DIGITAL TWINS STORAGE AND APPLICATION SERVICE HUB (TWINSTASH)

S. Haufe*, M. Bäßler*, C. Pätzold†, M. Tchorzewski†, H. Meyer‡, E. Arts‡, A. Kamtsiuris‡
* Institute of Software Methods for Product Virtualization, German Aerospace Center, Dresden
† Flight Experiments, German Aerospace Center, Braunschweig
‡ Institute of Maintenance, Repair and Overhaul, German Aerospace Center, Hamburg

Abstract

The *digital twins storage and application service hub*, in short *twinstash*, is a software system aiming to provide a basis for Digital Twins of DLR research aircraft. Initialized in the context of the *DigTwin* project and its successor project *DigECAT*, its current stable prototype supports the upload, download, and search for flight sensor data and metadata both programmatically via python and graphically via a browser-based user interface. The latter additionally allows to quickly and intuitively navigate through the data. It provides rich possibilities to visualize and filter flight trajectories as well as sensor data. In this paper, we give an in-depth view on the twinstash software system. We provide details on its architecture, continuous integration and deployment, authentication mechanisms, data model, search, main features of its graphical user interface, and its python client.

Keywords

Digital Twin; ISTAR; ATRA; CODE; HALO; twinstash; data management; web service; visualization

1. INTRODUCTION

The foundation of every Digital Twin is some well-architected, sound, and scalable software running on robust and reliable hardware. For DLR research aircraft, the *digital twins storage and application service hub*, in short *twinstash*, aims to provide exactly that. It has been initialized by the DLR Institute of Software Methods for Product Virtualization in the context of the *DigTwin* project [1] and its successor project *DigECAT*. Since then, its continuous development by a small team resulted in a stable prototype which already provides quite some useful components of a Digital Twin for research aircraft. It supports the upload, download, and search for flight sensor data and metadata such as departure airport/time, arrival airport/time, aircraft, ICAO Aircraft ID, aircraft registration, and flight trajectory. One way to access all data and functionality utilizes python and is therefore easily usable in fully automated workflows. Another way is twinstashs browser-based graphical user interface which allows to navigate through the data by project, time, aircraft, or by dedicated search queries. Beyond that it provides rich possibilities to display flight trajectories in 2D and 3D to visualize and filter flight sensor data. A fully functional Digital Twin, however, which provides simulations, predictions, real-time storage and processing is quite some time in the future. While an additional publication at DLRK 2022 [2] provides the broad context on DLR research aircraft and the utilization of twinstash from an application point of view, this paper focuses completely on the details of the twinstash software system itself. We will take a look at its architecture, authentication mechanisms, continuous integration and delivery pipelines, deployment, data model, and main features based on its current version 1.33.

2. ARCHITECTURE

The architecture of twinstash, depicted in Figure 1, follows the pattern of a web service and hence embodies a service-oriented architecture for distributed software systems. We utilize a backend which is composed of different software systems doing the main work like authentication, file serving, and data handling. We furthermore offer several light frontends which provide users the option to programmatically or graphically interact with the backend. Frontends and backend run on different machines and are communicating via a REST[1] API[2].

2.1. REST API

An *application programming interface*, in short API, allows computer programs to communicate with each other. The principle of *Representational State Transfer*, in short REST, imposes certain constraints on how data can be identified and manipulated through an API. A REST API allows computer programs to communicate via HTTP requests sent through the

[1] https://en.wikipedia.org/wiki/Representational_state_transfer
[2] https://en.wikipedia.org/wiki/API

doi: 10.25967/570104

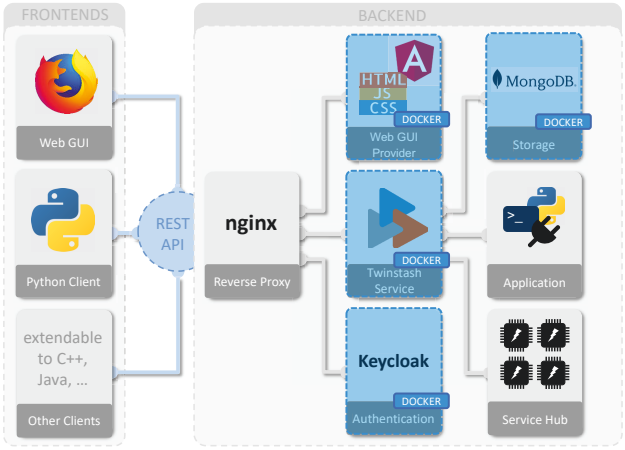

FIG 1. Web service architecture with frontends (left) and backend using Docker (right).

internet. HTTP requests are based on formatted strings and hence not bound to a certain programming language, implying that each involved computer program may be written in a different language. In our setting, the REST API only accepts TLS-encrypted HTTP requests, ensuring that no communication may be eavesdropped by third parties.

Our REST API provides the usual methods to download, upload, manipulate, and erase data. The endpoint `search` offers extensive search capabilities for data stored in twinstash. The REST API further provides all files which are necessary for rendering the graphical user interface in a web browser and some additional endpoints for authentication.

2.2. Frontends

Currently, we provide two frontends for twinstash, a *web GUI* and a *python client*.

The web GUI runs in any modern web browser. The user browses to the twinstash url[3] (which is currently available only within the DLR intranet). Approved DLR users may login with their respective DLR credentials (Section 2.4.1). They are then provided with browsing, searching, and visualization capabilities. We provide further details on the web GUI in Section 5.

The python client comes as a python package which is centrally deployed and can hence easily be included in the automated build process of any python software using twinstash. It allows to conveniently access all twinstash functionality through python functions and therewith shields the user from details of HTTP communication. Authentication is realized via API keys, we provide further details about that in Section 2.4.2. Two instances of python software which are utilizing the python client are presented in [2].

[3] https://stash.dlr.de

Similar to the python client, further clients for twinstash may easily be implemented, e.g. for languages such as Java or C++.

2.3. Backend

As 'gate keeper' of the backend, we employ an NGINX[4] service as reverse proxy. Its purpose is the termination of TLS encryption on HTTP requests and their delegation to and from three subservices of twinstash: the *web GUI provider*, the *Keycloak*, and the *twinstash service*. We dedicate Section 3 to details on how we build and host the backend using GitLab[5] and Docker[6].

2.3.1. Web GUI Provider

Our web GUI provider serves HTML, JavaScript, CSS, and image files which can be interpreted by each modern web browser to render the web GUI on the client side. We use Angular[7] for developing the web GUI. It provides high-level features to increase the efficiency in GUI development such as the separation of html and business logic, components for encapsulation, typescript for compile-time error handling, and efficient routing between subpages. Angular generates all the HTML, JavaScript, and CSS files needed by the browser.

