This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 861111

Drones4Safety

Research & Innovation Action (RIA)

Inspection Drones for Ensuring Safety in Transport Infrastructures

Platform implementation with modules for planning, monitoring, and control

D6.2

Due date of deliverable: 30.11.2021

Start date of project: June 1st, 2020

Type: Deliverable
WP number: WP6

Responsible institution: SDU
Editor and editor's address: Lea Matlekovic, SDU

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 861111

Version 1.0
Release Date: November 30, 2021

Project funded by the European Commission within the Horizon 2020 Programme		
Dissemination Level		
PU	Public	✓
CO	Confidential, only for members of the consortium (including the Commission Services)	

1 Executive Summary

The following deliverable is focused on software design and specification of Drone Inspection as a Service platform (DIaaS) and is part of the WP6 – Mission control and navigation. It describes the software architecture and implementation as well as tools used for development. The deliverable provides a manual for developers and users.

The document is structured in seven sections. After an introduction, the software architecture is presented. The section describes functional and non-functional requirement of the cloud system identified in D2.5. Furthermore, microservices architecture is described and services are functionally described. Fourth section describes software implementation in detail as well as tools used for implementation. Fifth section describes platform automatic deployment in Kubernetes cluster and tools integrated with the cluster. Last sections provide useful links for developers and users and describe how to contribute to the system as well as how to use it for autonomous infrastructure inspection mission planning, control, and monitoring.

Contents

References

FastAPI, https://fastapi.tiangolo.com/

Overpass API, https://wiki.openstreetmap.org/wiki/Overpass_API

MongoDB, https://www.mongodb.com/

NetworkX, https://networkx.org/

OR-Tools, https://developers.google.com/optimization

PostgreSQL, https://www.postgresql.org/

PostGIS, https://postgis.net/

Minikube, https://minikube.sigs.k8s.io/docs/

K3S Kubernetes, https://k3s.io/

Docker, https://www.docker.com/

Gazebo, http://gazebosim.org/

PX4, https://px4.io/

ROS2, https://docs.ros.org/en/foxy/index.html

GitLab https://docs.gitlab.com/ee/ci/

QGroundControl, http://qgroundcontrol.com/

Acronyms

Acronym	Description
D4S	Drones 4 Safety
API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
SQL	Structured Query Language
VNC	Virtual Network Computing
ROS	Robot Operating System
CI	Continuous Integration
CD	Continuous Deployment
CPU	Central Processing Unit
GPU	Graphics Processing Unit
OS	Operating System
PaaS	Platform as a Service
IaaS	Infrastructure as a Service
DIaaS	Drone Inspection as a Service
IP	Internet Protocol
URL	Uniform Resource Locator

2 Introduction

The main scope of the Drones4Safety (D4S) project is to develop a system of autonomous, self-charging, and collaborative drones that, inspecting an extensive portion of transportation infrastructures in a continuous operation, can increase the safety of the European civil transport network. The project outcomes, in forms of software services and hardware drone system, will offer to railway and bridge operators the chance to inspect their transportation infrastructure accurately, frequently, and autonomously.

The main purpose of this document is to provide a description and specification of software services needed for autonomous inspection. All the software enabling the infrastructure inspection is developed as a Drone Inspection as a Service platform. The platform includes cloud services for mission control, monitoring, and navigation. It also includes web interface and communication between the platform, web interface and drones.

The platform architecture is designed to support modifications and additions, so that old features can be easily modified, and new features can be easily added. Modularity is particularly important since we can expect to extend the platform with different infrastructure inspection capabilities in the future.

3 Software Architecture

Software architecture in general describes application or platform organization and structure. Architectural decisions impact application quality, performance, maintainability, and usability. Software architecture chosen at the beginning of the development can have a high impact on the application's future and affect its success. Architecture should be considered and planned if the application needs to be scalable, maintainable, and easily upgradable. When developing the application from scratch, the focus is usually on having a working product as soon as possible. However, that approach can become unsustainable when the number of platform features grows fast. If the architecture is not reconsidered, the development slows down and platform becomes difficult to maintain.

