerContext

get_database_uri(*db_env_var*)

Obtain the database uri from the environment or the configuration.

Parameters
db_env_var (*str*) – Name of the environment variable that contains the database uri

Returns
string representation of the database uri

Return type
str

4.10.4.7 vanatge6.cli.configuration_manager

class NodeConfiguration(**args*, ***kwargs*)

Bases: *Configuration*

Stores the node's configuration and defines a set of node-specific validators.

class NodeConfigurationManager(*name*, **args*, ***kwargs*)

Bases: *ConfigurationManager*

Maintains the node's configuration.

Parameters
name (*str*) – Name of the configuration file.

classmethod from_file(*path*)

Create a new instance of the NodeConfigurationManager from a configuration file.

Parameters
path (*str*) – Path to the configuration file.

Returns
A new instance of the NodeConfigurationManager.

Return type
NodeConfigurationManager

class ServerConfiguration(**args*, ***kwargs*)

Bases: *Configuration*

Stores the server's configuration and defines a set of server-specific validators.

class ServerConfigurationManager(*name*, **args*, ***kwargs*)

Bases: *ConfigurationManager*

Maintains the server's configuration.

Parameters
name (*str*) – Name of the configuration file.

classmethod from_file(*path*)

Create a new instance of the ServerConfigurationManager from a configuration file.

Parameters
path (*str*) – Path to the configuration file.

Returns
A new instance of the ServerConfigurationManager.

Return type
ServerConfigurationManager

class TestConfiguration(**args*, ***kwargs*)

Bases: *Configuration*

class TestingConfigurationManager(*name*, **args*, ***kwargs*)

Bases: *ConfigurationManager*

classmethod from_file(*path*)

Load a configuration from a file.

Parameters

- **path** (*Path | str*) – The path to the file to load the configuration from.
- **conf_class** (*Type[Configuration]*) – The class to use for the configuration.

Returns
The configuration manager with the configuration.

Return type
ConfigurationManager

Raises
AssertionError – If the name of the configuration could not be extracted from the file path.

4.10.4.8 vantage6.cli.configuration_wizard

algo_store_configuration_questionaire(*instance_name*)

Questionary to generate a config file for the algorithm store server instance.

Parameters
instance_name (*str*) – Name of the server instance.

Returns
Dictionary with the new server configuration

Return type
dict

configuration_wizard(*type_*, *instance_name*, *system_folders*)

Create a configuration file for a node or server instance.

Parameters

- **type** (*InstanceType*) – Type of the instance to create a configuration for
- **instance_name** (*str*) – Name of the instance
- **system_folders** (*bool*) – Whether to use the system folders or not

Returns
Path to the configuration file

Return type
Path

node_configuration_questionaire(*dirs*, *instance_name*)

Questionary to generate a config file for the node instance.

Parameters

- **dirs** (*dict*) – Dictionary with the directories of the node instance.
- **instance_name** (*str*) – Name of the node instance.

Returns
Dictionary with the new node configuration

Return type
dict

`select_configuration_questionaire`(*type_*, *system_folders*)

Ask which configuration the user wants to use. It shows only configurations that are in the default folder.

Parameters

- **type** (*InstanceType*) – Type of the instance to create a configuration for
- **system_folders** (*bool*) – Whether to use the system folders or not

Returns
Name of the configuration

Return type
str

`server_configuration_questionaire`(*instance_name*)

Questionary to generate a config file for the server instance.

Parameters
instance_name (*str*) – Name of the server instance.

Returns
Dictionary with the new server configuration

Return type
dict

4.10.4.9 vanatge6.cli.rabbitmq.queue_manager

`class RabbitMQManager`(*ctx*, *network_mgr*, *image=None*)

Manages the RabbitMQ docker container

Parameters

- **ctx** (ServerContext) – Configuration object
- **network_mgr** (NetworkManager) – Network manager for network in which server container resides
- **image** (*str*) – Docker image to use for RabbitMQ container. By default, the image harbor2.vantage6.ai/infrastructure/rabbitmq is used.

`is_running`()

Returns
Whether the container has fully initialized RabbitMQ or not

Return type
bool

`start()`

Start a docker container which runs a RabbitMQ queue

Return type
None

4.10.4.10 vanatge6.cli.rabbitmq

RabbitMQ utilities.

`split_rabbitmq_uri(rabbit_uri)`

Get details (user, pass, host, vhost, port) from a RabbitMQ uri.

Parameters
rabbit_uri (*str*) – URI of RabbitMQ service ('amqp://$user:$pass@$host:$port/$vhost')

Returns
The vhost defined in the RabbitMQ URI

Return type
dict[str]

4.10.4.11 vantage6.cli.utils

Utility functions for the CLI

`check_config_name_allowed(name)`

Check if configuration name is allowed

Parameters
name (*str*) – Name to be checked

Return type
None

`check_if_docker_daemon_is_running(docker_client)`

Check if Docker daemon is running

Parameters
docker_client (*docker.DockerClient*) – The docker client

Return type
None

`prompt_config_name(name)`

Get a new configuration name from the user, or simply return the name if it is not None.

Parameters
name (*str*) – Name to be checked

Returns
The name of the configuration

Return type
str

`remove_file(file, file_type)`

Remove a file if it exists.

Parameters

- **file** (*str*) – absolute path to the file to be deleted
- **file_type** (*str*) – type of file, used for logging

Return type
None

4.10.5 Python client

This page contains the API reference of the functions in the vantage-client package.

4.10.5.1 User Client

vantage6.client

Client
alias of *UserClient*

class UserClient(**args*, *log_level='debug'*, ***kwargs*)
Bases: ClientBase

User interface to the vantage6-server

class Collaboration(*parent*)
Bases: SubClient

Collection of collaboration requests

add_organization(*organization*, *collaboration=None*)
Add an organization to a collaboration
Parameters
- **organization** (*int*) – Id of the organization you want to add to the collaboration
- **collaboration** (*int, optional*) – Id of the collaboration you want to add the organization to. If no id is provided the value of setup_collaboration() is used.

Returns
Containing the updated list of organizations in the collaboration
Return type
list[dict]

create(*name*, *organizations*, *encrypted=False*)
Create new collaboration
Parameters
- **name** (*str*) – Name of the collaboration
- **organizations** (*list*) – List of organization ids which participate in the collaboration
- **encrypted** (*bool, optional*) – Whenever the collaboration should be encrypted or not, by default False

Returns
Containing the new collaboration meta-data
Return type
dict

delete(*id_=None*, *delete_dependents=None*)
Deletes a collaboration
Parameters
- **id** (*int*) – Id of the collaboration you want to delete

- **delete_dependents** (*bool, optional*) – Delete the tasks, nodes and studies that are part of the collaboration as well. If this is False, and dependents exist, the server will refuse to delete the collaboration. Default is False.

Returns
Message from the server
Return type
dict

get(*id_*)

View specific collaboration

Parameters
id (*int*) – Id from the collaboration you want to view
Returns
Containing the collaboration information
Return type
dict

list(*scope='organization'*, *name=None*, *encrypted=None*, *organization=None*, *page=1*, *per_page=20*)

View your collaborations

Parameters
- **scope** (*str, optional*) – Scope of the list, accepted values are *organization* and *global*. In case of *organization* you get the collaborations in which your organization participates. If you specify global you get the collaborations which you are allowed to see.
- **name** (*str, optional (with LIKE operator)*) – Filter collaborations by name
- **organization** (*int, optional*) – Filter collaborations by organization id
- **encrypted** (*bool, optional*) – Filter collaborations by whether or not they are encrypted
- **page** (*int, optional*) – Pagination page, by default 1
- **per_page** (*int, optional*) – Number of items on a single page, by default 20

Returns
Containing collabotation information
Return type
list[dict]

Notes

- Pagination does not work in combination with scope *organization* as pagination is missing at endpoint /organization/<id>/collaboration

remove_organization(*organization*, *collaboration=None*)

Remove an organization from a collaboration

Parameters
- **organization** (*int*) – Id of the organization you want to remove from the collaboration
- **collaboration** (*int, optional*) – Id of the collaboration you want to remove the organization from. If no id is provided the value of setup_collaboration() is used.

Returns
Containing the updated list of organizations in the collaboration
Return type
list[dict]

update(*id_=None*, *name=None*, *encrypted=None*, *organizations=None*)

Update collaboration information

Parameters

- **id** (*int*) – Id of the collaboration you want to update. If no id is provided the value of setup_collaboration() is used.
- **name** (*str, optional*) – New name of the collaboration
- **encrypted** (*bool, optional*) – New encryption status of the collaboration
- **organizations** (*list[int], optional*) – New list of organization ids which participate in the collaboration

Returns
Containing the updated collaboration information

Return type
dict

class Node(*parent*)

Bases: SubClient

Collection of node requests

create(*collaboration=None*, *organization=None*, *name=None*)

Register new node

Parameters
- **collaboration** (*int*) – Collaboration id to which this node belongs. If no ID was provided the, collaboration from *client.setup_collaboration()* is used.
- **organization** (*int, optional*) – Organization id to which this node belongs. If no id provided the users organization is used. Default value is None
- **name** (*str, optional*) – Name of the node. If no name is provided the server will generate one. Default value is None

Returns
Containing the meta-data of the new node

Return type
dict

delete(*id_*)

Deletes a node

Parameters
id (*int*) – Id of the node you want to delete

Returns
Message from the server

Return type
dict

get(*id_*)

View specific node

Parameters
id (*int*) – Id of the node you want to inspect

Returns
Containing the node meta-data

Return type
dict

kill_tasks(*id_*)

Kill all tasks currently running on a node

Parameters
id (*int*) – Id of the node of which you want to kill the tasks

Returns
Message from the server

Return type
dict

list(*name=None*, *organization=None*, *collaboration=None*, *study=None*, *is_online=None*, *ip=None*, *last_seen_from=None*, *last_seen_till=None*, *page=1*, *per_page=20*)

List nodes

Parameters

- **name** (*str, optional*) – Filter by name (with LIKE operator)
- **organization** (*int, optional*) – Filter by organization id
- **collaboration** (*int, optional*) – Filter by collaboration id. If no id is provided but collaboration was defined earlier by user, filtering on that collaboration
- **study** (*int, optional*) – Filter by study id
- **is_online** (*bool, optional*) – Filter on whether nodes are online or not
- **ip** (*str, optional*) – Filter by node VPN IP address
- **last_seen_from** (*str, optional*) – Filter if node has been online since date (format: yyyy-mm-dd)
- **last_seen_till** (*str, optional*) – Filter if node has been online until date (format: yyyy-mm-dd)
- **page** (*int, optional*) – Pagination page, by default 1
- **per_page** (*int, optional*) – Number of items on a single page, by default 20

Return type

list[dict]

Returns

-
- *list of dicts* – Containing meta-data of the nodes

update(*id_*, *name=None*, *clear_ip=None*)

Update node information

Parameters

- **id** (*int*) – Id of the node you want to update
- **name** (*str, optional*) – New node name, by default None
- **clear_ip** (*bool, optional*) – Clear the VPN IP address of the node, by default None

Returns

Containing the meta-data of the updated node

Return type

dict

class Organization(*parent*)

Bases: SubClient

Collection of organization requests

create(*name*, *address1*, *address2*, *zipcode*, *country*, *domain*, *public_key=None*)

Create new organization

Parameters

- **name** (*str*) – Name of the organization
- **address1** (*str*) – Street and number
- **address2** (*str*) – City
- **zipcode** (*str*) – Zip or postal code
- **country** (*str*) – Country
- **domain** (*str*) – Domain of the organization (e.g. vantage6.ai)
- **public_key** (*str, optional*) – Public key of the organization. This can be set later, by default None

Returns

Containing the information of the new organization

Return type

dict

get(*id_=None*)

View specific organization

Parameters

id (*int, optional*) – Organization *id* of the organization you want to view. In case no *id* is provided it will display your own organization, default value is None.

Returns

Containing the organization meta-data

Return type

dict

list(*name=None*, *country=None*, *collaboration=None*, *study=None*, *page=None*, *per_page=None*)

List organizations

Parameters

- **name** (*str, optional*) – Filter by name (with LIKE operator)
- **country** (*str, optional*) – Filter by country
- **collaboration** (*int, optional*) – Filter by collaboration id. If client.setup_collaboration() was called, the previously setup collaboration is used. Default value is None
- **study** (*int, optional*) – Filter by study id
- **page** (*int, optional*) – Pagination page, by default 1
- **per_page** (*int, optional*) – Number of items on a single page, by default 20

Returns

Containing meta-data information of the organizations

Return type

list[dict]

update(*id_=None*, *name=None*, *address1=None*, *address2=None*, *zipcode=None*, *country=None*, *domain=None*, *public_key=None*)

Update organization information

Parameters

- **id** (*int, optional*) – Organization id, by default None
- **name** (*str, optional*) – New organization name, by default None
- **address1** (*str, optional*) – Address line 1, by default None
- **address2** (*str, optional*) – Address line 2, by default None
- **zipcode** (*str, optional*) – Zipcode, by default None
- **country** (*str, optional*) – Country, by default None
- **domain** (*str, optional*) – Domain of the organization (e.g. *iknl.nl*), by default None
- **public_key** (*str, optional*) – public key, by default None

Returns

The meta-data of the updated organization

Return type

dict

class Result(*parent*)

Bases: SubClient

Client to get the results of one or multiple algorithm runs

from_task(*task_id*)

Get all results from a specific task

Parameters

task_id (*int*) – Id of the task to get results from

Returns

Containing the results

Return type
list[dict]

`get`(*id_*)

View a specific result

Parameters
id (*int*) – id of the run you want to inspect

Returns
Containing the run data

Return type
dict

`class Role`(*parent*)

Bases: `SubClient`

`create`(*name*, *description*, *rules*, *organization=None*)

Register new role

Parameters
- **name** (*str*) – Role name
- **description** (*str*) – Human readable description of the role
- **rules** (*list*) – Rules that this role contains
- **organization** (*int, optional*) – Organization to which this role belongs. In case this is not provided the users organization is used. By default None

Returns
Containing meta-data of the new role

Return type
dict

`delete`(*role*)

Delete role

Parameters
role (*int*) – CAUTION! Id of the role to be deleted. If you remove roles that are attached to you, you might lose access!

Returns
Message from the server

Return type
dict

`get`(*id_*)

View specific role

Parameters
id (*int*) – Id of the role you want to insepct

Returns
Containing meta-data of the role

Return type
dict

`list`(*name=None*, *description=None*, *organization=None*, *rule=None*, *user=None*, *include_root=None*, *page=1*, *per_page=20*)

List of roles

Parameters
- **name** (*str, optional*) – Filter by name (with LIKE operator)
- **description** (*str, optional*) – Filter by description (with LIKE operator)
- **organization** (*int, optional*) – Filter by organization id
- **rule** (*int, optional*) – Only show roles that contain this rule id
- **user** (*int, optional*) – Only show roles that belong to a particular user id

- **include_root** (*bool, optional*) – Include roles that are not assigned to any particular organization
- **page** (*int, optional*) – Pagination page, by default 1
- **per_page** (*int, optional*) – Number of items on a single page, by default 20

Returns
Containing roles meta-data

Return type
list[dict]

update(*role*, *name=None*, *description=None*, *rules=None*)

Update role

Parameters
- **role** (*int*) – Id of the role that updated
- **name** (*str, optional*) – New name of the role, by default None
- **description** (*str, optional*) – New description of the role, by default None
- **rules** (*list, optional*) – CAUTION! This will not *add* rules but replace them. If you remove rules from your own role you lose access. By default None

Returns
Containing the updated role data

Return type
dict

class Rule(*parent*)

Bases: SubClient

get(*id_*)

View specific rule

Parameters
id (*int*) – Id of the rule you want to view

Returns
Containing the information about this rule

Return type
dict

list(*name=None*, *operation=None*, *scope=None*, *role=None*, *page=1*, *per_page=20*)

List of all available rules

Parameters
- **name** (*str, optional*) – Filter by rule name
- **operation** (*str, optional*) – Filter by operation
- **scope** (*str, optional*) – Filter by scope
- **role** (*int, optional*) – Only show rules that belong to this role id
- **page** (*int, optional*) – Pagination page, by default 1
- **per_page** (*int, optional*) – Number of items on a single page, by default 20

Returns
Containing all the rules from the vantage6 server

Return type
list of dicts

class Run(*parent*)

Bases: SubClient

from_task(*task_id*, *include_task=False*)

Get all algorithm runs from a specific task

Parameters
- **task_id** (*int*) – Id of the task to get results from

- **include_task** (*bool, optional*) – Whenever to include the task or not, by default False

Returns
Containing the results

Return type
list[dict]

get(*id_*, *include_task=False*)

View a specific run

Parameters
- **id** (*int*) – id of the run you want to inspect
- **include_task** (*bool, optional*) – Whenever to include the task or not, by default False

Returns
Containing the run data

Return type
dict

list(*task=None*, *organization=None*, *state=None*, *node=None*, *include_task=False*, *started=None*, *assigned=None*, *finished=None*, *port=None*, *page=None*, *per_page=None*)

List runs

Parameters
- **task** (*int, optional*) – Filter by task id
- **organization** (*int, optional*) – Filter by organization id
- **state** (*int, optional*) – Filter by state: (‘open’,)
- **node** (*int, optional*) – Filter by node id
- **include_task** (*bool, optional*) – Whenever to include the task or not, by default False
- **started** (*tuple[str, str], optional*) – Filter on a range of start times (format: yyyy-mm-dd)
- **assigned** (*tuple[str, str], optional*) – Filter on a range of assign times (format: yyyy-mm-dd)
- **finished** (*tuple[str, str], optional*) – Filter on a range of finished times (format: yyyy-mm-dd)
- **port** (*int, optional*) – Port on which run was computed
- **page** (*int, optional*) – Pagination page number, defaults to 1
- **per_page** (*int, optional*) – Number of items per page, defaults to 20

Returns
A dictionary containing the key ‘data’ which contains a list of runs, and a key ‘links’ which contains the pagination metadata.