2.3.2. Keycloak

The Keycloak[8] is an Open Source Identity and Access Management which handles all authentication needs of the backend according to the OpenID Connect layer on top of the OAuth 2.0 protocol[9]. We dedicate Section 2.4 to more details on the twinstash authentication flow.

2.3.3. Twinstash Service

The twinstash service provides the core functionality of twinstash. It stores data according to defined integrity measures, allows to search that data for retrieval in small slices, and provides functionality for its manipulation. It is implemented in python and uses the Flask[10] framework for its REST API. The data is persisted in an instance of MongoDB[11] [3], a NoSQL Database Management System suited for Big Data applications. The architecture of the twinstash service allows scaling in order to serve growing demands towards a service hub by utilizing computation clusters or a cloud. External applications may be connected to the twinstash service either as microservice or native python code. For example, the two instances of python software presented in [2] which are currently

[4] https://www.nginx.com/
[5] https://about.gitlab.com/
[6] https://www.docker.com/
[7] https://angular.io/
[8] https://www.keycloak.org/
[9] https://openid.net/connect/
[10] https://flask.palletsprojects.com/en/2.2.x/
[11] https://www.mongodb.com

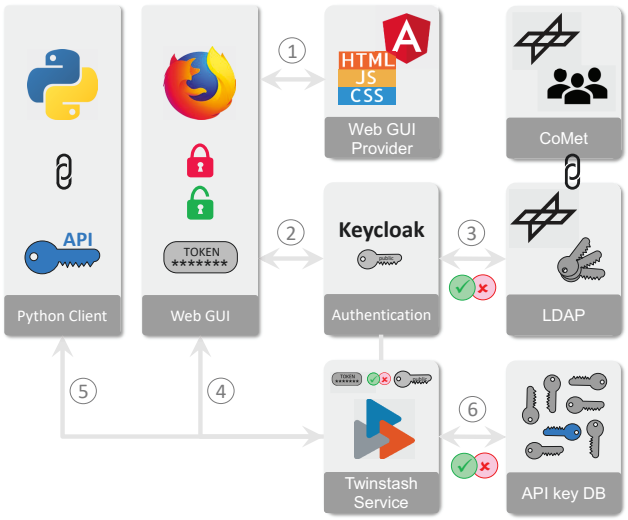

FIG 2. Authentication flow via credentials (steps ① to ④) and via API key (steps ④ to ⑥).

using the twinstash python client may be integrated as backend applications in a later stage of twinstash development.

2.4. Authentication

Some of the data in twinstash is subject to secrecy due to non-disclosure agreements with third parties. In order to grant access to dedicated people only, we need an authentication mechanism. To make authentication as simple as possible for the user, we decided to employ the established DLR approach, allowing users to login via their DLR credentials. An administrator maintains twinstash access restrictions by editing a DLR *CoMet* group. Due to authentication, twinstash knows which username performs each request and therewith annotates respective changes with it. The known username will also allow us to implement user-specific privileges in a later stage of twinstash development.

Authentication via DLR credentials cannot be used by the python client, as this would either force the user to provide his credentials on each run of a program (which requires manual interaction and thus renders automation impossible) or to put his credentials in a file (which is not secure). We therefore implemented a second authentication flow via API keys.

Both twinstash authentication flows are depicted in Figure 2. The flow via DLR credentials is shown in steps ① to ④ and the flow via API key in steps ④ to ⑥, we explain the details in the following two sections.

2.4.1. Authentication via DLR Credentials

When the user browses to the twinstash url[12], the browser requests all files which are necessary to render the twinstash web GUI (①). The requested files also contain JavaScript code which is run in the browser and detects that the user is not authenticated yet. The browser redirects to a login form provided by the Keycloak which asks for username and password (② →). The Keycloak passes the credentials to the DLR LDAP server asking for two things (③):

- Are the credentials correct?
- Does the username belong to the DLR *CoMet* group which holds all users who were approved for twinstash access?

If the answer to one of these questions is 'No', Keycloak aborts the login attempt by showing an error message in the login form (② ←). Otherwise, it provides the browser with a signed access JSON Web Token[13] (② ←) which contains the username. The browser now appends this token to each request sent to the twinstash service (④). To prevent fraud, e.g. by a changed username, the twinstash service has to verify that the presented access token originates unaltered from the trusted Keycloak. This can be done by using Keycloaks public key. The twinstash service retrieves the public key only once and performs the verification offline without time-consuming additional requests to the Keycloak.

2.4.2. Authentication via API Key

Since the authentication via API key requires the generation of an API key via the web GUI, the above described login flow has to be successfully performed beforehand once.

After successful login via DLR credentials the user is authenticated in the web GUI and can browse to a page which allows the creation of an API key by clicking a button. This triggers a request to route `generate_api_key` (④ →). The twinstash service retrieves the username from the provided access token and creates an API key for that username in a dedicated API key database (⑥). It then returns the key as result of the request (④ ←). The key is shown in the web GUI where the user can retrieve it.

Now the user can use the twinstash python client along with his API key. On each request, the twinstash python client passes this API key along (⑤ →). The twinstash service verifies that the given API key is known to its API key database (⑥). If not, the request is answered with HTTP code 401 Unauthorized (⑤ ←). Otherwise, the twinstash service retrieves the username for the given key from its API key database and answers the request according to its implemented functionality (⑤ ←).

3. RUNNING TWINSTASH

In the following sections are described our automated pipelines to reliably build, test, deliver, and host twinstash.

[12] `https://stash.dlr.de`

[13] `https://en.wikipedia.org/wiki/JSON_Web_Token`

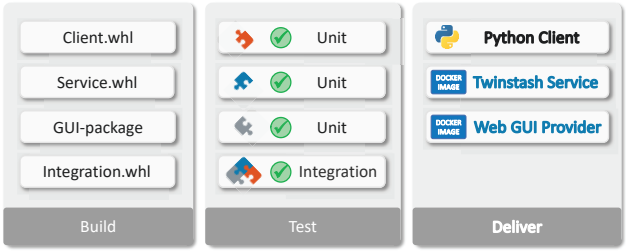

FIG 3. Continuous integration and delivery pipeline in GitLab which automatically builds, tests, and delivers twinstash.

3.1. Continuous Integration (CI)

For the development of every modern software, unit tests and integration tests have become indispensable. While unit tests focus on small software parts like functions and classes, integration tests ensure the overall correct functionality of the software system. Tests should execute often and hence be triggered effortlessly to ensure a high quality of the complete code basis. Triggering tests automatically on modified code has been framed as *Continuous Integration*.