3.1 Software requirements

In order to design the architecture suitable for the scope of this project, we focused on system requirements defined in D2.5. From these system requirements, we identified cloud software requirements and developed the software in appropriate architecture style. The following subsections present software functional and non-functional requirements. Functional requirements describe how the platform should work and how the user can interact with it. Non-functional requirements describe characteristics the platform should have to assure satisfying performance and stability when users and developers interact with it.

3.1.1 Functional requirements

The main actors interacting with the platform are developers and platform users such as drone operators. The developers should be able to easily add features; without spending time on understanding the codebase. They should not be concerned with deployment. Therefore, the deployment should be automatized. However, it should be easy to test new features by interacting with previously implemented functionalities and drones. The operator should be able to select the inspection targets and plan the safe inspection route. At the operator's request, the routes should be calculated and stored in the database as a set of waypoints representing locations

with latitude and longitude. The route calculation determines the order of visiting the targets for each drone participating in the mission. The routes should be visualized on the 2D map and sent to the drones as waypoints used for navigation. The operator should be able to monitor the inspection progress and receive alerts in real-time. Inspection data and drone telemetry should be stored in a database and visualized on the web interface upon request. After the mission, the operator should be able to retrieve the mission plan, mission data, and mission results.

3.1.2 Non-functional requirements

We identified scalability, interoperability, modifiability, and performance as the platform's non-functional requirements. When load increases, the platform should adapt to the load and uninterruptedly continue working. Interoperability describes the platform's ability to interact and exchange data between components within the system as well as with the outside systems. For autonomous drone inspection mission planning, the platform should be able to exchange information between components and communicate with the drone systems. The software architecture should be designed to facilitate the development, communication between components, and addition of new features without modifying the rest of the system. Modifiability specifies the degree to which the platform is robust to changes. The platform should perform to satisfy the requirements and to do so efficiently. We expect the platform's growth in features and increased traffic in the near future. These requirements, when met, will facilitate the platform's growth.

3.2 Architecture design

Based on functional and non-functional requirements, we designed the platform in microservice software architecture and automatized the delivery process.

Microservice architecture, also known as Microservices, is a software architecture that structures an application as a collection of small, loosely coupled services. The services are independently deployable, highly maintainable, and testable. The concept was developed to overcome the downsides of monolithic architecture. Microservices have clear boundaries between each other and communicate through the HTTP protocol, usually by exposing a REST API and sending requests. Each service represents one capability that makes it easier to understand and locate the code. It also makes them robust to changes. Since they are small and deployed independently, they are easy to update and maintain. Services can be scaled independently and automatically, depending on the load. They can use different technology stacks, including programming language and data storage. That gives high flexibility to the development teams. However, there are some drawbacks of microservice architecture. The fact that a microservices application is a distributed system requires handling of fallacies the distributed computing carries. It means that developers must deal with the additional complexity.

There are many practices and tools developed to facilitate testing, integration, and deployment of microservices. DevOps is a combination of practices and tools designed to facilitate the delivery of applications. It aims to increase an organization's ability to deploy applications faster by removing the barriers between development and operations teams. DevOps practices automatize the delivery process and are implemented as a part of a production pipeline. Applied tools and practices depend on the application delivery requirements and goals. Continuous integration is the practice of merging changes to the main branch as often as possible. When developers commit local changes to the remote repository, automated build and tests can run there before proceeding to production. Remote repository platforms with built-in version control, like GitLab and GitHub, facilitate the collaboration between developers and enable continuous integration. These

platforms also integrate with different DevOps tools to enable continuous delivery and deployment. The practice of continuous delivery includes continuous integration and after a successful build, automatically propagates the application to staging. In staging, the application is deployed to the testing environment. There, the application including all services can be run and tested. Continuous delivery requires manual approval for release into production. However, the continuous deployment includes all the steps described in continuous integration and delivery, but instead of manual, the release process is also automated. Described process can vary in complexity depending on the number of services, test requirements, and in general, the deployment strategy.