Return type
dict | list[dict]

class Task(*parent*)

Bases: SubClient

create(*organizations*, *name*, *image*, *description*, *input_*, *collaboration=None*, *study=None*, *databases=None*)

Create a new task

Parameters
- **organizations** (*list*) – Organization ids (within the collaboration) which need to execute this task
- **name** (*str*) – Human readable name
- **image** (*str*) – Docker image name which contains the algorithm
- **description** (*str*) – Human readable description

- **input** (*dict*) – Algorithm input
- **collaboration** (*int, optional*) – Id of the collaboration to which this task belongs. Should be set if the study is not set
- **study** (*int, optional*) – Id of the study to which this task belongs. Should be set if the collaboration is not set
- **databases** (*list[dict], optional*) – Databases to be used at the node. Each dict should contain at least a 'label' key. Additional keys are 'query' (if using SQL/SPARQL databases), 'sheet_name' (if using Excel databases), and 'preprocessing' information.

Returns
A dictionairy containing data on the created task, or a message from the server if the task could not be created

Return type
dict

delete(*id_*)

Delete a task

Also removes the related runs.

Parameters
id (*int*) – Id of the task to be removed

Returns
Message from the server

Return type
dict

get(*id_*, *include_results=False*)

View specific task

Parameters
- **id** (*int*) – Id of the task you want to view
- **include_results** (*bool, optional*) – Whenever to include the results or not, by default False

Returns
Containing the task data

Return type
dict

kill(*id_*)

Kill a task running on one or more nodes

Note that this does not remove the task from the database, but merely halts its execution (and prevents it from being restarted).

Parameters
id (*int*) – Id of the task to be killed

Returns
Message from the server

Return type
dict

list(*initiating_org=None*, *initiating_user=None*, *collaboration=None*, *study=None*, *image=None*, *parent=None*, *job=None*, *name=None*, *include_results=False*, *description=None*, *database=None*, *run=None*, *status=None*, *user_created=None*, *page=1*, *per_page=20*)

List tasks

Parameters
- **name** (*str, optional*) – Filter by the name of the task. It will match with a Like operator. I.e. E% will search for task names that start with an 'E'.
- **initiating_org** (*int, optional*) – Filter by initiating organization

- **initiating_user** (*int, optional*) – Filter by initiating user
- **collaboration** (*int, optional*) – Filter by collaboration. If no id is provided but collaboration was defined earlier by setup_collaboration(), filtering on that collaboration
- **study** (*int, optional*) – Filter by study
- **image** (*str, optional*) – Filter by Docker image name (with LIKE operator)
- **parent** (*int, optional*) – Filter by parent task
- **job** (*int, optional*) – Filter by job id
- **include_results** (*bool, optional*) – Whenever to include the results in the tasks, by default False
- **description** (*str, optional*) – Filter by description (with LIKE operator)
- **database** (*str, optional*) – Filter by database (with LIKE operator)
- **run** (*int, optional*) – Only show task that contains this run id
- **status** (*str, optional*) – Filter by task status (e.g. 'active', 'pending', 'completed', 'crashed')
- **user_created** (*bool, optional*) – If True, show only top-level tasks created by users. If False, show only subtasks created by algorithm containers.
- **page** (*int, optional*) – Pagination page, by default 1
- **per_page** (*int, optional*) – Number of items on a single page, by default 20

Returns
dictonairy containing the key 'data' which contains the tasks and a key 'links' containing the pagination metadata

Return type
dict

class User(*parent*)

Bases: SubClient

create(*username*, *firstname*, *lastname*, *password*, *email*, *organization=None*, *roles=[]*, *rules=[]*)

Create new user

Parameters
- **username** (*str*) – Used to login to the service. This can not be changed later.
- **firstname** (*str*) – Firstname of the new user
- **lastname** (*str*) – Lastname of the new user
- **password** (*str*) – Password of the new user
- **email** (*str*) – Email address of the new user
- **organization** (*int*) – Organization *id* this user should belong to
- **roles** (*list of ints*) – Role ids that are assigned to this user. Note that you can only assign roles if you own the rules within this role.
- **rules** (*list of ints*) – Rule ids that are assigned to this user. Note that you can only assign rules that you own

Returns
Containing data of the new user

Return type
dict

get(*id_=None*)

View user information

Parameters
id (*int, optional*) – User *id*, by default None. When no *id* is provided your own user information is displayed

Returns
Containing user information

Return type
dict

list(*username=None*, *organization=None*, *firstname=None*, *lastname=None*, *email=None*, *role=None*, *rule=None*, *last_seen_from=None*, *last_seen_till=None*, *page=1*, *per_page=20*)

List users

Parameters

- **username** (*str, optional*) – Filter by username (with LIKE operator)
- **organization** (*int, optional*) – Filter by organization id
- **firstname** (*str, optional*) – Filter by firstname (with LIKE operator)
- **lastname** (*str, optional*) – Filter by lastname (with LIKE operator)
- **email** (*str, optional*) – Filter by email (with LIKE operator)
- **role** (*int, optional*) – Show only users that have this role id
- **rule** (*int, optional*) – Show only users that have this rule id
- **last_seen_from** (*str, optional*) – Filter users that have logged on since (format yyyy-mm-dd)
- **last_seen_till** (*str, optional*) – Filter users that have logged on until (format yyyy-mm-dd)
- **page** (*int, optional*) – Pagination page, by default 1
- **per_page** (*int, optional*) – Number of items on a single page, by default 20

Returns

Containing the meta-data of the users

Return type

list of dicts

update(*id_=None*, *firstname=None*, *lastname=None*, *organization=None*, *rules=None*, *roles=None*, *email=None*)

Update user details

In case you do not supply a user_id, your user is being updated.

Parameters

- **id** (*int*) – User *id* from the user you want to update
- **firstname** (*str*) – Your first name
- **lastname** (*str*) – Your last name
- **organization** (*int*) – Organization id of the organization you want to be part of. This can only done by super-users.
- **rules** (*list of ints*) – USE WITH CAUTION! Rule ids that should be assigned to this user. All previous assigned rules will be removed!
- **roles** (*list of ints*) – USE WITH CAUTION! Role ids that should be assigned to this user. All previous assigned roles will be removed!
- **email** (*str*) – New email from the user

Returns

A dict containing the updated user data

Return type

dict

class Util(*parent*)

Bases: SubClient

Collection of general utilities

change_my_password(*current_password*, *new_password*)

Change your own password by providing your current password

Parameters

- **current_password** (*str*) – Your current password
- **new_password** (*str*) – Your new password

Returns

Message from the server

Return type
dict

`generate_private_key`(*file_=None*)

Generate new private key

Parameters
file (*str, optional*) – Path where to store the private key, by default None

Return type
None

`get_server_health`()

View the health of the vantage6-server

Returns
Containing the server health information

Return type
dict

`get_server_version`(*attempts_on_timeout=None*)

View the version number of the vantage6-server :type attempts_on_timeout: Optional[int] :param attempts_on_timeout: Number of attempts to make when the server is not responding.
Default is unlimited.

Returns
A dict containing the version number

Return type
dict

`reset_my_password`(*email=None*, *username=None*)

Start reset password procedure

Either a username of email needs to be provided.

Parameters
- **email** (*str, optional*) – Email address of your account, by default None
- **username** (*str, optional*) – Username of your account, by default None

Returns
Message from the server

Return type
dict

`reset_two_factor_auth`(*password*, *email=None*, *username=None*)

Start reset procedure for two-factor authentication

The password and either username of email must be provided.

Parameters
- **password** (*str*) – Password of your account
- **email** (*str, optional*) – Email address of your account, by default None
- **username** (*str, optional*) – Username of your account, by default None

Returns
Message from the server

Return type
dict

`set_my_password`(*token*, *password*)

Set a new password using a recovery token

Token can be obtained through *.reset_password(...)*

Parameters
- **token** (*str*) – Token obtained from *reset_password*

- **password** (*str*) – New password

Returns
Message from the server

Return type
dict

`set_two_factor_auth`(*token*)

Setup two-factor authentication using a recovery token after you have lost access.

Token can be obtained through *.reset_two_factor_auth(...)*

Parameters
token (*str*) – Token obtained from *reset_two_factor_auth*

Returns
Message from the server

Return type
dict

`authenticate`(*username*, *password*, *mfa_code=None*)

Authenticate as a user

It also collects some additional info about your user.

Parameters

- **username** (*str*) – Username used to authenticate
- **password** (*str*) – Password used to authenticate
- **mfa_code** (*str | int*) – Six-digit two-factor authentication code

Return type
None

`setup_collaboration`(*collaboration_id*)

Setup the collaboration.

This gets the collaboration from the server and stores its details in the client and sets the algorithm stores available for this collaboration. When this has been called, other functions no longer require the *collaboration_id* to be provided.

Parameters
collaboration_id (*int*) – Id of the collaboration

Return type
None

`wait_for_results`(*task_id*, *interval=1*)

Polls the server to check when results are ready, and returns the results when the task is completed.

Parameters

- **task_id** (*int*) – ID of the task that you are waiting for
- **interval** (*float*) – Interval in seconds between checks if task is finished. Default 1.

Returns
A dictionary with the results of the task, after it has completed.

Return type
dict

class StudySubClient(*parent*)

Bases: SubClient

Subclient for the algorithm store.

add_organization(*organization*, *study=None*)

Add an organization to a study

Parameters

- **organization** (*int*) – Id of the organization you want to add to the study
- **study** (*int, optional*) – Id of the study you want to add the organization to.

Returns
Containing the updated list of organizations in the study

Return type
list[dict]

create(*name*, *organizations*, *collaboration=None*)

Create new study

Parameters

- **name** (*str*) – Name of the study
- **organizations** (*list[int]*) – List of organization ids which participate in the study
- **collaboration** (*int | None*) – Id of the collaboration the study is part of. If None, the value of setup_collaboration() is used.

Returns
Containing the new study information

Return type
dict

delete(*id_=None*)

Deletes a study

Parameters
id (*int*) – Id of the study you want to delete

Returns
Message from the server

Return type
dict

get(*id_*)

Get a study by its id.

Parameters
id (*int*) – The id of the study

Returns
The study

Return type
dict

list(*name=None*, *organization=None*, *include_organizations=False*, *page=1*, *per_page=20*)

View your studies

Parameters

- **name** (*str, optional (with LIKE operator)*) – Filter studies by name
- **organization** (*int, optional*) – Filter studies by organization id
- **include_organizations** (*bool, optional*) – Include organizations in the response, by default False
- **page** (*int, optional*) – Pagination page, by default 1
- **per_page** (*int, optional*) – Number of items on a single page, by default 20

Returns
Containing collabotation information

Return type
list[dict]

remove_organization(*organization*, *study=None*)

Remove an organization from a study

Parameters

- **organization** (*int*) – Id of the organization you want to remove from the study
- **study** (*int, optional*) – Id of the study you want to remove the organization from

Returns
Containing the updated list of organizations in the study

Return type
list[dict]

update(*id_*, *name=None*, *organizations=None*)

Update study information

Parameters

- **id** (*int*) – Id of the study you want to update.
- **name** (*str, optional*) – New name of the study
- **organizations** (*list[int], optional*) – New list of organization ids which participate in the study

Returns
Containing the updated study information

Return type
dict

class AlgorithmSubClient(*parent*)

Bases: SubClient

Subclient for the algorithms from the algorithm store.

create(*name*, *description*, *image*, *partitioning*, *vantage6_version*, *functions*)

Add an algorithm to the algorithm store

Parameters

- **name** (*str*) – Name of the algorithm

- **description** (*str*) – Description of the algorithm
- **image** (*str*) – Docker image of the algorithm
- **partitioning** (*str*) – Partitioning of the algorithm (horizontal or vertical)
- **vantage6_version** (*str*) – Vantage6 version of the algorithm
- **functions** (*list[dict]*) – List of functions of the algorithm. Each function is a dict with the following keys: - name: str

  Name of the function

  - **description: str, optional**
    Description of the function
  - **type: string**
    Type of the function (central or federated)
  - **databases: list[dict]**
    List of databases of the function. Each database is a dict with the following keys: - name: str

    Name of the database

    * **description: str, optional**
      Description of the database
  - **arguments: list[dict]**
    List of arguments of the function. Each argument is a dict with the following keys: - name: str

    Name of the argument

    * **description: str, optional**
      Description of the argument
    * **type: str**
      Type of the argument. Can be 'string', 'integer', 'float', 'boolean', 'json', 'column', 'organization' or 'organizations'

Returns
The created algorithm

Return type
dict

delete(*id_*)

Delete an algorithm from the algorithm store

Parameters
id (*int*) – Id of the algorithm

Returns
The deleted algorithm

Return type
dict

get(*id_*)

Get an algorithm by its id.

Parameters
id (*int*) – The id of the algorithm.

Returns
The algorithm.

Return type
dict

list(*name=None*, *description=None*, *image=None*, *partitioning=None*, *v6_version=None*)

List algorithms

Parameters

- **name** (*str*) – Filter by name (with LIKE operator).
- **description** (*str*) – Filter by description (with LIKE operator).
- **image** (*str*) – Filter by image (with LIKE operator).
- **partitioning** (*str*) – Filter by partitioning (horizontal or vertical).
- **v6_version** (*str*) – Filter by version (with LIKE operator).

Returns
List of algorithms

Return type
list[dict]

class AlgorithmStoreSubClient(*parent*)

Bases: SubClient

Subclient for the algorithm store.

create(*algorithm_store_url*, *name*, *collaboration=None*, *all_collaborations=False*, *force=False*)

Link an algorithm store to one or more collaborations.

Parameters

- **algorithm_store_url** (*str*) – The url of the algorithm store.
- **name** (*str*) – The name of the algorithm store.
- **collaboration** (*int, optional*) – The id of the collaboration to link the algorithm store to. If not given and client.setup_collaboration() was called, the collaboration id from the setup is used. If neither is the case, all_collaborations must be set to True explicitly.
- **all_collaborations** (*bool, optional*) – If True, the algorithm store is linked to all collaborations. If False, the collaboration_id must be given.
- **force** (*bool, optional*) – If True, the algorithm store will be linked to the collaboration even for localhost urls - which is not recommended in production scenarios for security reasons.

Returns
The algorithm store.

Return type
dict

delete(*id_=None*)

Delete an algorithm store.

Parameters
id (*int*) – The id of the algorithm store. If not given, the algorithm store must be set with client.store.set().

Returns
The deleted algorithm store.

Return type
dict

get(*id_*)

Get an algorithm store by its id.

Parameters
id (*int*) – The id of the algorithm store.

Returns
The algorithm store.

Return type
dict

list(*name=None*, *url=None*, *collaboration=None*, *page=1*, *per_page=10*)

List all algorithm stores.

Parameters

- **name** (*str, optional*) – Filter by name (with LIKE operator)
- **url** (*str, optional*) – Filter by algorithm store url (with LIKE operator)
- **collaboration** (*int, optional*) – Filter by collaboration id. If not given and client.setup_collaboration() was called, the collaboration id from the setup is used. Otherwise, all algorithm stores are returned.
- **page** (*int, optional*) – The page number to retrieve.
- **per_page** (*int, optional*) – The number of items to retrieve per page.

Returns
The algorithm stores.

Return type
list[dict]

set(*id_*)

” Set the algorithm store to use for the client.

Parameters
id (*int*) – The id of the algorithm store.

Returns
The algorithm store.

Return type
dict

update(*id_=None*, *name=None*, *collaboration=None*, *all_collaborations=None*)

Update an algorithm store.

Parameters

- **id** (*int*) – The id of the algorithm store. If not given, the algorithm store must be set with client.store.set().
- **name** (*str, optional*) – The name of the algorithm store.