In twinstash, we utilize GitLab to implement the continuous integration pipeline depicted in Figure 3 (stages `Build` and `Test`). We have three separate code bases: a python code base for the *service*, a python code base for the *client* and an angular code base for the *GUI*. While developing code in any of those code bases, we write unit tests for the developed code alongside. Those unit tests function in isolation, meaning that e.g. the tests for the service do not use the code for the client and vice versa. To ensure the correct interaction of the client with the service, we implement integration tests in an additional python code base named *integration*. All tests can be run locally while developing by the mere push of a button.

After pushing the local code changes to a remote GitLab server, it first automatically triggers a build of all code bases (stage `Build` in Figure 3). If that step was successful, GitLab runs all unit and integration tests (stage `Test` in Figure 3). For python the tests are executed in all supported versions from 3.7 to 3.11. To test the Angular code, the Karma-Framework is used.

3.2. Continuous Delivery (CD)

Given that the tests run without error, pushing a version-tagged commit to the GitLab server results in triggering the delivery of the built and tested twinstash system (stage `Deliver` in Figure 3). This ships the following three components.

- *Python client:* The python client has been built to a package during the `Build` step of our continuous integration pipeline. It is now automatically pushed to the *Python Package Registry* of our GitLab project. Every software which uses the twinstash client can then access this registry by script to download the current version.
- *Docker image of the twinstash service:* The twinstash service has equally been built as a python package during the `Build` step. Our pipeline now automatically runs a dedicated Dockerfile to create a Docker image. This is done by taking an existing Docker image capable of running a flask service and pushing our service package into it. The newly created image is then delivered to the *Docker Container Registry* of our GitLab project.
- *Docker image of the web GUI provider:* Again, by running a dedicated Dockerfile, a new Docker image is created. It pushes the GUI package built by Angular beforehand to an existing NGINX Docker image. This new image then also gets delivered to the Docker Container Registry of our GitLab project.

In a separate process, we also deliver the Keycloak via a Docker image (cf. Figure 1). Again, we provide a Dockerfile to build a dedicated customized image. It takes an existing Keycloak Docker image and enriches it with configuration settings with respect to DLR LDAP[14] communication and the possibility of adding external non-LDAP users.

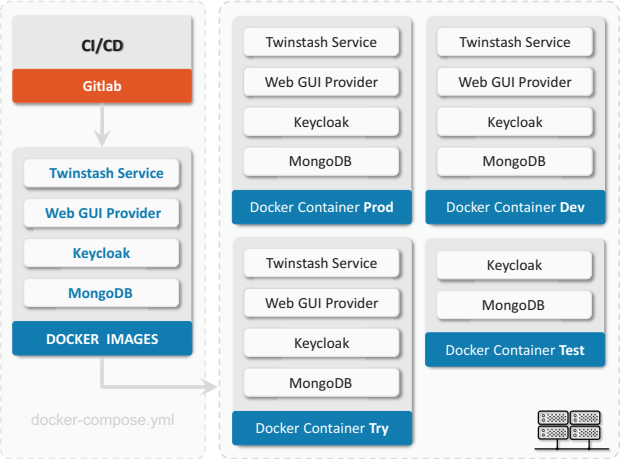

FIG 4. Deployment using four Docker images (left) to create four twinstash instances by spawning Docker containers (right).

3.3. Deployment

Our deployment utilizes four Docker images: twinstash service, web GUI provider, Keycloak, and MongoDB (cf. Figure 1). During Continuous Delivery described in Section 3.2, the images for the twinstash service and the web GUI provider are built and delivered. The Keycloak image is built and delivered in a separate process whenever we need an updated version. For the MongoDB we just take an off-the-shelf image. An instance of twinstash is launched using *Docker Compose*[15]. Our docker-compose file assembles all the necessary details on how to create

[14] https://de.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol

[15] https://docs.docker.com/compose/

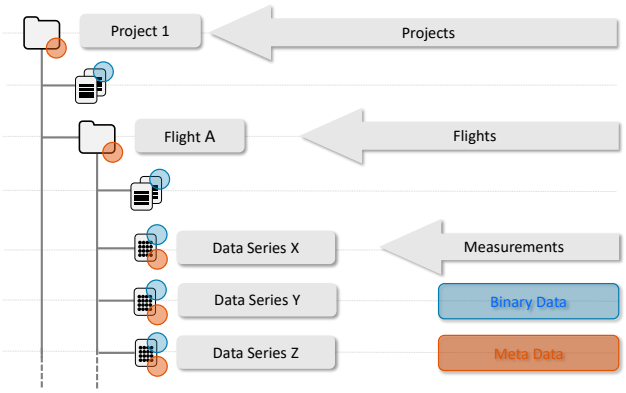

FIG 5. Hierarchical data model with projects, flights, and data series.

four Docker containers out of the four Docker images and how to correctly plug them together. The file is configurable with regard to passwords, ports and other parameters and thus allows to easily deploy multiple instances of twinstash even to the same machine.

Figure 4 shows our deployment of four twinstash instances. Besides the productive twinstash instance, we currently run a development, a test, and a try instance. The test instance only provides an environment to run integration tests against. It allows to test authentication with a special Keycloak instance and allows a non-persistent MongoDB instance to be filled with test data. All twinstash instances are currently self-hosted on servers of the Institute of Software Methods for Product Virtualization.

4. DATA MODEL

The twinstash data model depicted in Figure 5 resembles a filesystem tree. Each root element is a *project* which contains *flights* and some *files* such as project description or an image of the aircraft. Each flight contains data series objects and again some files like flight cards or images of the flight. Metadata can be attached on all levels to either project, flight, series or file.
Twinstash employs a MongoDB which provides storage based on *JSON*[16] *documents* and binary content. In our setting, we use JSON documents to store all metadata of a project, flight, series, or file. The tree structure depicted in Figure 5 is represented in these JSON documents implicitly by storing references to parent documents. The actual data of series and files is stored as binary content in the MongoDB.

4.1. Projects and Flights

An example of a project JSON document is shown (along with examples for flight and series) in Figure 6. Besides the properties `type` (which can have one of the values `project`, `flight`, `series`, or `file`), `id`, and `name`, the document has two system-generated properties `created_at` and `created_by` which store the time of creation and the username, respectively. The property `user_tags` can freely be filled by the user, also with a complex tree-like sub JSON. All the properties of a `project` JSON document are also present in types `flight`, `series`, and `file`.

FIG 6. JSON documents for project, flight, and series.

The JSON document for a flight (cf. Figure 6) has only one further property `parent` which references the project it belongs to.