3.2.1 Platform architecture

This subsection describes microservices design based on platform requirements. Services are designed to enable scaling of specific processes which we expect to experience heavier loads, e.g., route calculation. Figure 1 depicts architecture and communication flow. The drone operator interacts with the system using Web interface. Web interface communicates with the backend by sending requests and serving responses to the user. It is implemented and deployed as a microservice; therefore, it is described in this section. The operator selects inspection targets on the Web interface, which creates a POST request to the Missions service. The request contains target locations. Mission service has drone locations stored and, with received target locations, sends the POST request to the Routing Solver. Routing Solver uses A* Pathfinder to determine the order of visiting all the targets. A* Pathfinder requests the graph created from data stored in Towers, Railways, Bridges, and No-fly services to determine the shortest path for each combination of target location and drone location. Calculated routes are stored in the Missions service, visualized on the Web interface, and sent to the drones Simulation through the Message Broker. While the mission is in progress, the Message Broker sends drone telemetry data to the Drone Log service, where the data is stored. Drone Estimator service uses telemetry data to estimate drone location in periods when drones are not reporting to the cloud. Estimation data is stored in the Drone Log service database for mission simulation after the mission finishes. Message Broker updates Missions service database with inspection results. The individual services are described in following subsections as well as the Web interface.

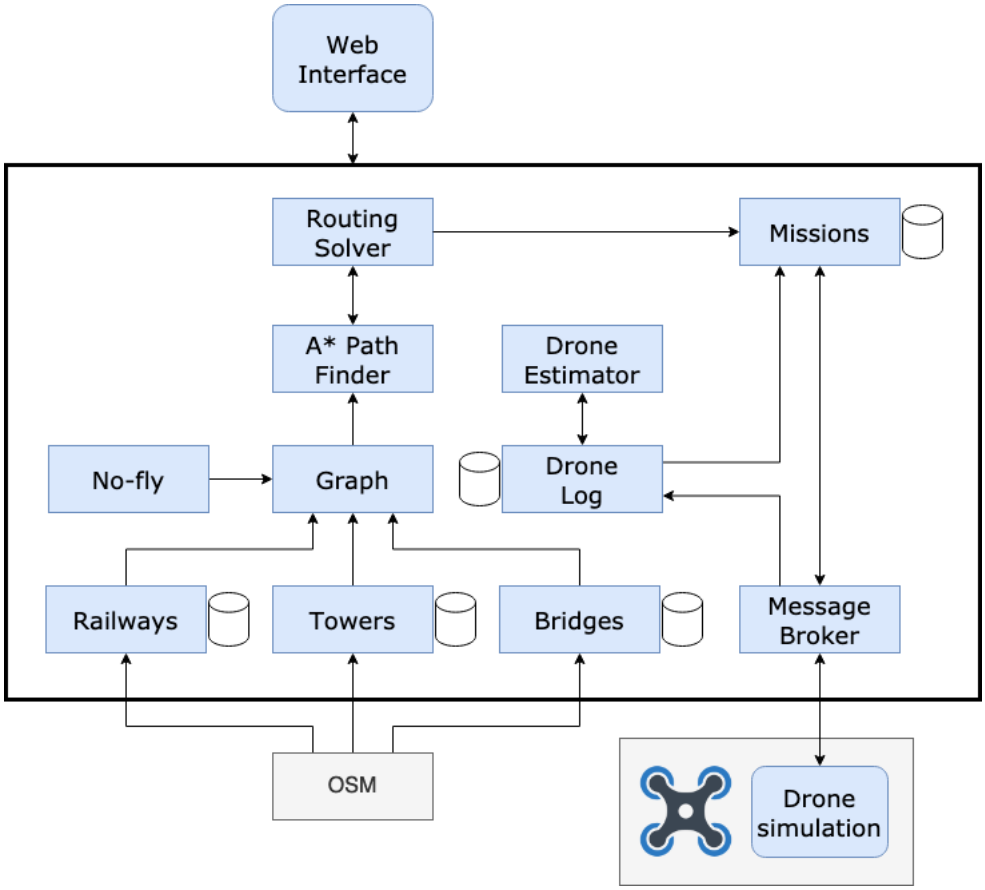

Figure 1 Microservices software architecture

3.2.1.1 Web interface

Based on the 2D map, the Web interface visualizes inspection targets on real locations and enables user interaction with the system. It allows the user to select the targets and to visualize calculated routes for each drone. The user can store and retrieve mission plans or execute them by sending the calculated navigation data to the drones.

3.2.1.2 Towers service

The service extracts power towers and power lines locations as points and lines from Open Street Map and stores it in the database. The service communicates locations data to the Graph service to create nodes and edges in the graph. The service also provides the data to the Web interface to visualize the inspection targets and enable the user target selection.

3.2.1.3 Railways service

The service extracts railways location as described in the Towers service, communicates with Graph service for graph creation, and provides the data to the Web interface.

3.2.1.4 Bridges service

The service extracts polygons around the bridges from Open Street Map and stores them to the database, communicates with Graph service for graph creation, and provides the data to the Web interface.

3.2.1.5 Graph service