- **collaboration** (*int, optional*) – The id of the collaboration to link the algorithm store to.
- **all_collaborations** (*bool, optional*) – If True, the algorithm store is linked to all collaborations. If False, the collaboration_id must be given.

Returns
The updated algorithm store.

Return type
dict

vantage6.client.utils

class LogLevel(*value*)

Enum for the different log levels

Variables

- **DEBUG** (*str*) – The debug log level
- **INFO** (*str*) – The info log level
- **WARN** (*str*) – The warn log level
- **ERROR** (*str*) – The error log level
- **CRITICAL** (*str*) – The critical log level

4.10.5.2 Custom client exceptions

vantage6.client.exceptions

exception DeserializationException

Exception raised when deserialization of algorithm input or result fails.

4.10.6 Algorithm client and tools

4.10.6.1 Algorithm Client

vantage6.algorithm.client

class AlgorithmClient(*token*, **args*, ***kwargs*)

Bases: ClientBase

Interface to communicate between the algorithm container and the central server via a local proxy server.

An algorithm container cannot communicate directly to the central server as it has no internet connection. The algorithm can, however, talk to a local proxy server which has interface to the central server. This way we make sure that the algorithm container does not share details with others, and we also can encrypt the results for a specific receiver. Thus, this not a interface to the central server but to the local proxy server - however, the interface looks identical to make it easier to use.

Parameters

- **token** (*str*) – JWT (container) token, generated by the node the algorithm container runs on

- ***args** – Arguments passed to the parent ClientBase class.
- ****kwargs** – Arguments passed to the parent ClientBase class.

class Collaboration(*parent*)

Bases: SubClient

Get information about the collaboration.

get()

Get the collaboration data.

Returns
Dictionary containing the collaboration data.

Return type
dict

class Node(*parent*)

Bases: SubClient

Get information about the node.

get()

Get the node data.

Returns
Dictionary containing data on the node this algorithm is running on.

Return type
dict

class Organization(*parent*)

Bases: SubClient

Get information about organizations in the collaboration.

get(*id_*)

Get an organization by ID.

Parameters
id (*int*) – ID of the organization to retrieve

Returns
Dictionary containing the organization data.

Return type
dict

list()

Obtain all organization in the collaboration.

The container runs in a Node which is part of a single collaboration. This method retrieves all organization data that are within that collaboration. This can be used to target specific organizations in a collaboration.

Returns
List of organizations in the collaboration.

Return type
list[dict]

class Result(*parent*)

Bases: SubClient

Result client for the algorithm container.

This client is used to get results from the central server.

from_task(*task_id*)

Obtain results from a specific task at the server.

Containers are allowed to obtain the results of their children (having the same job_id at the server). The permissions are checked at te central server.

Results are decrypted by the proxy server and decoded here before returning them to the algorithm.

Parameters
task_id (*int*) – ID of the task from which you want to obtain the results

Returns
List of results. The type of the results depends on the algorithm.

Return type
list[Any]

get(*id_*)

Obtain a specific result from the central server.

Parameters
id (*int*) – ID of the algorithm run of which the result should be obtained.

Returns
Result of the algorithm run.

Return type
Any

class Run(*parent*)

Bases: SubClient

Algorithm Run client for the algorithm container.

This client is used to obtain algorithm runs of tasks with the same job_id from the central server.

from_task(*task_id*)

Obtain algorithm runs from a specific task at the server.

Containers are allowed to obtain the runs of their children (having the same job_id at the server). The permissions are checked at te central server.

Note that the returned results are not decrypted. The algorithm is responsible for decrypting the results.

Parameters
task_id (*int*) – ID of the task from which you want to obtain the algorithm runs

Returns
List of algorithm run data. The type of the results depends on the algorithm.

Return type
list

`get`(*id_*)

Obtain a specific algorithm run from the central server.

Parameters
id (*int*) – ID of the algorithm run that should be obtained.

Returns
Algorithm run data.

Return type
dict

`class Study`(*parent*)

Bases: `SubClient`

Get information about the study or studies.

`get`(*id_*)

Get the study data by ID.

Parameters
id (*int*) – ID of the study to retrieve

Returns
Dictionary containing study data.

Return type
dict

`list`()

Obtain all studies in the collaboration.

The container runs in a node which is part of a single collaboration, which may contain zero or more studies. This method retrieves all studies that are part of the collaboration.

Returns
List of studies in the collaboration.

Return type
list[dict]

`class Task`(*parent*)

Bases: `SubClient`

A task client for the algorithm container.

It provides functions to get task information and create new tasks.

`create`(*input_*, *organizations=None*, *name='subtask'*, *description=None*)

Create a new (child) task at the central server.

Containers are allowed to create child tasks (having the same job_id) at the central server. The docker image must be the same as the docker image of this container self.

Parameters

- **input** (*bytes*) – Input to the task. Should be b64 encoded.
- **organizations** (*list[int]*) – List of organization IDs that should execute the task.
- **name** (*str, optional*) – Name of the subtask
- **description** (*str, optional*) – Description of the subtask

Returns
Dictionary containing information on the created task

Return type
dict

`get`(*task_id*)

Retrieve a task at the central server.

Parameters
task_id (*int*) – ID of the task to retrieve

Returns
Dictionary containing the task information

Return type
dict

`class VPN`(*parent*)

Bases: `SubClient`

A VPN client for the algorithm container.

It provides functions to obtain the IP addresses of other containers.

`get_addresses`(*only_children=False*, *only_parent=False*, *only_siblings=False*, *only_self=False*, *include_children=False*, *include_parent=False*, *label=None*)

Get information about the VPN IP addresses and ports of other algorithm containers involved in the current task. These addresses can be used to send VPN communication to.

Multiple ports may be exposed for a single algorithm run, so it is possible that multiple ports are returned for a single IP.

Parameters

- **only_children** (*bool, optional*) – Only return the IP addresses of the children of the current task, by default False. Incompatible with other only_* parameters.
- **only_parent** (*bool, optional*) – Only return the IP address of the parent of the current task, by default False. Incompatible with other only_* parameters.
- **only_siblings** (*bool, optional*) – Only return the IP addresses of the siblings of the current task, by default False. Incompatible with other only_* parameters.
- **only_self** (*bool, optional*) – Only return the IP address of the current task, by default False. Incompatible with other only_* parameters.
- **include_children** (*bool, optional*) – Include the IP addresses of the children of the current task, by default False. Incompatible with only_parent, superseded by only_children.
- **include_parent** (*bool, optional*) – Include the IP address of the parent of the current task, by default False. Incompatible with only_children, superseded by only_parent.
- **label** (*str, optional*) – The label of the port you are interested in, which is set in the algorithm Dockerfile. If this parameter is set, only the ports with this label will be returned.

Returns
List of dictionaries with algorithm addresses. Each dictionary contains the keys 'ip', 'port', 'label', 'organization_id', 'task_id', and 'parent_id'. If obtaining the VPN addresses from the server fails, a dictionary with a 'message' key is returned instead.

Return type
list[dict]

`get_child_addresses()`

Get the IP addresses and port numbers of the children of the current algorithm run.

Multiple ports may be exposed for a single algorithm run, so it is possible that multiple ports are returned for a single IP.

Returns
List of dictionaries with algorithm addresses. Each dictionary contains the keys 'ip', 'port', 'label', 'organization_id', 'task_id', and 'parent_id'. If obtaining the VPN addresses from the server fails, a dictionary with a 'message' key is returned instead.

Return type
list[dict]

`get_own_address()`

Get the IP address and port number of the current algorithm run.

Multiple ports may be exposed for a single algorithm run, so it is possible that multiple ports are returned for a single IP.

Returns
List of dictionaries with algorithm addresses. Each dictionary contains the keys 'ip', 'port', 'label', 'organization_id', 'task_id', and 'parent_id'. If obtaining the VPN addresses from the server fails, a dictionary with a 'message' key is returned instead.

Return type
list[dict]

`get_parent_address()`

Get the IP address and port number of the parent of the current algorithm run.

Multiple ports may be exposed for a single algorithm run, so it is possible that multiple ports are returned for a single IP.

Returns
List of dictionaries with algorithm addresses. Each dictionary contains the keys 'ip', 'port', 'label', 'organization_id', 'task_id', and 'parent_id'. If obtaining the VPN addresses from the server fails, a dictionary with a 'message' key is returned instead.

Return type
list[dict]

`get_sibling_addresses()`

Get the IP addresses and port numbers of the siblings of the current algorithm run.

Multiple ports may be exposed for a single algorithm run, so it is possible that multiple ports are returned for a single IP.

Returns
List of dictionaries with algorithm addresses. Each dictionary contains the keys 'ip', 'port', 'label', 'organization_id', 'task_id', and 'parent_id'. If obtaining the VPN addresses from the server fails, a dictionary with a 'message' key is returned instead.

Return type
list[dict]

`authenticate`(*credentials=None*, *path=None*)

Overwrite base authenticate function to prevent algorithm containers from trying to authenticate, which they would be unable to do (they are already provided with a token on container startup).

Function parameters have only been included to make the interface identical to the parent class. They are not used.

Parameters

- **credentials** (*dict*) – Credentials to authenticate with.
- **path** (*str*) – Path to the credentials file.

Raises
NotImplementedError – Always.

Return type
None

`refresh_token`()

Overwrite base refresh_token function to prevent algorithm containers from trying to refresh their token, which they would be unable to do.

Raises
NotImplementedError – Always.

Return type
None

`request`(**args*, ***kwargs*)

Make a request to the central server. This overwrites the parent function so that containers will not try to refresh their token, which they would be unable to do.

Parameters

- ***args** – Arguments passed to the parent ClientBase.request function.
- ****kwargs** – Arguments passed to the parent ClientBase.request function.

Returns
Response from the central server.

Return type
dict

`wait_for_results`(*task_id*, *interval=1*)

Poll the central server until results are available and then return them.

Parameters

- **task_id** (*int*) – ID of the task for which the results should be obtained.
- **interval** (*float*) – Interval in seconds to wait between checking server for results.

Returns
List of task results.

Return type
list

4.10.6.2 Algorithm tools

vantage6.tools.wrappers

This module contains algorithm wrappers. These wrappers are used to provide different data adapters to the algorithms. This way we ony need to write the algorithm once and can use it with different data adapters.

Currently the following wrappers are available:

- `DockerWrapper` (= `CSVWrapper`)
- `SparqlDockerWrapper`
- `ParquetWrapper`
- `SQLWrapper`
- `ExcelWrapper`

When writing the Docker file for the algorithm, the correct wrapper will automatically be selected based on the database type. The database type is set by the vantage6 node based on its configuration file.

`class DatabaseType(`*value*`)`

Bases: `str`, `Enum`

Enum for the different database types.

Variables

- **CSV** (*str*) – CSV database
- **SQL** (*str*) – SQL database
- **EXCEL** (*str*) – Excel database
- **SPARQL** (*str*) – SparQL database
- **PARQUET** (*str*) – Parquet database

`get_column_names(`*database_uri*, *db_type=None*, *query=None*, *sheet_name=None*`)`

Get the column names of dataframe that will be loaded into an algorithm

Parameters

- **database_uri** (*str*) – Path to the database file or URI of the database.
- **db_type** (*str*) – The type of the database. This should be one of the CSV, SQL, Excel, Sparql or Parquet.
- **query** (*str*) – The query to execute on the database. This is required for SQL and Sparql databases.
- **sheet_name** (*str*) – The sheet name to read from the Excel file. This is optional and only for Excel databases.

Returns
The column names of the dataframe

Return type
list[str]

load_csv_data(*database_uri*)

Load the local privacy-sensitive data from the database.

Parameters

database_uri (*str*) – URI of the csv file, supplied by te node

Returns

The data from the csv file

Return type

pd.DataFrame

load_data(*database_uri*, *db_type=None*, *query=None*, *sheet_name=None*)

Read data from database and give it back to the algorithm.

If the database type is unknown, this function will exit. Also, a 'query' is required for SQL and SparQL databases. If it is not present, this function will exit the algorithm.

Parameters

- **database_uri** (*str*) – Path to the database file or URI of the database.
- **db_type** (*str*) – The type of the database. This should be one of the CSV, SQL, Excel, Sparql or Parquet.
- **query** (*str*) – The query to execute on the database. This is required for SQL and Sparql databases.
- **sheet_name** (*str*) – The sheet name to read from the Excel file. This is optional and only for Excel databases.

Returns

The data from the database

Return type

pd.DataFrame

load_excel_data(*database_uri*, *sheet_name=None*)

Load the local privacy-sensitive data from the database.

Parameters

- **database_uri** (*str*) – URI of the excel file, supplied by te node
- **sheet_name** (*str | None*) – Sheet name to be read from the excel file. If None, the first sheet will be read.

Returns

The data from the excel file

Return type

pd.DataFrame

load_parquet_data(*database_uri*)

Load the local privacy-sensitive data from the database.

Parameters

database_uri (*str*) – URI of the parquet file, supplied by te node

Returns

The data from the parquet file

Return type

pd.DataFrame

load_sparql_data(*database_uri*, *query*)

Load the local privacy-sensitive data from the database.

Parameters

- **database_uri** (*str*) – URI of the triplestore, supplied by te node

- **query** (*str*) – Query to retrieve the data from the triplestore

Returns
The data from the triplestore

Return type
pd.DataFrame

`load_sql_data`(*database_uri*, *query*)

Load the local privacy-sensitive data from the database.

Parameters

- **database_uri** (*str*) – URI of the sql database, supplied by te node
- **query** (*str*) – Query to retrieve the data from the database

Returns
The data from the database

Return type
pd.DataFrame

vantage6.tools.wrap

`load_input`(*input_file*)

Load the input from the input file.

Parameters
input_file (*str*) – File containing the input

Returns
input_data – Input data for the algorithm

Return type
Any

Raises
DeserializationError – Failed to deserialize input data

`wrap_algorithm`(*log_traceback=True*)

Wrap an algorithm module to provide input and output handling for the vantage6 infrastructure.

Data is received in the form of files, whose location should be specified in the following environment variables:

- `INPUT_FILE`: input arguments for the algorithm. This file should be encoded in JSON format.
- `OUTPUT_FILE`: location where the results of the algorithm should be stored
- `TOKEN_FILE`: access token for the vantage6 server REST api
- `USER_REQUESTED_DATABASE_LABELS`: comma-separated list of database labels that the user requested
- `<DB_LABEL>_DATABASE_URI`: uri of the each of the databases that the user requested, where `<DB_LABEL>` is the label of the database given in `USER_REQUESTED_DATABASE_LABELS`.

The wrapper expects the input file to be a json file. Any other file format will result in an error.

Parameters

- **module** (*str*) – Python module name of the algorithm to wrap.
- **log_traceback** (*bool*) – Whether to print the full error message from algorithms or not, by default False. Algorithm developers should set this to False if the error messages may contain sensitive information. By default True.

Return type
`None`

vantage6.tools.mock_client

class MockAlgorithmClient(*datasets*, *module*, *collaboration_id=None*, *organization_ids=None*, *node_ids=None*)

The MockAlgorithmClient mimics the behaviour of the AlgorithmClient. It can be used to mock the behaviour of the AlgorithmClient and its communication with the server.

Parameters

- **datasets** (*list[list[dict]]*) – A list that contains the datasets that are used in the mocked algorithm. The inner list contains the datasets for each organization; the outer list is for each organization. A single dataset should be described as a dictionary with the same keys as in a node configuration:
  - database: str (path to file or SQL connection string) or pd.DataFrame
  - db_type (str, e.g. "csv" or "sql")

  There are also a number of keys that are optional but may be required depending on the database type: - query: str (required for SQL/Sparql databases) - sheet_name: str (optional for Excel databases) - preprocessing: dict (optional, see the documentation for

  preprocessing for more information)

  Note that if the database is a pandas DataFrame, the type and input_data keys are not required.
- **module** (*str*) – The name of the module that contains the algorithm.
- **collaboration_id** (*int, optional*) – Sets the mocked collaboration id to this value. Defaults to 1.
- **organization_ids** (*list[int], optional*) – Set the organization ids to this value. The first value is used for this organization, the rest for child tasks. Defaults to [0, 1, 2, ..].
- **node_ids** (*list[int], optional*) – Set the node ids to this value. The first value is used for this node, the rest for child tasks. Defaults to [0, 1, 2, . . .].

class Collaboration(*parent*)

Collaboration subclient for the MockAlgorithmClient

get(*is_encrypted=True*)

Get mocked collaboration

Parameters

is_encrypted (*bool*) – Whether the collaboration is encrypted or not. Default True.

Returns

A mocked collaboration.

Return type

dict

class Node(*parent*)

Node subclient for the MockAlgorithmClient

get(*is_online=True*)

Get mocked node

Parameters
is_online (*bool*) – Whether the node is online or not. Default True.