4.2. Series and Files

Each series JSON document (cf. Figure 6) also has the `parent` property and uses it to reference the flight it belongs to. Property `unit` stores the unit of the parameter represented by this series. Property `series_connector_id` allows to connect several series just by sharing the same series connector ID. Each series represents a 1D data set in our data model, and we use series connector IDs to connect parameter values to corresponding time values this way. They also allow to join multiple parameters together to form a table which is useful e.g. to save database storage space when several parameters rely on the same time sampling. Property `represents` can be used to annotate a series with system information such as `time`, `longitude`, `latitude` or `height` which is interpreted by twinstash in order to construct a flight trajectory. Another property `statistics` will be automatically generated by twinstash upon upload of each series. It contains information like the minimum, maximum, mean, median and standard deviation for the uploaded series data. This is useful in particular for search queries such as to return all

[16] https://en.wikipedia.org/wiki/JSON

flights flying to an altitude of 5000 meters since this can be executed on the (rather small) series metadata instead of the (usually huge) actual data. This is also the reason for storing series data as separate binary content in the MongoDB which is linked into the series JSON document via property `data_id`.

Files like PDFs or images are stored similarly to series by storing a JSON document with the respective metadata linking the binary content stored in the MongoDB. The properties of a file are `type`, `id`, `name`, `parent`, `data_id`, `size`, `user_tags`, `created_at`, and `created_by`. Property `parent` references either a project or a flight.

5. FEATURES

We will now briefly present the main features of the twinstash system.

5.1. Projects View and Project View

One possibility to access data in twinstash is the projects view which presents a list of all projects. Clicking on one of its elements opens the project view shown in Figure 7 (top). It lists the projects metadata and files as well as all its flights. Each flight can be selected which triggers the display of its trajectory on a 2D map. This map allows to toggle supplementary flight-related information and supports the joint display of multiple flight trajectories in separate colors.

In addition to the 2D map, the project view provides a 3D trajectory view shown in Figure 7 (center) which equally supports the joint display of multiple flight trajectories. It allows to freely rotate and zoom the view in all three dimensions and offers a replay feature which can be used e.g. to evaluate how close several aircraft approached each other at certain points in time. A twinstash use case which employs this feature is described in [2].

5.2. Flight View and Series View

From the flight list of the project view, a flight may be selected to open the detailed flight view shown in Figure 7 (bottom). Like the project view, it shows the metadata and files of this flight. Furthermore, a list of all its parameters is shown. This list can be filtered by names and allows a selection of multiple parameters for joint time series plots. The plots itself are interactive and support auto scaling to the range of selected values. Moreover, series metadata and data values may be displayed alongside.

5.3. Aircraft, Calendar, and Search Views

In the twinstash web GUI, there are several options to access data of the currently about 1200 flights. The views described in Sections 5.1 and 5.2 are useful to dive into the tree structure starting from the projects view, navigating through the project view down to the flight view, finally reaching the series view.

Twinstash also provides the aircraft view shown in Figure 8 (top). It shows an image for each available aircraft along with information about it. Selecting such an image shows the flights of this aircraft. Another option is the calendar view shown in Figure 8 (center) which provides a chronological overview of all flights in a calendar style. Clicking on a flight opens the corresponding flight view. Finally, twinstash offers the search view shown in Figure 8 (bottom). It allows to enter search strings to match metadata according to its JSON structure. Section 6 shows how search works based on a python client example.

6. DEMO OF THE PYTHON CLIENT

Let us now consider how our python client can be used to interact with the twinstash backend. At first, we import the `Client` class.[17]

```
from stashclient.client import Client
```

An object of this class can be initialized passing the name of a twinstash instance as follows.

```
client = Client.from_instance_name('prod')
```

Let's say we know there is a project whose name ends with `Archive` and we want to access its content. The most powerful and flexible approach to achieve that is twinstashs search feature.[18] It allows to match documents according to their JSON structure, also by utilizing wildcards such as `$*` (to match an arbitrary string) or `$exists` (to check if a property exists).

```
client.search({
    'type':  'project',
    'name':  '$*Archive'
})
```

This search query will return a list of python dictionaries, each holding the JSON metadata of a project. We assume that there is only one such project and store it into a python variable `project`. Its dictionary is as follows.

```
{
  'id':  '63c4e8f168e767a7319229f3',
  'type':  'project',
  'name':  'SP Flight Archive'
}
```

We will now query all flights in that project utilizing the fact that all those flights carry this projects ID in their `parent` property. We write `collection` for the type instead of `flight` as twinstash uses this more

[17] In this demo, we show shortened versions of the python code and sometimes skip or reorder parts of its output. A fully functional and complete jupyter notebook can be found in the Appendix in Figures 9, 10, 11, and 12.

[18] The search feature is not restricted to the python client—also the web GUI provides the full search functionality via a dedicated page which allows to put search queries, cf. Figure 8 (bottom).