The service creates a graph based on the data requested from Towers service, Railways service, Bridges service, and No-fly service. The graph contains nodes representing the infrastructure locations and edges containing pairs of neighboring nodes. The edges' cost is set up as the calculated distance between the neighboring towers. The service excludes nodes inside the no-fly areas using the data requested from the No-fly service. The graph is created only once, and it is served to the A* Pathfinder service to determine the shortest path between the set of inspection targets and drones. The new graph creation should run only when new locations are added to either towers, railways, or bridges databases.

3.2.1.6 Routing Solver service

The service receives inspection targets from the user via Missions service and drone locations from the vehicles. It determines which drone should inspect which target based on the shortest paths requested from the A* Pathfinder service. It returns the path for each drone as a set of waypoints containing graph locations in latitude and longitude. The results are stored in the Missions service database and sent to the Web interface for routes visualization.

3.2.1.7 A Pathfinder service*

The service requests a graph from the Graph service and performs the A* algorithm on the targets and drones' locations received from the Routing Solver service. It returns the shortest path between a target and a drone as well as the total path distance. The shortest path is calculated considering costs assigned to the edges. The shortest path calculations are used by the Routing Solver service to solve the vehicle routing problem.

3.2.1.8 No-fly service

The service contains areas where drone flights are not allowed and provides the data to the Graph service. Web interface requests No-fly areas and visualizes them on the 2D map.

3.2.1.9 Missions service

The service provides mission data management and storage. It stores missions planned by Routing Solver service, defining the mission route and tasks for each drone. Upon the user's request, the mission routes and tasks are sent to the drones for navigation. This service stores inspection results in the database after the mission finishes. The user requests a mission plan from the service, visualizes the routes, monitors the mission progress on the Web interface, or requests the mission visualization after the drone conducts an inspection.

3.2.1.10 Drone Log service

The service receives data from the drones and stores it in the database. The data consists of telemetry from drone sensors. For monitoring purposes, the service uses Drone Estimator to estimate the drone location and reports it to the Web interface. The location is updated with drone measurements every time they are received.

3.2.1.11 Drone Estimator service

The service estimates the drones' location in periods between the location data is received from the Drone Log service. Using the estimations, the user can monitor the drone estimated movements on the Web interface while the mission is in progress. Without the estimator, drone movements would be visualized only when received directly from the drones, which can be every 10 seconds or more. Estimations are sent back and stored in the Drone Log database.

3.2.1.12 Message broker service

The service enables communication with drones and manages the data distribution from the cloud to the drones as well as from the drones to the cloud. It keeps track of the successful delivery of messages and assures the distribution based on priorities.