Returns
A mocked node.

Return type
dict

class Organization(*parent*)

Organization subclient for the MockAlgorithmClient

get(*id_*)

Get mocked organization by ID

Parameters
id (*int*) – The id of the organization.

Returns
A mocked organization.

Return type
dict

list()

Get mocked organizations in the collaboration.

Returns
A list of mocked organizations in the collaboration.

Return type
list[dict]

class Result(*parent*)

Result subclient for the MockAlgorithmClient

from_task(*task_id*)

Return the results of the task with the given id.

Parameters
task_id (*int*) – The id of the task.

Returns
The results of the task.

Return type
list[Any]

get(*id_*)

Get mocked result by ID

Parameters
id (*int*) – The id of the result.

Returns
A mocked result.

Return type
Any

class Run(*parent*)

Run subclient for the MockAlgorithmClient

from_task(*task_id*)

Get mocked runs by task ID

Parameters
task_id (*int*) – The id of the task.

Returns
A list of mocked runs.

Return type
list[dict]

get(*id_*)

Get mocked run by ID

Parameters
id (*int*) – The id of the run.

Returns
A mocked run.

Return type
dict

class SubClient(*parent*)

Create sub groups of commands using this SubClient

Parameters
parent (MockAlgorithmClient) – The parent client

class Task(*parent*)

Task subclient for the MockAlgorithmClient

create(*input_*, *organizations*, *name='mock'*, *description='mock'*)

Create a new task with the MockProtocol and return the task id.

Parameters

- **input** (*dict*) – The input data that is passed to the algorithm. This should at least contain the key 'method' which is the name of the method that should be called. Other keys depend on the algorithm.
- **organizations** (*list[int]*) – A list of organization ids that should run the algorithm.
- **name** (*str, optional*) – The name of the task, by default "mock"
- **description** (*str, optional*) – The description of the task, by default "mock"

Returns
A dictionary with information on the created task.

Return type
task

get(*task_id*)

Return the task with the given id.

Parameters
task_id (*int*) – The id of the task.

Returns
The task details.

Return type
dict

`wait_for_results`(*task_id*, *interval=1*)

Mock waiting for results - just return the results as tasks are completed synchronously in the mock client.

Parameters

- **task_id** (*int*) – ID of the task for which the results should be obtained.
- **interval** (*float*) – Interval in seconds between checking for new results. This is ignored in the mock client but included to match the signature of the AlgorithmClient.

Returns
List of task results.

Return type
list

vantage6.tools.util

`error`(*msg*)

Print an error message to stdout.

Parameters
msg (*str*) – Error message to be printed

Return type
None

`get_env_var`(*var_name*, *default=None*)

Get the value of an environment variable. Environment variables are encoded by the node so they need to be decoded here.

Note that this decoding follows the reverse of the encoding in the node: first replace '=' back and then decode the base32 string.

Parameters

- **var_name** (*str*) – Name of the environment variable
- **default** (*str | None*) – Default value to return if the environment variable is not found

Returns
var_value – Value of the environment variable, or default value if not found

Return type
str | None

`info`(*msg*)

Print an info message to stdout.

Parameters
msg (*str*) – Message to be printed

Return type
None

`warn`(*msg*)

Print a warning message to stdout.

Parameters
msg (*str*) – Warning message to be printed

Return type
None

4.10.7 Backend common

4.10.7.1 Services resources base

vantage6.backend.common.services_resources.BaseServicesResources

vantage6.backend.common.resource.output_schema.BaseHATEOASModelSchema

vantage6.backend.common.resource.pagination

4.10.8 Common functions (vantage6-common)

This page contains the API reference of the functions in the vantage-common package.

4.10.8.1 vantage6.common.configuration_manager

class Configuration(**args*, ***kwargs*)

Base class to contain a single configuration.

property is_valid: bool

Check if the configuration is valid.

Returns
Whether or not the configuration is valid.

Return type
bool

class ConfigurationManager(*conf_class=<class 'vantage6.common.configuration_manager.Configuration'>*, *name=None*)

Class to maintain valid configuration settings.

Parameters

- **conf_class** (Configuration) – The class to use for the configuration.
- **name** (*str*) – The name of the configuration.

classmethod from_file(*path*, *conf_class=<class 'vantage6.common.configuration_manager.Configuration'>*)

Load a configuration from a file.

Parameters

- **path** (*Path | str*) – The path to the file to load the configuration from.
- **conf_class** (*Type[Configuration]*) – The class to use for the configuration.

Returns
The configuration manager with the configuration.

Return type
ConfigurationManager

Raises
AssertionError – If the name of the configuration could not be extracted from the file path.

`get()`

Get a configuration from the configuration manager.

Returns
The configuration.

Return type
Configuration

`property is_empty: bool`

Check if the configuration manager is empty.

Returns
Whether or not the configuration manager is empty.

Return type
bool

`load(path)`

Load a configuration from a file.

Parameters
path (*Path | str*) – The path to the file to load the configuration from.

Return type
None

`put(config)`

Add a configuration to the configuration manager.

Parameters
config (*dict*) – The configuration to add.

Raises
AssertionError – If the configuration is not valid.

Return type
None

`save(path)`

Save the configuration to a file.

Parameters
path (*Path | str*) – The path to the file to save the configuration to.

Return type
None

4.10.8.2 vantage6.common.context

`class AppContext(instance_type, instance_name, system_folders=False, config_file=None, print_log_header=True)`

Base class from which to create Node and Server context classes.

`INST_CONFIG_MANAGER`

alias of *ConfigurationManager*

`classmethod available_configurations(instance_type, system_folders)`

Returns a list of configuration managers and a list of paths to configuration files that could not be loaded.

Parameters

- **instance_type** (*InstanceType*) – Type of instance that is checked

- **system_folders** (*bool*) – Use system folders rather than user folders

Returns
A list of configuration managers and a list of paths to configuration files that could not be loaded.

Return type
list[*ConfigurationManager*], list[Path]

classmethod config_exists(*instance_type*, *instance_name*, *system_folders=False*)

Check if a config file exists for the given instance type and name.

Parameters

- **instance_type** (*InstanceType*) – Type of instance that is checked
- **instance_name** (*str*) – Name of the configuration
- **system_folders** (*bool*) – Use system folders rather than user folders

Returns
True if the config file exists, False otherwise

Return type
bool

property config_file: Path

Return the path to the configuration file.

Returns
Path to the configuration file

Return type
Path

property config_file_name: str

Return the name of the configuration file.

Returns
Name of the configuration file

Return type
str

static configure_logger(*name*, *level*)

Set the logging level of a logger.

Parameters

- **name** (*str*) – Name of the logger to configure. If *None*, the root logger is configured.
- **level** (*str*) – Logging level to set. Must be one of 'debug', 'info', 'warning', 'error', 'critical'.

Returns
The logger object and the logging level that was set.

Return type
Tuple[Logger, int]

classmethod find_config_file(*instance_type*, *instance_name*, *system_folders*, *config_file=None*, *verbose=True*)

Find a configuration file.

Parameters

- **instance_type** (*InstanceType*) – Type of instance that is checked
- **instance_name** (*str*) – Name of the configuration
- **system_folders** (*bool*) – Use system folders rather than user folders
- **config_file** (*str | None*) – Name of the configuration file. If None, the name of the configuration is used.
- **verbose** (*bool*) – Print the directories that are searched for the configuration file.

Returns
Path to the configuration file

Return type
str

Raises
Exception – If the configuration file is not found

classmethod from_external_config_file(*path*, *instance_type*, *system_folders=False*)

Create a new AppContext instance from an external config file.

Parameters

- **path** (*str*) – Path to the config file
- **instance_type** (*InstanceType*) – Type of instance for which the config file is used
- **system_folders** (*bool*) – Use system folders rather than user folders

Returns
A new AppContext instance

Return type
AppContext

get_data_file(*filename*)

Return the path to a data file.

Parameters
filename (*str*) – Name of the data file

Returns
Path to the data file

Return type
str

initialize(*instance_type*, *instance_name*, *system_folders=False*, *config_file=None*, *print_log_header=True*)

Initialize the AppContext instance.

Parameters

- **instance_type** (*str*) – 'server' or 'node'
- **instance_name** (*str*) – Name of the configuration
- **system_folders** (*bool*) – Use system folders rather than user folders
- **config_file** (*str*) – Path to a specific config file. If left as None, OS specific folder will be used to find the configuration file specified by *instance_name*.
- **print_log_header** (*bool*) – Print a banner to the log file.

Return type
None

`static instance_folders(instance_type, instance_name, system_folders)`

Return OS and instance specific folders for storing logs, data and config files.

Parameters

- **instance_type** (*InstanceType*) – Type of instance that is checked
- **instance_name** (*str*) – Name of the configuration
- **system_folders** (*bool*) – Use system folders rather than user folders

Returns
Dictionary with Paths to the folders of the log, data and config files.

Return type
dict

`property log_file: Path`

Return the path to the log file.

Returns
Path to the log file

Return type
Path

`log_file_name(type_)`

Return a path to a log file for a given log file type

Parameters
type (*str*) – The type of log file to return.

Returns
The path to the log file.

Return type
Path

Raises
AssertionError – If the configuration manager is not initialized.

`print_log_header()`

Print the log file header.

Return type
None

`set_folders(instance_type, instance_name, system_folders)`

Set the folders where the configuration, data and log files are stored.

Parameters

- **instance_type** (*InstanceType*) – Type of instance that is checked
- **instance_name** (*str*) – Name of the configuration
- **system_folders** (*bool*) – Whether to use system folders rather than user folders

Return type
None

`setup_logging()`

Setup a basic logging mechanism.

Exits if the log file can't be created.

Return type
None

static type_data_folder(*instance_type*, *system_folders*)

Return OS specific data folder.

Parameters

- **instance_type** (*InstanceType*) – Type of instance that is checked
- **system_folders** (*bool*) – Use system folders rather than user folders

Returns
Path to the data folder

Return type
Path

4.10.8.3 vantage6.common.encryption

Encryption between organizations

Module to provide async encrpytion between organizations. All input and result fields should be encrypted when communicating to the central server.

All incomming messages (input/results) should be encrypted using the public key of this organization. This way we can decrypt them using our private key.

In the case we are sending messages (input/results) we need to encrypt it using the public key of the receiving organization. (retreiving these public keys is outside the scope of this module).

class CryptorBase(**args*, ***kwargs*)

Base class/interface for encryption implementations.

static bytes_to_str(*data*)

Encode bytes as base64 encoded string.

Parameters
data (*bytes*) – The data to encode.

Returns
The base64 encoded string.

Return type
str

decrypt_str_to_bytes(*data*)

Decrypt base64 encoded *string* data.

Parameters
data (*str*) – The data to decrypt.

Returns
The decrypted data.

Return type
bytes

encrypt_bytes_to_str(*data*, *pubkey_base64*)

Encrypt bytes in *data* using a (base64 encoded) public key.

Note that the public key is ignored in this base class. If you want to encode your data with a public key, use the *RSACryptor* class.

Parameters

- **data** (*bytes*) – The data to encrypt.
- **pubkey_base64** (*str*) – The public key to use for encryption. This is ignored in this base class.

Returns
The encrypted data encoded as base64 string.

Return type
str

static str_to_bytes(*data*)

Decode base64 encoded string to bytes.

Parameters
data (*str*) – The base64 encoded string.

Returns
The encoded string converted to bytes.

Return type
bytes

class DummyCryptor(**args*, ***kwargs*)

Does absolutely nothing to encrypt the data.

class RSACryptor(*private_key_file*)

Wrapper class for the cryptography package.

It loads the private key, and has an interface to encrypt en decrypt messages. If no private key is found, it can generate one, and store it at the default location. The encrpytion can be done via a public key from another organization, make sure the key is in the right data-type.

Communication between node and server requires serialization (and deserialization) of the encrypted messages (which are in bytes). The API can not communicate bytes, therefore a base64 conversion needs to be executed (and also a utf-8 encoding needs to be applied because of the way python implemented base64). The same goes for sending and receiving the public_key.

Parameters
private_key_file (*Path*) – The path to the private key file.

static create_new_rsa_key(*path*)

Creates a new RSA key for E2EE.

Parameters
path (*Path*) – The path to the private key file.

Returns
The newly created private key.

Return type
RSAPrivateKey

static create_public_key_bytes(*private_key*)

Create a public key from a private key.

Parameters
private_key (*RSAPrivateKey*) – The private key to use.

Returns
The public key as bytes.

Return type
bytes

`decrypt_str_to_bytes`(*data*)

Decrypt base64 encoded *string* data.

Parameters

data (*str*) – The data to decrypt.

Returns

The decrypted data.

Return type

bytes

`encrypt_bytes_to_str`(*data*, *pubkey_base64s*)

Encrypt bytes in *data* using a (base64 encoded) public key.

Parameters

- **data** (*bytes*) – The data to encrypt.
- **pubkey_base64s** (*str*) – The public key to use for encryption.

Returns

The encrypted data encoded as base64 string.

Return type

str

`property public_key_bytes: bytes`

Returns the public key bytes from the organization.

Returns

The public key as bytes.

Return type

bytes

`property public_key_str: str`

Returns a JSON safe public key, used for the API.

Returns

The public key as base64 encoded string.

Return type

str

`verify_public_key`(*pubkey_base64*)

Verifies the public key.

Compare a public key with the generated public key from the private key that is stored in this instance. This is usefull for verifying that the public key stored on the server is derived from the currently used private key.

Parameters

pubkey_base64 (*str*) – The public key to verify as returned from the server.

Returns

True if the public key is valid, False otherwise.

Return type

bool

4.10.8.4 vantage6.common

class ClickLogger

"Logs output to the click interface.

static debug(*msg*)

Print a debug message to the click interface.

Parameters
msg (*str*) – The message to print.

Return type
None

static error(*msg*)

Print an error message to the click interface.

Parameters
msg (*str*) – The message to print.

Return type
None

static info(*msg*)

Print an info message to the click interface.

Parameters
msg (*str*) – The message to print.

Return type
None

static warn(*msg*)

Print a warning message to the click interface.

Parameters
msg (*str*) – The message to print.

Return type
None

class Singleton

Singleton metaclass. It allows us to create just a single instance of a class to which it is the metaclass.

class WhoAmI(*type_: str*, *id_: int*, *name: str*, *organization_name: str*, *organization_id: int*)

Data-class to store Authenticatable information in.

Variables

- **type** (*str*) – The type of the authenticatable (user or node).
- **id** (*int*) – The id of the authenticatable.
- **name** (*str*) – The name of the authenticatable.
- **organization_name** (*str*) – The name of the organization of the authenticatable.
- **organization_id** (*int*) – The id of the organization of the authenticatable.

id_: int

Alias for field number 1

name: str

Alias for field number 2

organization_id: int

Alias for field number 4

organization_name: str

Alias for field number 3

type_: str

Alias for field number 0

base64s_to_bytes(*bytes_string*)

Convert base64 encoded string to bytes.

Parameters

bytes_string (*str*) – The base64 encoded string.

Returns

The encoded string converted to bytes.

Return type

bytes

bytes_to_base64s(*bytes_*)

Convert bytes into base64 encoded string.

Parameters

bytes (*bytes*) – The bytes to convert.

Returns

The base64 encoded string.

Return type

str

check_config_writeable(*system_folders=False*)

Check if the user has write permissions to create the configuration file.

Parameters

system_folders (*bool*) – Whether to check the system folders or the user folders.

Returns

Whether the user has write permissions to create the configuration file or not.

Return type

bool

debug(*msg*)

Print a debug message to the CLI.

Parameters

msg (*str*) – The message to print.

Return type

None

echo(*msg*, *level='info'*)

Print a message to the CLI.

Parameters

- **msg** (*str*) – The message to print.
- **level** (*str*) – The level of the message. Can be one of: "error", "warn", "info", "debug".

Return type

None

error(*msg*)

Print an error message to the CLI.

Parameters
msg (*str*) – The message to print.
Return type
None

generate_apikey()
Creates random api_key using uuid.
Returns
api_key
Return type
str

get_config_path(*dirs*, *system_folders=False*)
Get the path to the configuration directory.
Parameters
- **dirs** (*appdirs.AppDirs*) – The appdirs object.
- **system_folders** (*bool*) – Whether to get path to the system folders or the user folders.

Returns
The path to the configuration directory.
Return type
str

get_database_config(*databases*, *label*)
Get database configuration from config file
Parameters
- **databases** (*list[dict]*) – List of database configurations
- **label** (*str*) – Label of database configuration to retrieve

Returns
Database configuration, or None if not found
Return type
Dict | None

info(*msg*)
Print an info message to the CLI.
Parameters
msg (*str*) – The message to print.
Return type
None

is_ip_address(*ip*)
Test if input IP address is a valid IP address
Parameters
ip (*str*) – IP address to validate
Returns
bool
Return type
whether or not IP address is valid

logger_name(*special__name__*)
Return the name of the logger.
Parameters
special__name__ (*str*) – The __name__ variable of a module.
Returns
The name of the logger.