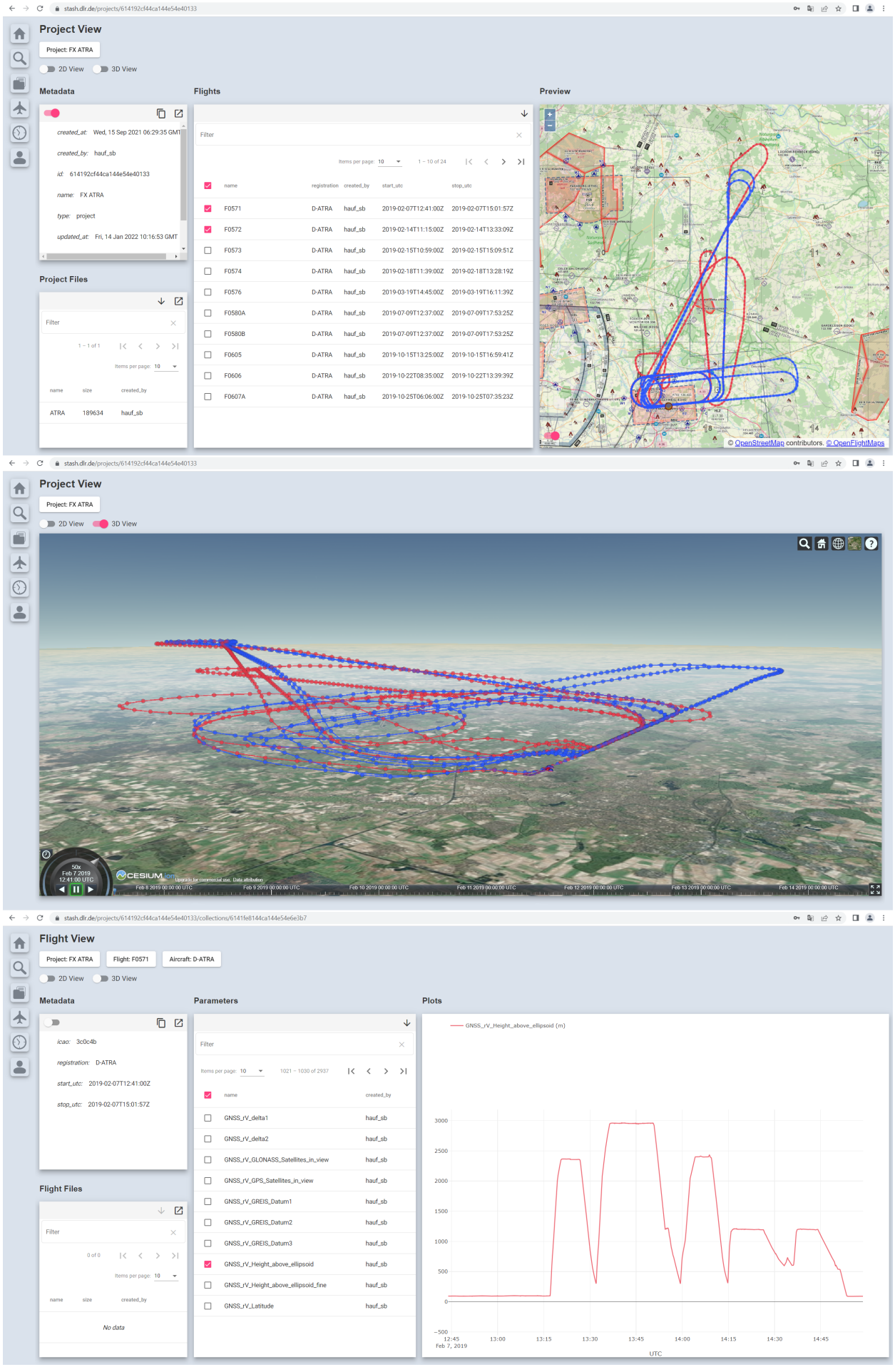

FIG 7. ***Top:*** **Project View with flight list and 2D map;** ***Center:*** **3D trajectory plot;** ***Bottom:*** **Flight View with parameter list and time series plot for selected parameter.**

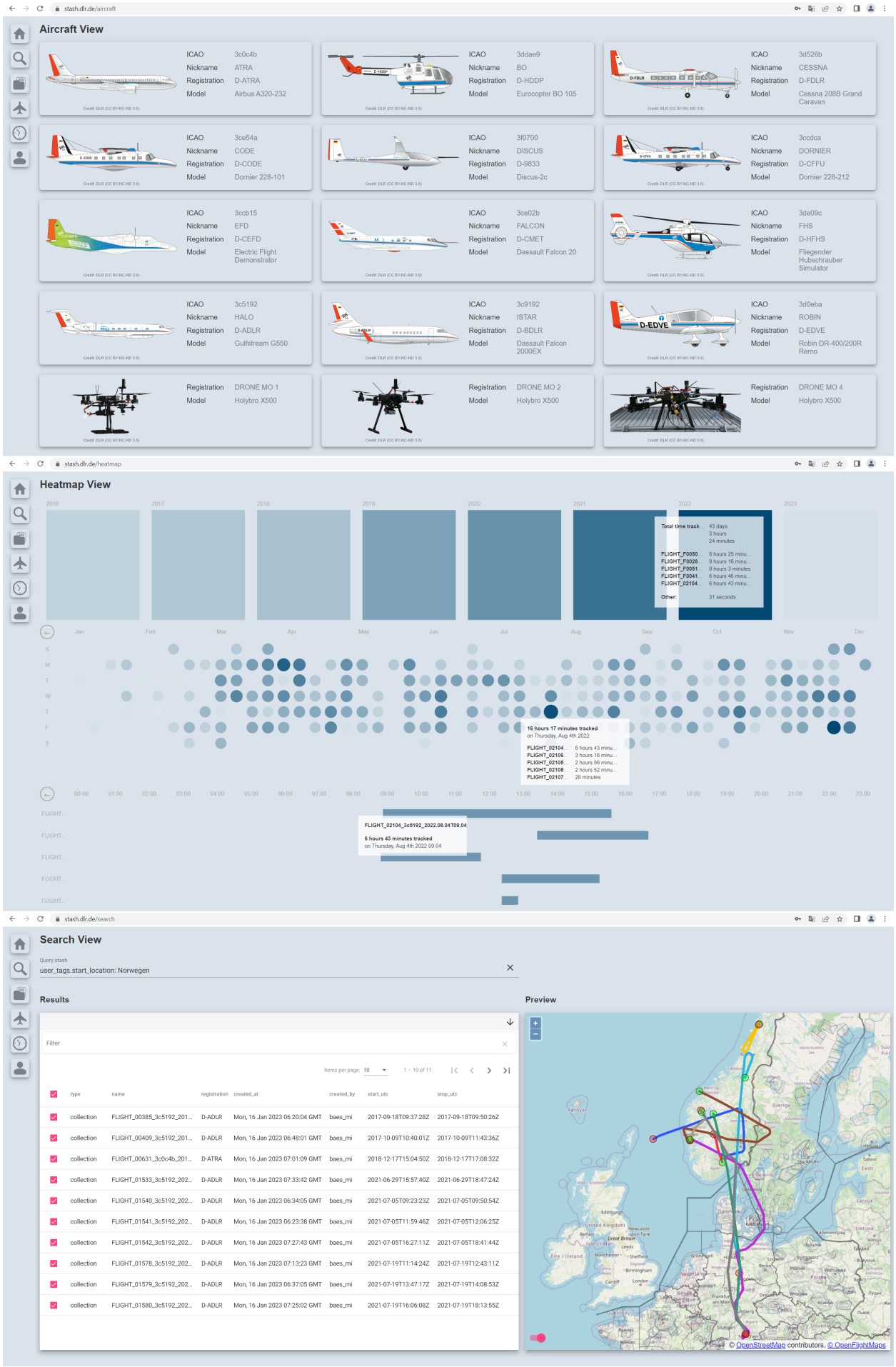

FIG 8. *Top:* Aircraft View; *Center:* Calendar View; *Bottom:* Search View with query and map preview of selected results.

generic name.

```
client.search({
    'type': 'collection',
    'parent': project['id']
})
```