3.2.1.13 Drone Simulation service

The service runs a drone model in the Gazebo simulator and provides the testing environment for both developers and users. For developers, it assures faultless integration of newly developed features or services into the system. For operators, the simulation allows the mission testing before the execution with the real drones.

4 Software Implementation

4.1 Microservices

The services use FastAPI web framework for exposing APIs to the Web interface or other services. FastAPI enables input validation by defining the expected request structure and provides interactive, automatically generated documentation where API for each service is visualized and tested. The documentation is available to the developers to test the services when adding new features. For extracting power towers, power lines, railways, and bridges data from Open Street Map, we use Overpass API and store it into the MongoDB no-SQL database. The MongoDB database stores data in JSON-like format and can be easily queried. Power towers are stored as a collection of objects with a unique identifier and location containing the tower's coordinates. Power lines are stored as a collection of objects with unique identifiers containing the array of unique identifiers representing the location points the line is passing through. Most of the points inside the array are also towers, but they can also be line intersections. For that reason, we created another collection of objects storing locations contained in the power line arrays. These point objects are stored the same as towers. Power line collection also contains tags with information on the number of cables, frequency, and voltage. The need for storing the power line information emerged because we want to find towers' direct neighbours to build the graph edges. Tower's direct neighbour is the closest tower, or more of them, laying on the same power line. Furthermore, MongoDB provides a geo-based search we use to find indirect neighbouring infrastructure locations. When creating the graph, we want to give a higher cost to the edges between the indirect neighbours than direct because the drones should fly close to the infrastructure whenever that is possible. Such a strategy is implemented for drone flight regulations reasons. We store railways in the same way as towers and lines, while bridges are stored as polygons. Each polygon is described as a way type containing unique identifiers for point locations in the polygon. The point objects are stored in the same database containing the points' locations. Bridges' locations can be combined with the power lines and railways locations to create a graph and enable the drones to reach a bridge. The graph is generated using NetworkX library, allowing us to create nodes and edges from point locations. The Routing Solver service uses OR-Tools to determine which drone should inspect which target based on the distance between them. It creates asynchronous requests to the A* Pathfinder service to get the shortest path for each combination of drone and target. A* algorithm is implemented from the NetworkX library and applied on the graph requested from the Graph service. When A* Pathfinder service determines the shortest path for each combination of target and drone, the Routing Solver service builds the distance matrix and based on the path costs, finds a solution to the vehicle routing problem. The Routing Solver service returns the path for each drone as a set of waypoints containing locations in latitude and longitude. Missions service implements database and stores paths calculated by the Routing Solver service. We created the database using PostgreSQL with PostGIS database extender, adding support for geographic objects and allowing location queries to be run in SQL. It stores missions with unique identifiers and dedicated routes and tasks. Routes store waypoints for navigation for each drone participating in the mission, while tasks store details of what should be inspected during the navigation. Drone Log creates swarm and drone unique identifiers and stores telemetry data binding it to each drone. Swarm represents a group of drones participating

in the mission. Drone Estimator service implements simple physics formulas to determine the drone's location based on the last two location measurements received from the drone. From the measurements, the service estimates velocity and acceleration vectors and calculates position. The calculation is sent and stored in the Drone Log database.

4.2 Python frameworks

4.2.1 FastAPI

FastAPI is web framework for developing RESTful APIs. FastAPI is based on Pydantic and type hints to validate, serialize, and deserialize data, and automatically auto-generate OpenAPI documents. It fully supports asynchronous programming and can run with Uvicorn and Gunicorn. It is a useful tool for designing software architecture in microservices. Each service runs as independent application with defines data models and communicates with other services through synchronous or asynchronous requests.

4.2.2 Overpass API

The Overpass API is a read-only API that serves data of the OpenStreetMap. It acts as a database over the web: the client sends a query to the API and gets back the data set that corresponds to the query. OpenStreetMap is a collaborative project creating a free, editable geographic database of the world. The geodata underlying the maps is considered the primary output of the project. It contains infrastructure data like locations, infrastructure types, and infrastructure specifications. We extracted the infrastructure data and stored it in our database for better organization and faster queries.