Return type
str

`warning`(*msg*)

Print a warning message to the CLI.

Parameters
msg (*str*) – The message to print.

Return type
None

4.10.8.5 vantage6.common.docker.addons

`class ContainerKillListener`

Listen for signals that the docker container should be shut down

`exit_gracefully`(**args*)

Set kill_now to True. This will trigger the container to stop

Return type
None

`check_docker_running`()

Check if docker engine is running. If not, exit the program.

Return type
None

`delete_network`(*network*, *kill_containers=True*)

Delete network and optionally its containers

Parameters

- **network** (*Network*) – Network to delete
- **kill_containers** (*bool*) – Whether to kill the containers in the network (otherwise they are merely disconnected)

Return type
None

`delete_volume_if_exists`(*client*, *volume_name*)

Delete a volume if it exists

Parameters

- **client** (*docker.DockerClient*) – Docker client
- **volume** (*Volume*) – Volume to delete

Return type
None

`get_container`(*docker_client*, ***filters*)

Return container if it exists after searching using kwargs

Parameters

- **docker_client** (*DockerClient*) – Python docker client
- ****filters** – These are arguments that will be passed to the client.container.list() function. They should yield 0 or 1 containers as result (e.g. name='something')

Returns
Container if it exists, else None

Return type
Container or None

get_network(*docker_client*, ***filters*)

Return network if it exists after searching using kwargs

Parameters

- **docker_client** (*DockerClient*) – Python docker client
- ****filters** – These are arguments that will be passed to the client.network.list() function. They should yield 0 or 1 networks as result (e.g. name='something')

Returns

Container if it exists, else None

Return type

Container or None

get_networks_of_container(*container*)

Get list of networks the container is in

Parameters

container (*Container*) – The container in which we are interested

Returns

Describes container's networks and their properties

Return type

dict

get_num_nonempty_networks(*container*)

Get number of networks the container is in where it is not the only one

Parameters

container (*Container*) – The container in which we are interested

Returns

Number of networks in which the container resides in which there are also other containers

Return type

int

get_server_config_name(*container_name*, *scope*)

Get the configuration name of a server from its docker container name

Docker container name of the server is formatted as f"{APPNAME}-{self.name}-{self.scope}-server". This will return {self.name}

Parameters

- **container_name** (*str*) – Name of the docker container in which the server is running
- **scope** (*str*) – Scope of the server (e.g. 'system' or 'user')

Returns

A server's configuration name

Return type

str

pull_image(*docker_client*, *image*)

Pull a docker image

Parameters

- **docker_client** (*DockerClient*) – A Docker client
- **image** (*str*) – Name of the image to pull

Raises

docker.errors.APIError – If the image could not be pulled

Return type

None

remove_container(*container*, *kill=False*)

Removes a docker container

Parameters

- **container** (*Container*) – The container that should be removed
- **kill** (*bool*) – Whether or not container should be killed before it is removed

Return type

None

remove_container_if_exists(*docker_client*, ***filters*)

Kill and remove a docker container if it exists

Parameters

- **docker_client** (*DockerClient*) – A Docker client
- ****filters** – These are arguments that will be passed to the client.container.list() function. They should yield 0 or 1 containers as result (e.g. name='something')

Return type

None

running_in_docker()

Check if this code is executed within a Docker container.

Returns

True if the code is executed within a Docker container, False otherwise

Return type

bool

stop_container(*container*, *force=False*)

Stop a docker container

Parameters

- **container** (*Container*) – The container that should be stopped
- **force** (*bool*) – Whether to kill the container or if not, try to stop it gently

4.10.8.6 vantage6.common.docker.network_manager

class NetworkManager(*network_name*)

Handle a Docker network

connect(*container_name*, *aliases=None*, *ipv4=None*)

Connect a container to the network.

Parameters

- **container_name** (*str*) – Name of the container that should be connected to the network
- **aliases** (*list[str]*) – A list of aliases for the container in the network
- **ipv4** (*str | None*) – An IP address to assign to the container in the network

Return type

None

contains(*container*)

Whether or not this network contains a certain container

Parameters

container (*Container*) – container to look for in network

Returns
Whether or not container is in the network

Return type
bool

`create_network`(*is_internal=True*)

Creates an internal (docker) network

Used by algorithm containers to communicate with the node API.

Parameters
is_internal (*bool*) – True if network should only be able to communicate internally

Return type
None

`delete`(*kill_containers=True*)

Delete network

Parameters
kill_containers (*bool*) – If true, kill and remove any containers in the network

Return type
None

`disconnect`(*container_name*)

Disconnect a container from the network.

Parameters
container – Name of the container to disconnect

Return type
None

`get_container_ip`(*container_name*)

Get IP address of a container in the network

Parameters
container_name (*str*) – Name of the container whose IP address is sought

Returns
IP address of the container in the network

Return type
str

`remove_subnet_mask`(*ip*)

Remove the subnet mask of an ip address, e.g. 172.1.0.0/16 -> 172.1.0.0

Parameters
ip (*str*) – IP subnet, potentially including a mask

Returns
IP subnet address without the subnet mask

Return type
str

4.10.8.7 vantage6.common.task_status

class TaskStatus(*value*)

Enum to represent the status of a task

has_task_failed(*status*)

Check if task has failed to run to completion

Parameters

status (TaskStatus | *str*) – The status of the task

Returns

True if task has failed, False otherwise

Return type

bool

has_task_finished(*status*)

Check if task has finished or crashed

Parameters

status (TaskStatus | *str*) – The status of the task

Returns

True if task has finished or failed, False otherwise

Return type

bool

4.10.8.8 vantage6.common.colors

ColorStreamHandler

alias of *_AnsiColorStreamHandler*

class _AnsiColorStreamHandler(*stream=None*)

Handler for logging colors to a stream, for example sys.stderr or sys.stdout.

This handler is used for non-Windows systems.

classmethod _get_color(*level*)

Define which color to print for each log level.

Parameters

level (*int*) – The log level.

Returns

The color to print.

Return type

str

format(*record*)

Format the log record.

Parameters

record (*logging.LogRecord*) – The log record to format.

Returns

The formatted log record.

Return type

str

class _WinColorStreamHandler(*stream=None*)

Color stream handler for Windows systems.

classmethod _get_color(*level*)

Define which color to print for each log level.

Parameters
level (*int*) – The log level.

Returns
The color to print.

Return type
str

emit(*record*)

Write a log record to the stream.

Parameters
record (*logging.LogRecord*) – The log record to write.

Return type
None

4.10.8.9 vantage6.common.exceptions

exception AuthenticationException

Exception to indicate authentication has failed

4.11 Glossary

The following is a list of definitions used in vantage6.

A

- **Autonomy:** the ability of a party to be in charge of the control and management of its own data.

C

- **Collaboration**: an agreement between two or more parties to participate in a study (i.e., to answer a research question).

D

- **Distributed learning**: see *Federated Learning*
- **Docker:** a platform that uses operating system virtualization to deliver software in packages called containers. It is worth noting that although they are often confused, Docker containers are not virtual machines.
- **Data Station**: Virtual Machine containing the vantage6-node application and a database.

F

- **FAIR data**: data that are Findable, Accessible, Interoperable, and Reusable. For more information, see the original paper.
- **Federated learning**: an approach for analyzing data that are spread across different parties. Its main idea is that parties run computations on their local data, yielding either aggregated parameters or encrypted values. These are then shared to generate a global (statistical) model. In other words, instead of bringing the data to the algorithms, federated learning brings the algorithms to the data. This way, patient-sensitive information is not disclosed. Federated learning is some times known as *distributed learning*. However, we try to avoid this term,

since it can be confused with distributed computing, where different computers share their processing power to solve very complex calculations.

H

- **Heterogeneity**: the condition in which in a federated learning scenario, parties are allowed to have differences in hardware and software (i.e., operating systems).
- **Horizontally-partitioned data**: data spread across different parties where the latter have the same features of different instances (i.e., patients). See also vertically-partitioned data.

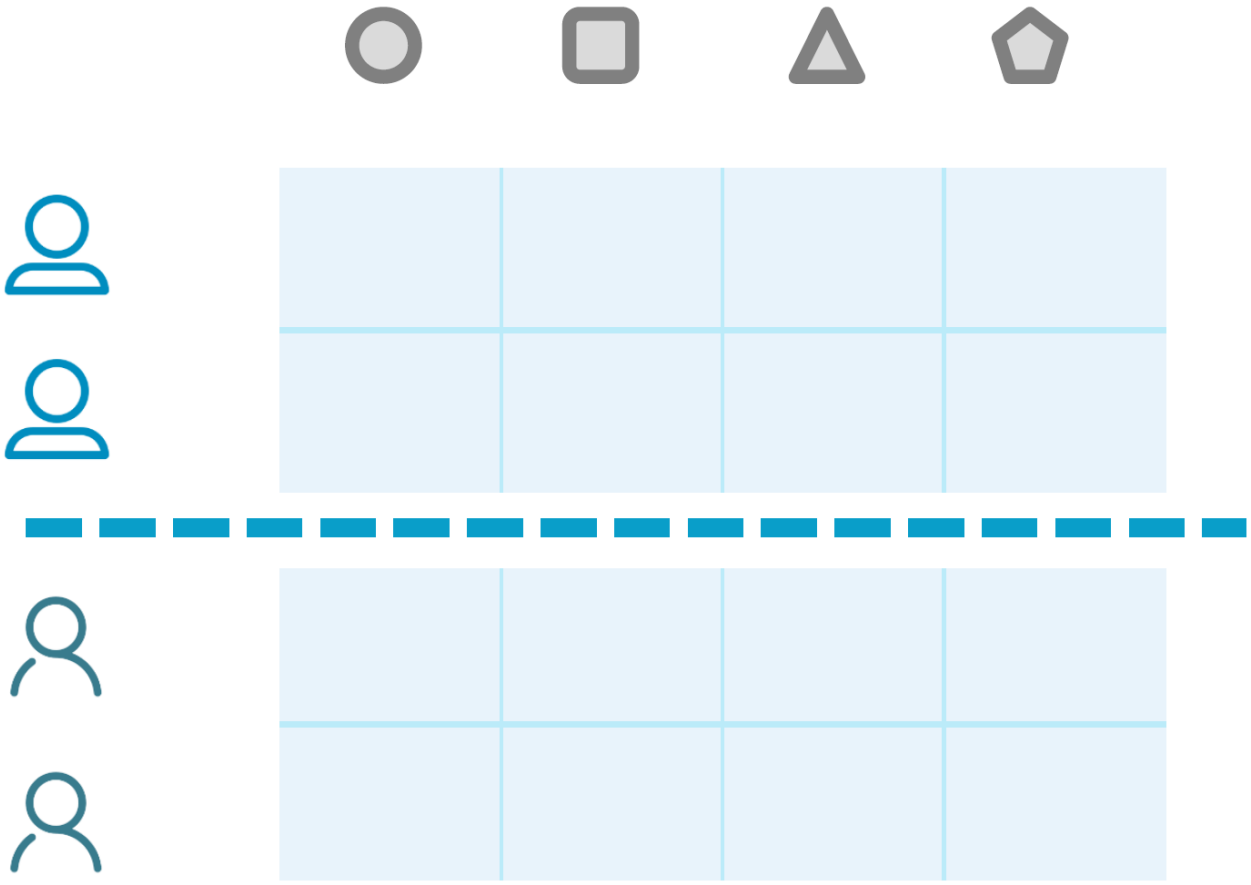


Fig. 4.9: Horizontally-partitioned data

N

- **Node**: vantage6 node application that runs at a **Data Station** which has access to the local data.

M

- **Multi-party computation**: an approach to perform analyses across different parties by performing operations on encrypted data.

P

- **Party**: an entity that takes part in one (or more) collaborations
- **Python**: a high-level general purpose programming language. It aims to help programmers write clear, logical code. vantage6 is written in Python.

S

- **Secure multi-party computation**: see *Multi-party computation*
- **Server**: Public access point of the vantage6 infrastructure. Contains at least the **vantage6-server** application but can also host the optional components: Docker registry, VPN server and RabbitMQ. In this documentation space we try to be explicit when we talk about *server* and *vantage6 server*, however you might encounter *server* where *vantage6 server* should have been.

V

- **vantage6**: priVAcy preserviNg federaTed leArninG infrastructurE for Secure Insight eXchange. In short, vantage6 is an infrastructure for executing federated learning analyses. However, it can also be used as a FAIR data station and as a model repository.
- **Vertically-partitioned data**: data spread across different parties where the latter have different features of the same instances (i.e., patients). See also horizontally-partitioned data.

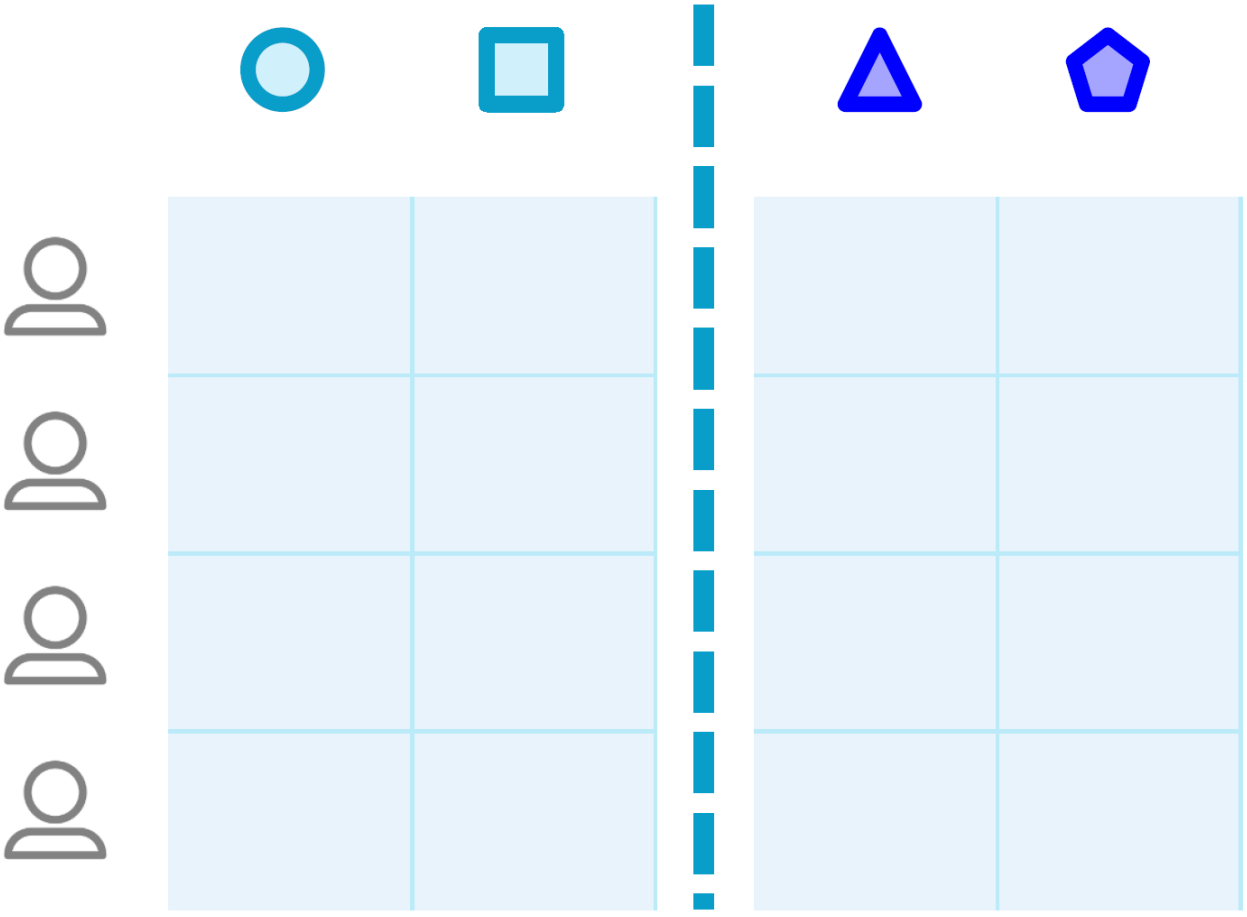


Fig. 4.10: Vertically partitioned data

4.12 Release notes

4.12.1 4.4.0

15 April 2024

- **Feature**
- Added visualization of a results table to the UI. The algorithm store is used to store how the table should be visualized. (Issue#1057, PR#1195).
- Support for more types of algorithm arguments via the UI: lists of strings, ints, floats and columns, and booleans (Issue#1119, PR#1190).
- Added configuration option to link algorithm stores to a server via the server configuration (PR#1156).
- Added a bunch of custom exceptions for algorithms to the algorithm tools (Issue#1185, PR#1205).
- Decoding the environment variables automatically in the algorithm wrapper, to prevent that a user has to decode them manually (Issue#1056, PR#1197).
- Add option to delete roles in the UI (Issue#1113, PR#1199).
- Add option to register a node in the UI *after* creating/editing the collaboration (Issue#1122, PR#1202).
- **Change**
- Updated idna dependency
- **Bugfix**
- Do not mark algorithm runs as killed if they were completed before the user killed the task to which the runs belong (Issue#1045, PR#1204).
- Fix UI code in a few places where pagination was not implemented properly (Issue#1126, PR#1203).