The query returns 2537 dictionaries with metadata on flights, we will restrict the query to get a more meaningful result. Let's say we are interested in all flights the aircraft with an ICAO Aircraft ID of `3c5192` performed on the 28th of June 2021. The ICAO Aircraft ID is stored in the `user_tags` property of our flights, as is the coordinated universal time (in short UTC) for the start of the flight. We can utilize this information as follows.

```
client.search({
    'type': 'collection',
    'parent': project['id'],
    'user_tags.icao': '3c5192',
    'user_tags.start_utc':'2021-06-28$*'
})
```

Please note that we have therewith searched into subproperties of `user_tags`. In fact, our search supports arbitrarily deep subproperty queries. From our query result, we just take the first flight matching this query and store it into a python variable `flight`. It looks as follows.

```
{
  'id': '63c4ff8668e767a73194292a',
  'type': 'collection',
  'parent': '63c4e8f168e767a7319229f3',
  'user_tags': {
    'icao': '3c5192',
    'start_utc': '2021-06-28T08:07:15Z'
  }
}
```

Let us now take a closer look at the parameters that are stored for this flight. The query works quite similar to the one we used to obtain all flights.

```
client.search({
    'type': 'series',
    'parent': flight['id']
})
```

The query returns a list of python dictionaries, each containing the metadata of a parameter. Assume we want to have a closer look at parameter `geoaltitude`, the ellipsoidal height with respect to WGS84.

```
client.search({
    'type': 'series',
    'parent': flight['id'],
    'name': 'geoaltitude'
})
```

Only one parameter has this name. We therefore store the first and only entry of the returned list into a python variable `geoaltitude`. This is what we get.

```
{
  'id': '63c4ff8868e767a73194294f',
  'type': 'series',
  'parent': '63c4ff8668e767a73194292a',
  'unit': 'm',
  'represents': ['height'],
  'series_connector_id': 'scid_63c...',
  'statistics': {
    'max': 13601.7,
    'mean': 10573.5898521682,
    'min': 701.04
  }
}
```

As can be seen in the output, this parameter has a `height` tag that allows twinstash to know that this parameter contains height information for the trajectory calculation. The other parameters twinstash needs in that respect are tagged with `time`, `longitude`, and `latitude` (cf. Section 4.2). The parameter also has a `series_connector_id` which can be used together with the `time` tag to find the time series belonging to this parameter. The `statistics` property provides a lot of generated details about the parameter. For example we see that the flight went up to a maximum height of 13601.7 meters and was above 10000 meters on average. If we are interested in the actual data of this parameter, we can retrieve it as follows.

```
client.data(geoaltitude['id'])
```

This query returns a numpy array of parameter values. In order to get the time data, a second similar query would be needed. Let's have a closer look at the trajectory calculation that is performed in the twinstash backend. As mentioned before, twinstash is able to take the flight parameters tagged with `height`, `time`, `longitude`, or `latitude` to calculate dedicated trajectory information. The frontend for example uses this backend functionality to query the trajectories it renders in its plots shown in Figure 7 (top and center). The python client can access trajectory information as follows.

```
client.trajectory(
    flight['id'],
    sampling_rate_in_secs=10,
    altitude_unit='m'
)
```

The parameter sampling rate allows to control the time interval (and therewith the dataset size). While a preview in the browser must be quickly loaded but does not need a lot of precision, a dedicated flight route analysis may need all the details but could load slower. Trajectory information has the following structure.

```
{
  'id':  '63c4ff8668e767a73194292a',
  'units':  {
    'alt':  'm',
    'lat':  'deg',
    'lon':  'deg'
  },
  'items':  [{
    'alt':  701.04,
    'lat':  48.09009099410752,
    'lon':  11.29471998948317,
    'utc':  '2021-06-28T08:07:15+00:00'
    }, ...]
}
```

Property `id` provides the ID of the flight this trajectory belongs to. Property `units` contains the units of the altitude, latitude, and longitude. Property `items` finally lists all sampled coordinates. With a little python code, we can transform the trajectory data to a different format in order to pass it on to a matplotlib function which generates a 3D plot of the downloaded trajectory. The code and plot can be found in Figure 12 in the Appendix. This is only a small example on what is achievable with the python client. We might for example automatically search for broad ranges of data of a certain kind and utilize machine learning algorithms to obtain further insights.

7. CHALLENGES AND OUTLOOK

During the development of twinstash it was quickly (and obviously) clear that one single IT platform may not provide a general, all purpose, Digital Twin. Comparing ISTAR with, for example, a glider reveals differences that are difficult to address in the same pieces of code (even before considering domain). The main development challenges often lie in real life problems like ambiguous naming, unclean data, legacy data without support, incompatible data formats, proprietary closed source data formats, and manual steps involving proprietary software in a to-be-automated process chain. Those challenges usually require fundamental changes in already established processes of different institutes and departments.

The most important next steps in the development of twinstash are the composition of an automated pipeline for the import of new ISTAR data. Accumulating more and more ISTAR flight sensor data with an increasing number of sensors makes this feature a top priority item. Also, the demand for ISTAR data continuously rises together with the number of users of the twinstash system, rendering features like a user privilege management and a scalable distributed backup mechanism more and more important. These are about to be addressed in a separate project which focuses on enhancing the *Application Readiness Level* in order to prepare twinstash for productive usage. Within the DigECAT project, twinstash will embed measured 3D real geometry data [4] as a further important step towards ISTARs Digital Twin. Further developments will have a clear focus on providing intelligent functionality via the twinstash backend, e.g. a *parameter health monitoring* and an automated *safety reporting tool*.

ACKNOWLEDGEMENT

We kindly thank the OpenSky Network association [5] for granting free access to their database of historic flight data[19] within the scope of our research project. The intended prototype for an automated *safety reporting tool* shall among other criteria automatically determine the distance of the examined aircraft to other neighboring objects in airspace within a predefined range. This feature would not be feasible without the access to the OpenSky Network database.
The research associated with this paper was performed within the project *Digital Twin for Engine, Components and Aircraft Technologies* (DigECAT), which is the 2nd phase of the *Digital Twin* project within the aviation program of the German Aerospace Center (DLR).

Contact address:

sebastian.haufe@dlr.de

References

[1] Hendrik Meyer, Jonas Zimdahl, Alexander Kamtsiuris, Robert Meissner, Florian Raddatz, Sebastian Haufe, and Michael Bäßler. Development of a Digital Twin for Aviation Research. In *Deutscher Luft- und Raumfahrt Kongress*, September 2020.

[2] Emy Arts, Michael Bäßler, Sebastian Haufe, Alexander Kamtsiuris, Hendrik Meyer, Christina Pätzold, and Matheus Tchorzewski. Digital Twin for Research Aircraft. In *Deutscher Luft- und Raumfahrt Kongress*, September 2022.