4.12.2 4.3.4

09 April 2024

- **Security**
- Updated express dependency in UI to 4.19.2
- **Feature**
- Added option to add hostname mappings in the node configuration (Issue#1094, PR#1167).
- **Change**
- Always pull new Docker images instead of checking timestamps and only pulling image if the remote image is newer (Issue#1188, Issue#1105, PR#1169).
- Changed behaviour of `v6 algorithm update` to skip previously-answered questions by default, and added flag that allows changing them. Also added flag to allow using a Python script in the updated copier template (PR#1176).
- **Bugfix**
- Fix encoding of non-string algorithm environment variables by casting them to string (PR#1186).
- Fix bug in algorithm client: only send study ID when it is defined (PR#1184).
- Update copier dependency which was causing a CLI error (PR#1187).

4.12.3 4.3.3

25 March 2024

- **Change**
- Improved integration algorithm store in UI (PR#1163).
- Improve picking an online node when creating task in the UI: pick one that shares configuration and give more specific information to the user in case certain data could not be retrieved (PR#1164).
- UI dependency updates
- **Bugfix**
- Fix pulling algorithms from registries that require authentication (Issue#1168, PR#1169).
- Fix bug in showing create task button in UI (PR#1165).
- Could not view studies with collaboration scope permissions (Issue#1154, PR#1157).
- Fix bug when viewing algorithm stores with organization scope permissions (PR#1159).
- Detect whitelisted server in algorithm store if port `443` or `80` at the end of the URL is the only difference with the whitelisted URL (Issue#1155, PR#1162).
- Better error message in Python client when trying to send requests to algorithm store when it has not yet been set up (Issue#1134, PR#1158).

4.12.4 4.3.2

20 March 2024

- **Change**
- Integrated user interface in main repository PR#1112).
- **Bugfix**
- Allow usernames to contain dots and don't apply username validation to login endpoints until v5 to allow existing users to login (PR#1148).

4.12.5 4.3.1

18 March 2024

- **Feature**
- New configuration option to set a server name in the server configuration file, which will be used to identify the server in a two-factor app. (Issue#1016, PR#1075).
- **Change**
- Allow user with organization scope permission to view studies to retrieve studies for a particular collaboration, even though they may not be able to view them all (PR#1104).
- Add option to set policies on openness of algorithm viewing in algorithm store to configuration wizard (PR#1106).
- Improved help text in UI in several places and show the username in the top right (PR#254, PR#257)
- **Bugfix**

- Update default roles on server startup if they have changed. This may happen on minor version updates (Issue#1102, PR#1103).
- Update selected collaboration in the UI when it is updated in the administration section (PR#253)
- Fix showing the create task button if user has no global permissions (PR#259)
- Remove wrong message for CORS not functioning properly with default settings (PR#1107).

4.12.6 4.3.0

12 March 2024

- **Security**
- Implemented configuration option to set CORS origins on the central server. This may be used to further enhance the security profile of your server (advisory, commit).
- Prevent username enumeration attack on endpoints where password and 2FA are reset (advisory, commit).
- Added HTTP security headers on the user interface to provide an additional layer of security to help mitigate attacks and vulnerabilites (advisory, commit).
- Updated cryptography dependency
- **Feature**
- New user interface. The new UI is a complete rewrite of the old UI and is more focused on facilitating the researcher in running tasks and viewing their progress and results (PR#930).
- New infrastructure component: the algorithm store. The algorithm store is a place to make algorithms easily findable and easier to run. Algorithm stores can be made available to specific collaborations or to all collaborations in an entire vantage6 server. By doing so, the new UI will automatically pick up these algorithms and guide the user through running analyses with them (Issue#911, PR#1048 and several other PRs)
- Introducing 'study' concept. A study is essentially a 'sub-collaboration', where a subset of organizations of the collaboration can work together on a specific research question. Tasks and results are then easily grouped together for the study (Issue#812, PR#1069).
- Add flag whether role is default or not (Issue#949, PR#1063).
- Report username/password combination at the end of the logs when it is created (Issue#830, PR#1041).
- **Change**
- Introducing new package `vantage6-backend-common` for code that is shared between the central server and the algorithm store (Issue#979, PR#1037).
- Show the default values for CLI commands when displaying the help text (Issue#1000, PR#1070).
- Setting the allowed algorithms is now part of the questionnaire on node setup (PR#1046).
- Usernames are now required to be at least three characters long and contain only roman letters, numbers, and the characters '_' and '-' (PR#1060).
- Remove OMOP wrapper since we now have specific connectors to connect to this database type and wrapper was therefore not used (Issue#1002, PR#1067).
- `v6 node` commands no longer require full path when using the `--config` option (Issue#870, PR#1042).
- Apply black code formatting to the entire repository (Issue#968, PR#1012).
- Remove option to update organization or collaboration of an existing node. Preferred workflow in that case is to delete and re-create it. Also add option `clear_ip` to clear the VPN IP address of the node (PR#1053).

- **Bugfix**
- Fix VPN network cleanup if `iptables-legacy` is installed, and improve cleanup of the node's containers, volumes and networks when the node is stopped (Issue#1058, PR#1059).
- Prevent logger thread to crash on input that it cannot read (Issue#879, PR#1043).
- Fixed setting up VPN network on Ubuntu 22.04 (Issue#724, PR#1044).

4.12.7 4.2.3

21 February 2024

- **Security**
  - Updated `cryptography` dependency to version `42.0.2` (PR#1047, PR#1048).
- **Feature**
  - Added the option to specify a private key file when using the `v6 test feature-test` command (Issue#1018, PR#1019).
- **Bugfix**
  - Using the whitelisting feature without VPN prevented algorithm containers from starting (PR#1055)
  - Shutting down the node did not properly remove all containers, volumes and networks (PR#1059).

4.12.8 4.2.2

26 January 2024

- **Feature**
- Configuration options for the node to add extra mounts and extra environment variables for the node itself (Issue#961, PR#963).
- **Change**
- The entire repository is now formatted with Black code style. Additionally, a pipeline was added to check this for new PRs and commit hooks are provided for developers (PR#992).
- When the `PKG_NAME` environeent variable was not set in the Dockerfile, a clear error is now raised (Issue#995, PR#1010).
- **Bugfix**
- Running encrypted algorithms failed due to a bug in the proxy server (Issue#955, PR#1008).
- Node logs were not persisted properly. This has been fixed (Issue#993, PR#1009).

4.12.9 4.2.1

19 January 2024

- **Bugfix**
- Add back binary installation of `psycopg2` to support Postgres databases (PR#932).

4.12.10 4.2.0

18 January 2024

- **Security**
- Remove option to SSH into node and server containers. The configuration was not completely secure (advisory, commit).
- Prevent code injection into environment variables (advisory, commit).
- Prevent that user can accidentally upload non-encrypted input to the server for an encrypted collaboration. (advisory, commit).
- Prevent that usernames are findable in brute force attack due to a difference in response time when they exist versus when they don't exist (advisory, commit).
- Updated dependencies of jinja2, cryptography and Werkzeug. (PR#984).
- **Feature**
- Introduced the `v6 test` commands that will run the test algorithm `v6-diagnostics` (Issue#918, PR#930).
- Extended `v6 dev` commands with options to add extra configuration to the server and node configuration files. Also, added the `v6 server remove` command. (Issue#860, PR#930).
- **Change**
- Changed some log messages to a more appropriate log level (Issue#667)
- Improved message when node starts so as to make it clearer to users that the node has not yet authenticated (PR#957).
- Changed socket event `on_new_task` to also include the parent ID of the task that was created (PR#950).
- **Bugfix**
- Added check whether database labels are properly specified when creating a task (Issue#910, PR#932).
- Fix bug in creating task with VPN client image when it has `iptables-legacy` installed (Issue#966, PR#982).
- Add missing `email` argument from `client.user.create` function (Issue#837, PR#934).

4.12.11 4.1.3

19 December 2023

- **Bugfix**
- Server logs were not persisted properly (Issue#951, PR#953).
- Fixed validation of request to recover two-factor authentication secret (PR#941).
- Default roles were visible via GET `/role` but not via GET `/role/<id>` for users without global role view permission. Now they are visible via both (PR#948).

4.12.12 4.1.2

14 November 2023

- **Security**
- Improved check which algorithms are allowed - no longer trusting an algorithm with a *parent_id* by default (advisory, commit).

4.12.13 4.1.1

1 November 2023

- **Bugfix**
- Added OpenPyxl dependency to algorithm tools which is required to read Excel databases (PR#923).
- Explicitly define the resource on which sorting is done in the API. This prevents SQL errors when SQLAlchemy tries to sort on a column in a joined table (PR#925).
- Fixed retrieving column names for Excel databases (PR#924).

4.12.14 4.1.0

19 October 2023

- **Feature**
- Renamed CLI commands. The new commands are:
  - `vnode` → `v6 node`
  - `vserver` → `v6 server`
  - `vdev` → `v6 dev`

  The old commands will still be available until version 5.0 is released.
- Added CLI command `v6 algorithm create` which is a starting point for creating new algorithms (Issue#400, PR#904).
- Added `@database_connection(type_)` algorithm decorator. This enables algorithm developers to inject a database connection into their algorithm instead of a dataframe. The only type that currently is support is `omop`, which injects a `OHDSI/DatabaseConnection` object into your algorithm. (PR#902).
- Added endpoint */column* for the UI to get the column names of the database. This is achieved either by sharing column names by the node for file-based databases or by sending a task using the `basics` algorithm. The latter is now an allowed algorithm by default, unless the node is configured to not allow it. ((Issue#778, PR#908).
- Added `only_siblings` and `only_self` options to the `client.vpn.get_addresses` function. These options allow you to get the VPN addresses of only the siblings or only the node itself, respectively. This is useful for algorithms that need to communicate with other algorithms on the same node or with the node itself. (Issue#729, PR#901).

4.12.15 4.0.3

16 October 2023

- **Bugfix**
- Fix where custom Docker image for node was defined in config file but not used in practice (PR#896).
- Fixed getting VPN algorithm addresses from `AlgorithmClient` (PR#898).

4.12.16 4.0.2

9 October 2023

- **Bugfix**
- Fix socket connection from node to server due to faulty callback, which occurred when server was deployed. This bug was introduced in v4.0.1 (PR#892).

4.12.17 4.0.1

5 October 2023

- **Security**
- Updating dependencies `cryptography`, `gevent`, and `urllib3` to fix vulnerabilities (PR#889)
- **Bugfix**
- Fix node connection issues if server without constant JWT secret key is restarted (Issue#840, PR#866).
- Improved algorithm_client decorator with `@wraps` decorator. This fixes an issue with the data decorator in the AlgorithmMockClient (Issue#874, PR#882).
- Decoding the algorithm results and algorithm input has been made more robust, and input from `vserver import` is now properly encoded (Issue#836, PR#864).
- Improve error message if user forgot to specify `databases` when creating a task (Issue#854, PR#865).
- Fix data loading in AlgorithmMockClient (Issue#872, PR#881).

4.12.18 4.0.0

20 September 2023

- **Security**
- No longer using Python pickles for serialization and deserialization of algorithm results. Using JSON instead (CVE#CVE-2023-23930, commit).
- Not allowing resources to have an integer name (CVE#CVE-2023-28635, PR#744).
- Users allowed to view collaborations but not allowed to view tasks may be able to view them via `/api/collaboration/<id>/task` (CVE#CVE-2023-41882, PR#741).
- Users allowed to view tasks but not results may be able to view them via `/api/task?include=results` (CVE#CVE-2023-41882, PR#711).
- Deleting all linked tasks when a collaboration is deleted (CVE#CVE-2023-41881, PR#748).
- **Feature**

- A complete permission scope has been added at the collaboration level, allowing projects to assign one user to manage everything within that collaboration level without requiring global access (Issue#245, PR#711).
- Added decorators `@algorithm_client` and `@data()` to make the signatures and names of algorithm functions more flexible and also to allow for multiple databases (Issue#440, PR#652).
- Allow a single algorithm function to make use of multiple databases (Issue#804, PR#652, PR#807).
- Enforce pagination in the API to improve performance, and add a *sort* parameter for GET requests which yield multiple resources (Issue#392, PR#611).
- Share a node's database labels and types with the central server, so that the server can validate that these match between nodes and offer them as suggestions to the user when creating tasks (Issue#750, PR#751).
- `vnode new` now automatically retrieves information on e.g. whether the collaboration is encrypted, so that the user doesn't have to specify this information themselves (Issue#434, PR#739).
- Allow only unique names for organizations, collaborations, and nodes (Issue#242, PR#515).
- New function `client.task.wait_for_completion()` for the *AlgorithmClient* to allow waiting for subtasks to complete (Issue#651, PR#727).
- Improved validation of the input for all POST and PATCH requests using marshmallow schemas (Issue#76, PR#744).
- Added option `user_created` to filter tasks that have been directly created by a user and are thus not subtasks (Issue#583, PR#599).
- Users can now assign rules to other users that they don't have themselves if they do have higher permisions on the same resource (Issue#443, PR#781).
- **Change**
- Changed the API response structure: no longer returning as many linked resources for performance reasons (Issue#49, PR#709)
- The `result` endpoint has been renamed to `run` as this was a misnomer that concerns algorithm runs (Issue#436, PR#527), PR#620).
- Split the *vantage6-client* package: the Python user client is kept in this package, and a new *vantage6-algorithm-tools* PyPI package is created for the tools that help algorithm developers. These tools were part of the client package, but moving them reduces the sizes of both packages (Issue#662, PR#763)
- Removed environments *test*, *dev*, *prod*, *acc* and *application* from vantage6 servers and nodes as these were used little (Issue#260, PR#643)
- Harmonized the interfaces between the *AlgorithmClient* and the *MockClient* (Issue#669, PR#722)
- When users request resources where they are not allowed to see everything, they now get an unauthorized error instead of an incomplete or empty response (Issue#635, PR#711).
- Node checks the server's version and by default, it pulls a matching image instead of the latest image of it's major version (Issue#700, PR#706).
- `vserver-local` commands have been removed if they were not used within the docker images or the CLI (Issue#663, PR#728).
- The way in which RabbitMQ is started locally has been changed to make it easier to run RabbitMQ locally. Now, a user indicates with a configuration flag whether they expect RabbitMQ to be started locally (Issue#282, PR#795).
- The place in which server configuration files were stored on Linux has been changed fro `/etc/xdg` to `/etc/vantage6/` (Issue#269, PR#789).

- Backwards compatibility code that was present to make different v3.x versions compatible has been removed. Additionally, small improvements have been made that were not possible to do without breaking compatibility (Issue#454, PR#740, PR#758).
- **Bugfix**
- Remove wrong dot in the version for prereleases (PR#764).
- Users were not assigned any permissions if *vserver import* was run before the server had ever been started (Issue#634, PR#806).

4.12.19 3.11.1

11 September 2023

- **Bugfix**
- Setting up the host network for VPN did not work properly if the host had `iptables-legacy` installed rather than `iptables`. Now, the code has been made compatible with both (Issue#725, PR#802).

4.12.20 3.11.0

21 August 2023

- **Feature**
- A suite of *vdev* commands has been added to the CLI. These commands allow you to easily create a development environment for vantage6. The commands allow you to easily create a server configuration, add organizations and collaborations to it, and create the appropriate node configurations. Also, you can easily start, stop, and remove the network. (Issue#625, PR#624).
- User Interface can now be started from the CLI with *vserver start –with-ui* (Issue#730, PR#735).
- Added *created_at* and *finished_at* timestamps to tasks (Issue#621, PR#715).
- **Change**
- Help text for the CLI has been updated and the formatting has been improved (Issue#745, PR#791).
- With *vnode list*, the terms *online* and *offline* have been replaced by *running* and *not running*. This is more accurate, since a node may be unable to authenticate and thus be offline, but still be running. (Issue#733, PR#734).
- Some legacy code that no longer fulfilled a function has been removed from the endpoint to create tasks (Issue#742, PR#747).
- **Bugfix**
- In the docs, the example file to import server resources with *vserver import* was accidentally empty; now it contains example data. (PR#792).

4.12.21 3.10.4

27 June 2023

- **Change**
- Extended the AlgorithmMockClient so that algorithm developers may pass it organization id's and node id's (PR#737).
- **Bugfix**
- Speed up starting algorithm using VPN (Issue#681, PR#732).
- Updated VPN configurator Dockerfile so that VPN configuration works on Ubuntu 22 (Issue#724, PR#725).

4.12.22 3.10.3

20 June 2023

- **Bugfix**
- Fixed bug in copying the MockClient itself to pass it on to a child task (PR#723).

Note: Release 3.10.2 failed to be published to PyPI due to a gateway error, so that version was skipped.

4.12.23 3.10.1

19 June 2023

- **Bugfix**
- Fixed bug in setting organization_id for the AlgorithmClient (Issue#719, PR#720).

4.12.24 3.10.0

19 June 2023

- **Feature**
- There is a new implementation of a mock client, the `MockAlgorithmClient`. This client is an improved version of the old `ClientMockProtocol`. The new mock client now contains all the same functions as the regular client with the same signatures, and it returns the same data fields as those functions. Also, you may submit all supported data formats instead of just CSV files, and you may also submit pandas Dataframes directly (Issue#683, PR#702).
- **Change**
- Updated cryptography dependency from 39.0.1 to 41.0.0 (PR#707, PR#708).
- **Bugfix**
- A node's VPN IP address was previously only updated when a new task was started on that node. Instead, it is now updated properly on VPN connect/ disconnect (Issue#520, PR#704).

4.12.25 3.9.0

25 May 2023

- **Feature**
- Data sources may now be whitelisted by IP address, so that an algorithm may access those IP addresses to obtain data. This is achieved via a Squid proxy server (Issue#162, PR#626).
- There is a new configuration option to let algorithms access gpu's (Issue#597, PR#623).
- Added option to get VPN IP addresses and ports of just the children or just the parent of an algorithm that is running. These options may be used to simplify VPN communication between algorithms running on different nodes. In the AlgorithmClient, the functions `client.vpn.get_child_addresses()` and `client.vpn.get_parent_address()` have been added (PR#610).
- New option to print the full stack trace of algorithm errors. Note that this option may leak sensitive information if used carelessly. The option may be activated by setting `log_traceback=True` in the algorithm wrapper (Issue#675, PR#680).
- Configuration options to control the log levels of individual dependencies. This allows easier debugging when a certain dependency is causing issues (Issue#641, PR#642).
- **Change**
- Better error message for `vnode attach` when no nodes are running (Issue#606, PR#607).
- The number of characters of the task input printed to the logs is now limited to prevent flooding the logs with very long input (Issue#549, PR#550).
- Node proxy logs are now written to a separate log file. This makes the main node log more readable (Issue#546, PR#619).
- Update code in which the version is updated (PR#586).
- Finished standardizing docstrings - note that this was already partially done in earlier releases (Issue#255).
- Cleanup and moving of unused code and duplicate code (PR#571).
- It is now supported to run the release pipeline from `release/v<x.y.z>` branches (Issue#467, PR#488).
- Replaced deprecated `set-output` method in Github actions release pipeline (Issue#474, PR#601).
- **Bugfix**
- Fixed checking for newer images (node, server, and algorithms). Previously, the dates used were not sufficient to check if an image was newer. Now, we are also checking the image digest (Issue#507, PR#602).
- Users are prevented from posting socket events that are meant for nodes - note that nothing harmful could be done but it should not be possible nevertheless (Issue#615, PR#616).
- Fixed bug with detecting if database was a file as '/mnt/' was not properly prepended to the file path (PR#691).

4.12.26 3.8.8

11 May 2023

- **Bugfix**
  - Fixed a bug that prevented the node from shutting down properly (Issue#649, PR#677)
  - Fixed a bug where the node did not await the VPN client to be ready (Issue#656, PR#676)
  - Fixed database label logging (PR#664)
  - Fixed a bug were VPN messages to the originating node where not always sent/received (Issue#671, PR#673)
  - Fixed a bug where an exceptions is raised when the websocket connection was lost and a ping was attempted to be send (Issue#672, PR#674)
  - Fixed a formatting in CLI print statement (PR#661)
  - Fixed bug where '/mnt/' was erroneously prepended to non-file based databases (PR#658)
  - Fix in `autowrapper` for algorithms with CSV input (PR#655)
  - Fixed a bug in syncing tasks from the server to the node, when the node lost socket connection and then reconnected (Issue#654, PR#657)
  - Fix construction of database URI in `vserver files` (Issue#650, PR#659)

4.12.27 3.8.7

10 May 2023

- **Bugfix**
  - Socket did connect before Docker was initialized, resulting in an exception at startup (PR#644)

4.12.28 3.8.6

9 May 2023

- **Bugfix**
  - Fixed bug that resulted in broken algorithm networks when the socket connection was lost (PR#640, Issue#637)

4.12.29 3.8.3 - 3.8.5

25 April 2023 - 2 May 2023

- **Bugfix**
- Fixed bug where a missing container lead to a complete node crash (PR#628, PR#629, PR#632).
- Restored algorithm wrapper namespace for backward compatibility (PR#618)
- Prevent error with first socket ping on node startup by waiting a few seconds (PR#609)

4.12.30 3.8.2

22 march 2023

- **Feature**
- Location of the server configuration file in server shell script can now be specified as an environment variable (PR#604)
- **Change**
- Changed ping/pong mechanism over socket connection between server and nodes, as it did not function properly in combination with RabbitMQ. Now, the node pushes a ping and the server periodically checks if the node is still alive (PR#593)
- **Bugfix**
- For `vnode files`, take the new formatting of the databases in the node configuration file into account (PR#600)
- Fix bugs in new algorithm client where class attributes were improperly referred to (PR#596)
- Fixed broken links in Discord notification (PR#591)

4.12.31 3.8.1

8 march 2023

- **Bugfix**
- In 3.8.0, starting RabbitMQ for horizontal scaling caused a server crash due to a missing `kombu` dependency. This dependency was wrongly removed in updating all dependencies for python 3.10 (PR#585).

4.12.32 3.8.0

8 march 2023

- **Security**
- Refresh tokens are no longer indefinitely valid (CVE#CVE-2023-23929, commit).
- It was possible to obtain usernames by brute forcing the login since v3.3.0. This was due to a change where users got to see a message their account was blocked after N failed login attempts. Now, users get an email instead if their account is blocked (CVE#CVE-2022-39228, commit).
- Assigning existing users to a different organizations was possible. This may lead to unintended access: if a user from organization A is accidentally assigned to organization B, they will retain their permissions and therefore might be able to access resources they should not be allowed to access (CVE#CVE-2023-22738, commit).
- **Feature**
- Python version upgrade to 3.10 and many dependencies are upgraded (PR#513, Issue#251).
- Added `AlgorithmClient` which will replace `ContainerClient` in v4.0. For now, the new `AlgorithmClient` can be used by specifying `use_new_client=True` in the algorithm wrapper (PR#510, Issue#493).
- It is now possible to request some of the node configuration settings, e.g. which algorithms they allow to be run (PR#523, Issue#12).
- Added `auto_wrapper` which detects the data source types and reads the data accordingly. This removes the need to rebuild every algorithm for every data source type (PR#555, Issue#553).

- New endpoint added `/vpn/algorithm/addresses` for algorithms to obtain addresses for containers that are part of the same computation task (PR#501, Issue#499).
- Added the option to allow only allow certain organization and/or users to run tasks on your node. This can be done by using the `policies` configuration option. Note that the `allowed_images` option is now nested under the `policies` option (Issue#335, PR#556)
- **Change**
- Some changes have been made to the release pipeline (PR#519, PR#488, PR#500, Issue#485).
- Removed unused script to start the shell (PR#494).
- **Bugfix**
- Algorithm containers running on the same node could not communicate with each other through the VPN. This has been fixed (PR#532, Issue#336).

4.12.33 3.7.3

22 february 2023

- **Bugfix**
- A database commit in 3.7.2 was done on the wrong variable, this has been corrected (PR#547, Issue#534).
- Delete entries in the VPN port table after the algorithm has completed (PR#548).
- Limit number of characters of the task input printed to the logs (PR#550).

4.12.34 3.7.2

20 february 2023

- **Bugfix**
- In 3.7.1, some sessions were closed, but not all. Now, sessions are also terminated in the socketIO events (PR#543, Issue#534).
- Latest versions of VPN images were not automatically downloaded by node on VPN connection startup. This has been corrected (PR#542).

4.12.35 3.7.1

16 february 2023

- **Change**
- Some changes to the release pipeline.
- **Bugfix**
- `iptables` dependency was missing in the VPN client container (PR#533 Issue#518).
- Fixed a bug that did not close Postgres DB sessions, resulting in a dead server (PR#540, Issue#534).

4.12.36 3.7.0

25 january 2023

- **Feature**
- SSH tunnels are available on the node. This allows nodes to connect to other machines over SSH, thereby greatly expanding the options to connect databases and other services to the node, which before could only be made available to the algorithms if they were running on the same machine as the node (PR#461, Issue#162).
- For two-factor authentication, the information given to the authenticator app has been updated to include a clearer description of the server and username (PR#483, Issue#405).
- Added the option to run an algorithm without passing data to it using the CSV wrapper (PR#465)
- In the UI, when users are about to create a task, they will now be shown which nodes relevant to the task are offline (PR#97, Issue#96).
- **Change**
- The `docker` dependency is updated, so that `docker.pull()` now pulls the *default* tag if no tag is specified, instead of all tags (PR#481, Issue#473).
- If a node cannot authenticate to the server because the server cannot be found, the user now gets a clearer error message(PR#480, Issue#460).
- The default role 'Organization admin' has been updated: it now allows to create nodes for their own organization (PR#489).
- The release pipeline has been updated to 1) release to PyPi as last step (since that is irreversible), 2) create release branches, 3) improve the check on the version tag, and 4) update some soon-to-be-deprecated commands (PR#488.
- Not all nodes are alerted any more when a node comes online (PR#490).
- Added instructions to the UI on how to report bugs (PR#100, Issue#57).
- **Bugfix**
- Newer images were not automatically pulled from harbor on node or server startup. This has been fixed (PR#482, Issue#471).

4.12.37 3.6.1

12 january 2023

- **Feature**
- Algorithm containers can be killed from the client. This can be done for a specific task or it possible to kill all tasks running at a specific node (PR#417, Issue#167).
- Added a `status` field for an algorithm, that tracks if an algorithm has yet to start, is started, has finished, or has failed. In the latter case, it also indicates how/when the algorithm failed (PR#417).
- The UI has been connected to the socket, and gives messages about node and task status changes (UI PR#84, UI Issue #73). There are also new permissions for socket events on the server to authorize users to see events from their (or all) collaborations (PR#417).
- It is now possible to create tasks in the UI (UI version >3.6.0). Note that all tasks are then JSON serialized and you will not be able to run tasks in an encrypted collaboration (as that would require uploading a private key to a browser) (*PR#90*).

> **Warning:** If you want to run the UI Docker image, note that from this version onwards, you have to define the `SERVER_URL` and `API_PATH` environment variables (compared to just a `API_URL` before).

- There is a new multi-database wrapper that will forward a dictionary of all node databases and their paths to the algorithm. This allows you to use multiple databases in a single algorithm easily. (PR#424, Issue #398).
- New rules are now assigned automatically to the default root role. This ensures that rules that are added in a new version are assigned to system administrators, instead of them having to change the database (PR#456, Issue #442).
- There is a new command `vnode set-api-key` that facilitates putting your API key into the node configuration file (PR#428, Issue #259).
- Logging in the Python client has been improved: instead of all or nothing, log level is now settable to one of debug, info, warn, error, critical (PR#453, Issue #340).
- When there is an error in the VPN server configuration, the user receives clearer error messages than before (PR#444, Issue #278).
- **Change**
- The node status (online/offline) is now checked periodically over the socket connection via a ping/pong construction. This is an improvement over the older version where a node's status was changed only when it connected or disconnected (PR#450, Issue #40).

> **Warning:** If a server upgrades to 3.6.1, the nodes should also be upgraded. Otherwise, the node status will be incorrect and the logs will show errors periodically with each attempted ping/pong.

- It is no longer possible for any user to change the username of another user, as this would be confusing for that user when logging in (PR#433, Issue #396).
- The server has shorter log messages when someone calls a non-existing route. The resulting 404 exception is no longer logged (PR#452, Issue #393).
- Removed old, unused scripts to start a node (PR#464).
- **Bugfix**
- Node was unable to pull images from Docker Hub; this has been corrected. (PR#432, Issue#422).
- File-based database extensions were always converted to `.csv` when they were mounted to a node. Now, files keep their original file extensions (PR#426, Issue #397).
- When a node configuration defined a wrong VPN subnet, the VPN connection didn't work but this was not detected until VPN was used. Now, the user is alerted immediately and VPN is turned off (PR#444).
- If a user tries to write a node or server config file to a non-existing directory, they are now getting a clear error message instead of an incorrect one (PR#455, Issue #1)
- There was a circular import in the infrastructure code, which has now been resolved (PR#451, Issue #53).
- In PATCH `/user`, it was not possible to set some fields (e.g. `firstname`) to an empty string if there was a value before. (PR#439, Issue #334).

Note: Release 3.6.0 was skipped due to an issue in the release process.

4.12.38 3.5.2

30 november 2022

- **Bugfix**
- Fix for automatic addition of column. This failed in some SQL dialects because reserved keywords (i.e. 'user' for PostgresQL) were not escaped (PR#415)
- Correct installation order for uWSGI in node and server docker file (PR#414)

4.12.39 3.5.1

30 november 2022

- **Bugfix**
- Backwards compatibility for which organization initiated a task between v3.0-3.4 and v3.5 (PR#412)
- Fixed VPN client container. Entry script was not executable in Github pipelines (PR#413)

4.12.40 3.5.0

30 november 2022

> **Warning:** When upgrading to 3.5.0, you might need to add the **otp_secret** column to the **user** table manually in the database. This may be avoided by upgrading to 3.5.2.

- **Feature**
- Multi-factor authentication via TOTP has been added. Admins can enforce that all users enable MFA (PR#376, Issue#355).
- You can now request all tasks assigned by a given user (PR#326, Issue#43).
- The server support email is now settable in the configuration file, used to be fixed at `support@vantage6.ai` (PR#330, Issue#319).
- When pickles are used, more task info is shown in the node logs (PR#366, Issue#171).
- **Change**
- The `harbor2.vantag6.ai/infrastructure/algorithm-base:[TAG]` is tagged with the vantage6-client version that is already in the image (PR#389, Issue#233).
- The infrastructure base image has been updated to improve build time (PR#406, Issue#250).

4.12.41 3.4.2

3 november 2022

- **Bugfix**
- Fixed a bug in the local proxy server which made algorithm containers crash in case the *client.create_new_task* method was used (PR#382).
- Fixed a bug where the node crashed when a non existing image was sent in a task (PR#375).

4.12.42 3.4.0 & 3.4.1

25 oktober 2022

- **Feature**
- Add columns to the SQL database on startup (PR#365, ISSUE#364). This simpifies the upgrading proces when a new column is added in the new release, as you do no longer need to manually add columns. When downgrading the columns will **not** be deleted.
- Docker wrapper for Parquet files (PR#361, ISSUE#337). Parquet provides a way to store tabular data with the datatypes included which is an advantage over CSV.
- When the node starts, or when the client is verbose initialized a banner to cite the vantage6 project is added (PR#359, ISSUE#356).
- In the client a waiting for results method is added (PR#325, ISSUE#8). Which allows you to automatically poll for results by using `client.wait_for_results(...)`, for more info see `help(client.wait_for_results)`.
- Added Github releases (PR#358, ISSUE#357).
- Added option to filter GET `/role` by user id in the Python client (PR#328, ISSUE#213). E.g.: `client.role.list(user=...)`.
- In release process, build and release images for both ARM and x86 architecture.
- **Change**
- Unused code removed from the Makefile (PR#324, ISSUE#284).
- Pandas version is frozen to version 1.3.5 (PR#363 , ISSUE#266).
- **Bugfix**
- Improve checks for non-existing resources in unittests (PR#320, ISSUE#265). Flask did not support negative ints, so the tests passed due to another 404 response.
- `client.node.list` does no longer filter by offline nodes (PR#321, ISSUE#279).

Note: 3.4.1 is a rebuild from 3.4.0 in which the all dependencies are fixed, as the build led to a broken server image.

4.12.43 3.3.7

- **Bugfix**
- The function `client.util.change_my_password()` was updated (Issue #333)

4.12.44 3.3.6

- **Bugfix**
- Temporary fix for a bug that prevents the master container from creating tasks in an encrypted collaboration. This temporary fix disables the parallel encryption module in the local proxy. This functionality will be restored in a future release.