[3] Shannon Bradshaw, Eoin Brazil, and Kristina Chodorow. *MongoDB: The Definitive Guide, 3rd Edition*. O'Reilly Media, Incorporated, 2019. ISBN: 9781491954454.

[4] Fiete Rauscher, Jörn Biedermann, Pia Allebrodt, Christina Pätzold, Frank Meller, and Björn Nagel. Permanent aktualisierte 3D-Realgeometrie des ISTAR im Digitalen Zwilling. In *Deutscher Luft- und Raumfahrt Kongress*, September 2022.

[5] Matthias Schaefer, Martin Strohmeier, Vincent Lenders, Ivan Martinovic, and Matthias Wilhelm. Bringing up OpenSky: A large-scale ADS-B sensor network for research. ACM/IEEE International Conference on Information Processing in Sensor Networks, April 2014. DOI: 10.1109/ipsn.2014.6846743.

[19] http://www.opensky-network.org

demo_dlrk_2022.ipynb

+ Code + Markdown | Run All | Clear All Outputs | Restart | Variables | Outline | ... | venv (Python 3.10.4)

```
[1]  from stashclient.client import Client
     client = Client.from_instance_name('prod')
     client.info()
```

✓ 0.4s Python

```
{'version': '1.33'}
```

```
[2]  projects = client.search({'type': 'project', 'name': '$*Archive'})
     projects
```

✓ 0.1s Python

```
[{'created_at': 'Mon, 16 Jan 2023 06:04:33 GMT',
  'created_by': 'baes_mi',
  'id': '63c4e8f168e767a7319229f3',
  'name': 'SP Flight Archive',
  'type': 'project',
  'version': '1.27'}]
```

```
[3]  project = projects[0]
     flights = client.search({'type': 'collection', 'parent': project['id']})
     len(flights)
```

✓ 1.6s Python

```
2537
```

```
[4]  flight = client.search({'type': 'collection', 'parent': project['id'], 'user_tags.icao': '3c5192', 'user_tags.
     start_utc': '2021-06-28$*'})[0]
     flight
```

✓ 0.1s Python

```
{'created_at': 'Mon, 16 Jan 2023 07:40:54 GMT',
 'created_by': 'baes_mi',
 'id': '63c4ff8668e767a73194292a',
 'name': 'FLIGHT_01527_3c5192_2021.06.28T08.07.15Z_2021.06.28T15.40.18Z',
 'parent': '63c4e8f168e767a7319229f3',
 'type': 'collection',
 'updated_at': 'Mon, 16 Jan 2023 17:50:25 GMT',
 'updated_by': 'baes_mi',
 'user_tags': {'aggregator': {'annotated_by': 'metrics_annotator.py',
   'annotated_on': '2023-01-16 18:50:25.366374',
   'average_altitude [m]': 10573.59,
   'average_velocity [km/h]': 762.3,
   'average_velocity [m/s]': 211.75,
   'total_distance [m]': 5756019.04},
  'aircraft': {'acars': False,
   'adsb': False,
   'categoryDescription': 'Large (75000 to 300000 lbs)',
   'icao': '3c5192',
   'manufacturername': 'Gulfstream Aerospace',
   'model': 'G550',
   'modes': False,
   'owner': 'Dlr Flugbetrieb',
   'registration': 'D-ADLR',
   'serialnumber': '5093',
   'typecode': 'G550'},
  'area_of_extent': {'delta_latitude_average [m]': 1114607.94,
   'delta_longitude_average [m]': 532092.56,
   'latitude_maximum [degree]': 53.6972614870233,
   'latitude_minimum [degree]': 43.66127014160156,
   'longitude_maximum [degree]': 13.28168596540179,
```

FIG 9. Jupyter Notebook Part 1/4

demo_dlrk_2022.ipynb

+ Code + Markdown | Run All | Clear All Outputs | Restart | Variables | Outline ··· venv (Python 3.10.4)


```
 'longitude_minimum [degree]': 6.02325439453125},
'creation_settings': {'date_of_conversion': '2023-01-15T23:49:09Z',
 'ground_test_maximum_delta_latitude_metric': 3000.0,
 'ground_test_maximum_delta_longitude_metric': 3000.0,
 'maximum_allowed_gap_within_flight_in_seconds': 600,
 'outlier_filtering_factor4maxspeed_to_still_accept': 1.0,
 'perform_ground_test_check': True,
 'perform_nominatim_request': True,
 'perform_outlier_filtering': True,
 'reject_flights_shorter_than_seconds': 600},
'data_source': 'opensky network',
'flight_duration [s]': 27183,
'flight_duration [timedelta]': '7:33:03',
'ground_test_screening_result': False,
'icao': '3c5192',
'outlier_filter': {'aircraft_max_speed [km/h]': 1093.0,
 'aircraft_max_speed [m/s]': 303.61,
 'aircraft_max_speed_accepted [km/h]': 1093.0,
 'aircraft_max_speed_accepted [m/s]': 303.61,
 'factor': 1.0,
...
 'start_utc': '2021-06-28T08:07:15Z',
 'stop_country': 'Deutschland',
 'stop_location': 'Bavaria, Deutschland, Germany, Bayern',
 'stop_utc': '2021-06-28T15:40:18Z'},
'version': '1.27'}
```


Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

```
parameters = client.search({'type': 'series', 'parent': flight['id']})
for parameter in parameters:
    print(parameter['name'])
```

[5] ✓ 0.1s Python

```
alert
baroaltitude
delta_time
determined_velocity_geoaltitude
geoaltitude
heading
lastcontact
lastposupdate
lat
lon
onground
spi
squawk
time
utc
velocity
vertrate
```

```
geoaltitude = client.search({'type': 'series', 'parent': flight['id'], 'name': 'geoaltitude'})[0]
geoaltitude
```

[6] ✓ 0.2s Python


```
{'created_at': 'Mon, 16 Jan 2023 07:40:56 GMT',
 'created_by': 'baes_mi',
 'id': '63c4ff8868e767a73194294f',
 'is_basis_series': False,
 'name': 'geoaltitude',
 'parent': '63c4ff8668e767a73194292a',
 'represents': ['height'],
 'series_connector_id': 'scid_63c4ff8668e767a73194292b',
```


FIG 10. Jupyter Notebook Part 2/4

demo_dlrk_2022.ipynb

+ Code + Markdown | Run All | Clear All Outputs | Restart | Variables | Outline ··· venv (Python 3.10.4)

```
 'statistics': {'max': 13601.7,
  'mean': 10573.5898521682,
  'median': 10789.92,
  'min': 701.04,
  'nans': 0.0,
  'size': 24352,
  'std': 2757.892162746514},
 'type': 'series',
 'unit': 'm',
 'version': '1.27'}
```

```
geoaltitude_values = client.data(geoaltitude['id'])
geoaltitude_values
```