Note: This version is also the first version where the User Interface is available in the right version. From this point onwards, the user interface changes will also be part of the release notes.

4.12.45 3.3.5

- **Feature**
- The release pipeline has been expanded to automatically push new Docker images of node/server to the harbor2 service.
- **Bugfix**
- The VPN IP address for a node was not saved by the server using the PATCH `/node` endpoint, while this functionality is required to use the VPN

Note: Note that 3.3.4 was only released on PyPi and that version is identical to 3.3.5. That version was otherwise skipped due to a temporary mistake in the release pipeline.

4.12.46 3.3.3

- **Bugfix**
- Token refresh was broken for both users and nodes. (Issue#306, PR#307)
- Local proxy encrpytion was broken. This prefented algorithms from creating sub tasks when encryption was enabled. (Issue#305, PR#308)

4.12.47 3.3.2

- **Bugfix**
- `vpn_client_image` and `network_config_image` are settable through the node configuration file. (PR#301, Issue#294)
- The option `--all` from `vnode stop` did not stop the node gracefully. This has been fixed. It is possible to force the nodes to quit by using the `--force` flag. (PR#300, Issue#298)
- Nodes using a slow internet connection (high ping) had issues with connecting to the websocket channel. (PR#299, Issue#297)

4.12.48 3.3.1

- **Bugfix**
- Fixed faulty error status codes from the `/collaboration` endpoint (PR#287).
- *Default* roles are always returned from the `/role` endpoint. This fixes the error when a user was assigned a *default* role but could not reach anything (as it could not view its own role) (PR#286).
- Performance upgrade in the `/organization` endpoint. This caused long delays when retrieving organization information when the organization has many tasks (PR#288).
- Organization admins are no longer allowed to create and delete nodes as these should be managed at collaboration level. Therefore, the collaboration admin rules have been extended to include create and delete nodes rules (PR#289).
- Fixed some issues that made `3.3.0` incompatible with `3.3.1` (Issue#285).

4.12.49 3.3.0

- **Feature**
- Login requirements have been updated. Passwords are now required to have sufficient complexity (8+ characters, and at least 1 uppercase, 1 lowercase, 1 digit, 1 special character). Also, after 5 failed login attempts, a user account is blocked for 15 minutes (these defaults can be changed in a server config file).
- Added endpoint `/password/change` to allow users to change their password using their current password as authentication. It is no longer possible to change passwords via `client.user.update()` or via a PATCH `/user/{id}` request.
- Added the default roles 'viewer', 'researcher', 'organization admin' and 'collaboration admin' to newly created servers. These roles may be assigned to users of any organization, and should help users with proper permission assignment.
- Added option to filter get all roles for a specific user id in the GET `/role` endpoint.
- RabbitMQ has support for multiple servers when using `vserver start`. It already had support for multiple servers when deploying via a Docker compose file.
- When exiting server logs or node logs with Ctrl+C, there is now an additional message alerting the user that the server/node is still running in the background and how they may stop them.
- **Change**
- Node proxy server has been updated
- Updated PyJWT and related dependencies for improved JWT security.
- When nodes are trying to use a wrong API key to authenticate, they now receive a clear message in the node logs and the node exits immediately.
- When using `vserver import`, API keys must now be provided for the nodes you create.
- Moved all swagger API docs from YAML files into the code. Also, corrected errors in them.
- API keys are created with UUID4 instead of UUID1. This prevents that UUIDs created milliseconds apart are not too similar.
- Rules for users to edit tasks were never used and have therefore been deleted.
- **Bugfix**
- In the Python client, `client.organization.list()` now shows pagination metadata by default, which is consistent all other `list()` statements.
- When not providing an API key in `vnode new`, there used to be an unclear error message. Now, we allow specifying an API key later and provide a clearer error message for any other keys with inadequate values.
- It is now possible to provide a name when creating a name, both via the Python client as via the server.
- A GET `/role` request crashed if parameter `organization_id` was defined but not `include_root`. This has been resolved.
- Users received an 'unexpected error' when performing a GET `/collaboration?organization_id=<id>` request and they didn't have global collaboration view permission. This was fixed.
- GET `/role/<id>` didn't give an error if a role didn't exist. Now it does.

4.12.50 3.2.0

- **Feature**
- Horizontal scaling for the vantage6-server instance by adding support for RabbitMQ.
- It is now possible to connect other docker containers to the private algorithm network. This enables you to attach services to the algorithm network using the `docker_services` setting.
- Many additional select and filter options on API endpoints, see swagger docs endpoint (`/apidocs`). The new options have also been added to the Python client.
- Users are now always able to view their own data
- Usernames can be changed though the API
- **Bugfix**
- (Confusing) SQL errors are no longer returned from the API.
- Clearer error message when an organization has multiple nodes for a single collaboration.
- Node no longer tries to connect to the VPN if it has no `vpn_subnet` setting in its configuration file.
- Fix the VPN configuration file renewal
- Superusers are no longer able to post tasks to collaborations its organization does not participate in. Note that superusers were never able to view the results of such tasks.
- It is no longer possible to post tasks to organization which do not have a registered node attach to the collaboration.
- The `vnode create-private-key` command no longer crashes if the ssh directory does not exist.
- The client no longer logs the password
- The version of the `alpine` docker image (that is used to set up algorithm runs with VPN) was fixed. This prevents that many versions of this image are downloaded by the node.
- Improved reading of username and password from docker registry, which can be capitalized differently depending on the docker version.
- Fix error with multiple-database feature, where default is now used if specific database is not found

4.12.51 3.1.0

- **Feature**
- Algorithm-to-algorithm communication can now take place over multiple ports, which the algorithm developer can specify in the Dockerfile. Labels can be assigned to each port, facilitating communication over multiple channels.
- Multi-database support for nodes. It is now also possible to assign multiple data sources to a single node in Petronas; this was already available in Harukas 2.2.0. The user can request a specific data source by supplying the *database* argument when creating a task.
- The CLI commands `vserver new` and `vnode new` have been extended to facilitate configuration of the VPN server.
- Filter options for the client have been extended.
- Roles can no longer be used across organizations (except for roles in the default organization)
- Added `vnode remove` command to uninstall a node. The command removes the resources attached to a node installation (configuration files, log files, docker volumes etc).

- Added option to specify configuration file path when running `vnode create-private-key`.
- **Bugfix**
- Fixed swagger docs
- Improved error message if docker is not running when a node is started
- Improved error message for `vserver version` and `vnode version` if no servers or nodes are running
- Patching user failed if users had zero roles - this has been fixed.
- Creating roles was not possible for a user who had permission to create roles only for their own organization - this has been corrected.

4.12.52 3.0.0

- **Feature**
- Direct algorithm-to-algorithm communication has been added. Via a VPN connection, algorithms can exchange information with one another.
- Pagination is added. Metadata is provided in the headers by default. It is also possible to include them in the output body by supplying an additional parameter`include=metadata`. Parameters `page` and `per_page` can be used to paginate. The following endpoints are enabled:
  - GET `/result`
  - GET `/collaboration`
  - GET `/collaboration/{id}/organization`
  - GET `/collaboration/{id}/node`
  - GET `/collaboration/{id}/task`
  - GET `/organization`
  - GET `/role`
  - GET `/role/{id}/rule`
  - GET `/rule`
  - GET `/task`
  - GET `/task/{id}/result`
  - GET `/node`
- API keys are encrypted in the database
- Users cannot shrink their own permissions by accident
- Give node permission to update public key
- Dependency updates
- **Bugfix**
- Fixed database connection issues
- Don't allow users to be assigned to non-existing organizations by root
- Fix node status when node is stopped and immediately started up
- Check if node names are allowed docker names

4.12.53 2.3.0 - 2.3.4

- **Feature**
- Allows for horizontal scaling of the server instance by adding support for RabbitMQ. Note that this has not been released for version 3(!)
- **Bugfix**
- Performance improvements on the `/organization` endpoint

4.12.54 2.2.0

- **Feature**
- Multi-database support for nodes. It is now possible to assign multiple data sources to a single node. The user can request a specific data source by supplying the *database* argument when creating a task.
- The mailserver now supports TLS and SSL options
- **Bugfix**
- Nodes are now disconnected more gracefully. This fixes the issue that nodes appear offline while they are in fact online
- Fixed a bug that prevented deleting a node from the collaboration
- A role is now allowed to have zero rules
- Some http error messages have improved
- Organization fields can now be set to an empty string

4.12.55 2.1.2 & 2.1.3

- **Bugfix**
- Changes to the way the application interacts with the database. Solves the issue of unexpected disconnects from the DB and thereby freezing the application.

4.12.56 2.1.1

- **Bugfix**
- Updating the country field in an organization works again\
- The `client.result.list(...)` broke when it was not able to deserialize one of the in- or outputs.

4.12.57 2.1.0

- **Feature**
- Custom algorithm environment variables can be set using the `algorithm_env` key in the configuration file. See this Github issue.
- Support for non-file-based databases on the node. See this Github issue.
- Added flag `--attach` to the `vserver start` and `vnode start` command. This directly attaches the log to the console.
- Auto updating the node and server instance is now limited to the major version. See this Github issue.
  - e.g. if you've installed the Trolltunga version of the CLI you will always get the Trolltunga version of the node and server.
  - Infrastructure images are now tagged using their version major. (e.g. `trolltunga` or `harukas`)
  - It is still possible to use intermediate versions by specifying the `--image` option when starting the node or server. (e.g. `vserver start --image harbor.vantage6.ai/infrastructure/server:2.0.0.post1`)
- **Bugfix**
- Fixed issue where node crashed if the database did not exist on startup. See this Github issue.

4.12.58 2.0.0.post1

- **Bugfix**
- Fixed a bug that prevented the usage of secured registry algorithms

4.12.59 2.0.0

- **Feature**
- Role/rule based access control
  - Roles consist of a bundle of rules. Rules profided access to certain API endpoints at the server.
  - By default 3 roles are created: 1) Container, 2) Node, 3) Root. The root role is assigned to the root user on the first run. The root user can assign rules and roles from there.
- Major update on the *python*-client. The client also contains management tools for the server (i.e. to creating users, organizations and managing permissions. The client can be imported from `from vantage6.client import Client` .
- You can use the agrument `verbose` on the client to output status messages. This is usefull for example when working with Jupyter notebooks.
- Added CLI `vserver version` , `vnode version` , `vserver-local version` and `vnode-local version` commands to report the version of the node or server they are running
- The logging contains more information about the current setup, and refers to this documentation and our Discourd channel
- **Bugfix**
- Issue with the DB connection. Session management is updated. Error still occurs from time to time but can be reset by using the endpoint `/health/fix` . This will be patched in a newer version.

4.12.60 1.2.3

- **Feature**
- The node is now compatible with the Harbor v2.0 API

4.12.61 1.2.2

- **Bug fixes**
- Fixed a bug that ignored the `--system` flag from `vnode start`
- Logging output muted when the `--config` option is used in `vnode start`
- Fixed config folder mounting point when the option `--config` option is used in `vnode start`

4.12.62 1.2.1

- **Bug fixes**
- starting the server for the first time resulted in a crash as the root user was not supplied with an email address.
- Algorithm containers could still access the internet through their host. This has been patched.

4.12.63 1.2.0

- **Features**
- Cross language serialization. Enabling algorithm developers to write algorithms that are not language dependent.
- Reset password is added to the API. For this purpose two endpoints have been added: `/recover/lost`and `recover/reset` . The server config file needs to extended to be connected to a mail-server in order to make this work.
- User table in the database is extended to contain an email address which is mandatory.
- **Bug fixes**
- Collaboration name needs to be unique
- API consistency and bug fixes:
  - GET `organization` was missing domain key
  - PATCH `/organization` could not patch domain
  - GET `/collaboration/{id}/node` has been made consistent with `/node`
  - GET `/collaboration/{id}/organization` has been made consistent with `/organization`
  - PATCH `/user` root-user was not able to update users
  - DELETE `/user` root-user was not able to delete users
  - GET `/task` null values are now consistent: `[]` is replaced by `null`
  - POST, PATCH, DELETE `/node` root-user was not able to perform these actions
  - GET `/node/{id}/task` output is made consistent with the
- **other**
- `questionairy` dependency is updated to 1.5.2

- `vantage6-toolkit` repository has been merged with the `vantage6-client` as they were very tight coupled.

4.12.64 1.1.0

- **Features**
- new command `vnode clean` to clean up temporary docker volumes that are no longer used
- Version of the individual packages are printed in the console on startup
- Custom task and log directories can be set in the configuration file
- Improved **CLI** messages
- Docker images are only pulled if the remote version is newer. This applies both to the node/server image and the algorithm images
- Client class names have been simplified (`UserClientProtocol` -> `Client`)
- **Bug fixes**
- Removed defective websocket watchdog. There still might be disconnection issues from time to time.

4.12.65 1.0.0

- **Updated Command Line Interface (CLI)**
- The commands `vnode list` , `vnode start` and the new command`vnode attach` are aimed to work with multiple nodes at a single machine.
- System and user-directories can be used to store configurations by using the `--user/--system` options. The node stores them by default at user level, and the server at system level.
- Current status (online/offline) of the nodes can be seen using `vnode list` , which also reports which environments are available per configuration.
- Developer container has been added which can inject the container with the source. `vnode start --develop [source]`. Note that this Docker image needs to be build in advance from the `development.Dockerfile` and tag `devcon`.
- `vnode config_file` has been replaced by `vnode files` which not only outputs the config file location but also the database and log file location.
- **New database model**
- Improved relations between models, and with that, an update of the Python API.
- Input for the tasks is now stored in the result table. This was required as the input is encrypted individually for each organization (end-to-end encryption (E2EE) between organizations).
- The `Organization` model has been extended with the `public_key` (String) field. This field contains the public key from each organization, which is used by the E2EE module.
- The `Collaboration` model has been extended with the `encrypted` (Boolean) field which keeps track if all messages (tasks, results) need to be E2EE for this specific collaboration.
- The `Task` keeps track of the initiator (organization) of the organization. This is required to encrypt the results for the initiator.
- **End to end encryption**
- All messages between all organizations are by default be encrypted.

- Each node requires the private key of the organization as it needs to be able to decrypt incoming messages. The private key should be specified in the configuration file using the `private_key` label.
- In case no private key is specified, the node generates a new key an uploads the public key to the server.
- If a node starts (using `vnode start`), it always checks if the `public_key` on the server matches the private key the node is currently using.
- In case your organization has multiple nodes running they should all point to the same private key.
- Users have to encrypt the input and decrypt the output, which can be simplified by using our client `vantage6.client.Client` __ for Python __ or `vtg::Client` __ for R.
- Algorithms are not concerned about encryption as this is handled at node level.
- **Algorithm container isolation**
- Containers have no longer an internet connection, but are connected to a private docker network.
- Master containers can access the central server through a local proxy server which is both connected to the private docker network as the outside world. This proxy server also takes care of the encryption of the messages from the algorithms for the intended receiving organization.
- In case a single machine hosts multiple nodes, each node is attached to its own private Docker network.
- **Temporary Volumes**
- Each algorithm mounts temporary volume, which is linked to the node and the `job_id` of the task
- The mounting target is specified in an environment variable `TEMPORARY_FOLDER`. The algorithm can write anything to this directory.
- These volumes need to be cleaned manually. (`docker rm VOLUME_NAME`)
- Successive algorithms only have access to the volume if they share the same `job_id` . Each time a **user** creates a task, a new `job_id` is issued. If you need to share information between containers, you need to do this through a master container. If a central container creates a task, all child tasks will get the same `job_id`.
- **RESTful API**
- **All RESTful API output is HATEOS formatted.**
  (wiki**)**
- **Local Proxy Server**
- Algorithm containers no longer receive an internet connection. They can only communicate with the central server through a local proxy service.
- It handles encryption for certain endpoints (i.e. `/task`, the input or `/result` the results)
- **Dockerized the Node**
- All node code is run from a Docker container. Build versions can be found at our Docker repository: `harbor.distributedlearning.ai/infrastructure/node` . Specific version can be pulled using tags.
- For each running node, a Docker volume is created in which the data, input and output is stored. The name of the Docker volume is: `vantage-NODE_NAME-vol` . This volume is shared with all incoming algorithm containers.
- Each node is attached to the public network and a private network: `vantage-NODE_NAME-net`.

4.13 Partners

Our community is open to everyone. The following people and organizations made a significant contribution to the design and implementation of vantage6.

- Anja van Gestel
- Bart van Beusekom
- Frank Martin
- Hasan Alradhi
- Melle Sieswerda
- Gijs Geleijnse

- Djura Smits
- Lourens Veen

- Johan van Soest

Would you like to contribute? Check out *how to contribute!* Find and chat with us via the Discord chat!

PYTHON MODULE INDEX

V

INDEX

Symbols

D

E

F

K

L

M

N

O

P

R

S

T

U

V

W