[7] ✓ 0.1s Python

```
array([701.04, 716.28, 746.76, ..., 784.86, 777.24, 777.24])
```

```
trajectory = client.trajectory(flight['id'], sampling_rate_in_secs=10, altitude_unit='m')
trajectory
```

[8] ✓ 0.9s Python

```
{'id': '63c4ff8668e767a73194292a',
 'items': [{'alt': 701.04,
   'lat': 48.09009099410752,
   'lon': 11.29471998948317,
   'utc': '2021-06-28T08:07:15+00:00'},
  {'alt': 876.3,
   'lat': 48.09651520292638,
   'lon': 11.30274846003606,
   'utc': '2021-06-28T08:07:25+00:00'},
  {'alt': 990.6,
   'lat': 48.1031256206965,
   'lon': 11.31077693058894,
   'utc': '2021-06-28T08:07:35+00:00'},
  {'alt': 1066.8,
   'lat': 48.1099222474179,
   'lon': 11.31901667668269,
   'utc': '2021-06-28T08:07:45+00:00'},
  {'alt': 1165.86,
   'lat': 48.1154154114804,
   'lon': 11.32922832782452,
   'utc': '2021-06-28T08:07:55+00:00'},
  {'alt': 1264.92,
   'lat': 48.11783612784693,
   'lon': 11.34246826171875,
   'utc': '2021-06-28T08:08:05+00:00'},
  {'alt': 1615.44,
   'lat': 48.08357368081302,
   'lon': 11.38345571664663,
   'utc': '2021-06-28T08:08:55+00:00'},
  {'alt': 1607.82,
   'lat': 48.07314597954184,
   'lon': 11.38282189002404,
   'utc': '2021-06-28T08:09:05+00:00'},
  {'alt': 1592.58,
...
   'lon': 6.778433663504464,
   'utc': '2021-06-28T10:53:45+00:00'},
  ...],
 'name': 'FLIGHT_01527_3c5192_2021.06.28T08.07.15Z_2021.06.28T15.40.18Z',
 'units': {'alt': 'm', 'lat': 'deg', 'lon': 'deg'}}
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

FIG 11. Jupyter Notebook Part 3/4

demo_dlrk_2022.ipynb

+ Code + Markdown | Run All | Clear All Outputs | Restart | Variables | Outline ... venv (Python 3.10.4)

```
lon, lat, alt, utc = [], [], [], []
for item in trajectory['items']:
    lon.append(item['lon'])
    lat.append(item['lat'])
    alt.append(item['alt'])
    utc.append(item['utc'])
```

[9] ✓ 0.0s Python

```
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
fig = plt.figure(figsize=(18, 18), dpi=72)
ax = plt.axes(projection='3d')
ax.scatter3D(lon, lat, alt, c=alt, cmap='viridis')
```

[10] ✓ 0.6s Python

```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x179a79ada50>
```

</>

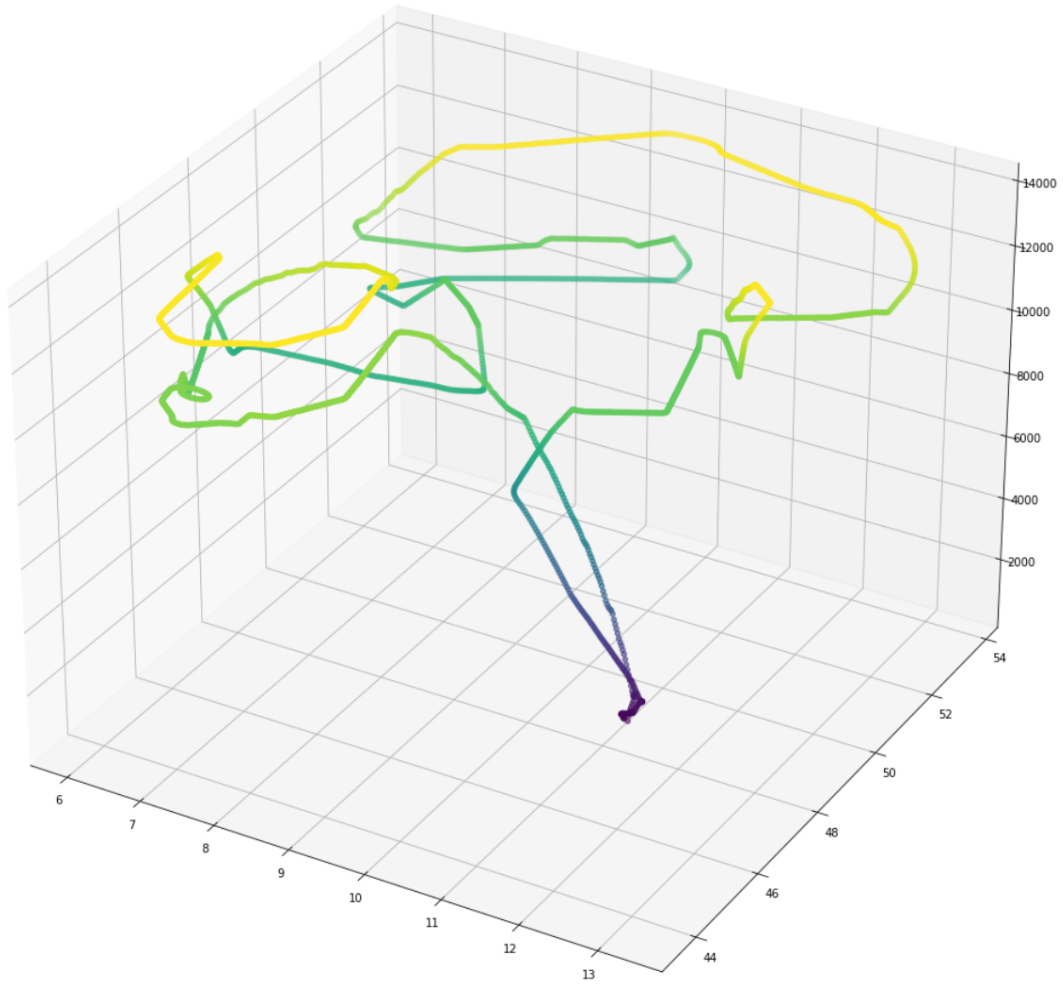

FIG 12. Jupyter Notebook Part 4/4