4.2.3 NetworkX

NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. Specifically, we use it to create graph structure and to manipulate through it. Graph structure contains nodes and edges. In our case, nodes are points in space representing infrastructure location with latitude and longitude. Edges are connections between nodes and contain information on edge type and weight. When using A* algorithm, edge weight has a crucial role in determining the shortest path. A* algorithm will take into consideration edge weight and if the weigh equals distance between two nodes, the algorithm finds the shortest path. Even if a path is the shortest in distance, there might be some other criterium worth considering while determining the best route for drones. In that case, we add additional weigh or scaling factor.

4.2.4 OR-Tools

OR-Tools is an open source software suite for optimization, tuned for tackling the world's toughest problems in vehicle routing, flows, integer and linear programming, and constraint programming. In our case, we use it to solve multiple vehicle routing problem with open ends; determine a set of routes for multiple drones given the target destination where the drones are not required to go back to the starting locations.

4.3 Databases frameworks

4.3.1 MongoDB

MongoDB is a source-available cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with optional schemas. We use it to store infrastructure data in JSON-like documents which are easily queried. MongoDB supports geo-queries when

database contains geographic locations. The user is able to query elements within specified distance or range, which is very useful in our case for determining which nodes in the graph are neighboring nodes. Services which use MongoDB are Towers, Railways, and Bridges. The data is represented with models shown in Table 1. Data models determine a way the data is stored in database. Database contains information about power towers, lines, bridges, and railways locations and specifications. Towers are defined as "nodes" and data model contains location. Lines are defined as "ways" and contain array with unique identifiers for nodes. Nodes' location is stored with the same data model used for towers. Bridges locations are stored as nodes and bridge unique identifier represents specific bridge. Railways are stored using the same data model used for lines.

Table 1 Data models as stored in the database

Towers	Lines	Bridges	Railways
_id: ObjectId type: "node" id: tower_id location: Object	_id: ObjectId type: "way" id: line_id nodes: Array tags: Object _id: ObjectId type: "node" id: node_id location: Object	_id: ObjectId type: "node" id: bridge_id lat: latitude lon: longitude	_id: ObjectId type: "way" id: railway_id nodes: Array tags: Object _id: ObjectId type: "node" id: node_id location: Object

4.3.2 PostgreSQL

PostgreSQL, also known as Postgres, is a free and open-source relational database management system emphasizing extensibility and SQL-compliance. In Missions and Drone Log services we created relations using PostgreSQL because this database will experience user queries and complex join queries for which NoSQL databases are not well adapted.

Data needs to be transferred between services in the cloud, to the drone, and from the drone to the cloud. Database relations specify mission data (including inspection tasks), telemetry data, and inspection results data. Therefore, we designed databases in Missions and Drone Log services to create global data space. Figure 2 shows relations implemented in these services and relationships between them. Relations contain attributes described as:

- **Missions relation** - the relation contains mission data, like mission name, status of the mission, swarm participating in the mission, routes associated with it, geofence regions where the flight is permitted, and time information.

- **Task relation** - the relation specifies tasks given to the drone. It describes the inspection type, location of task execution, and the drone the task has been delegated. It stores specifications describing which sensors to use e.g., RGB camera, frequency of data acquisition, speed of the drone during data acquisition, etc.

- **Result relation** - the relation stores inspection results and related data. It contains time data when the results are received, the location where the result was obtained, the fault description, and the image ID associated with the fault image store in object storage.

- **Telemetry relation** - the relation stores drone telemetry data like position, orientation, velocity, speed from different sensors. It also stores the drone's status, battery status, and onboard computer status.

- **Swarm relation** - the relation stores swarm identifiers for swarms participating in the mission.
- **Drone relation** - the relation stores drones participating in the mission.

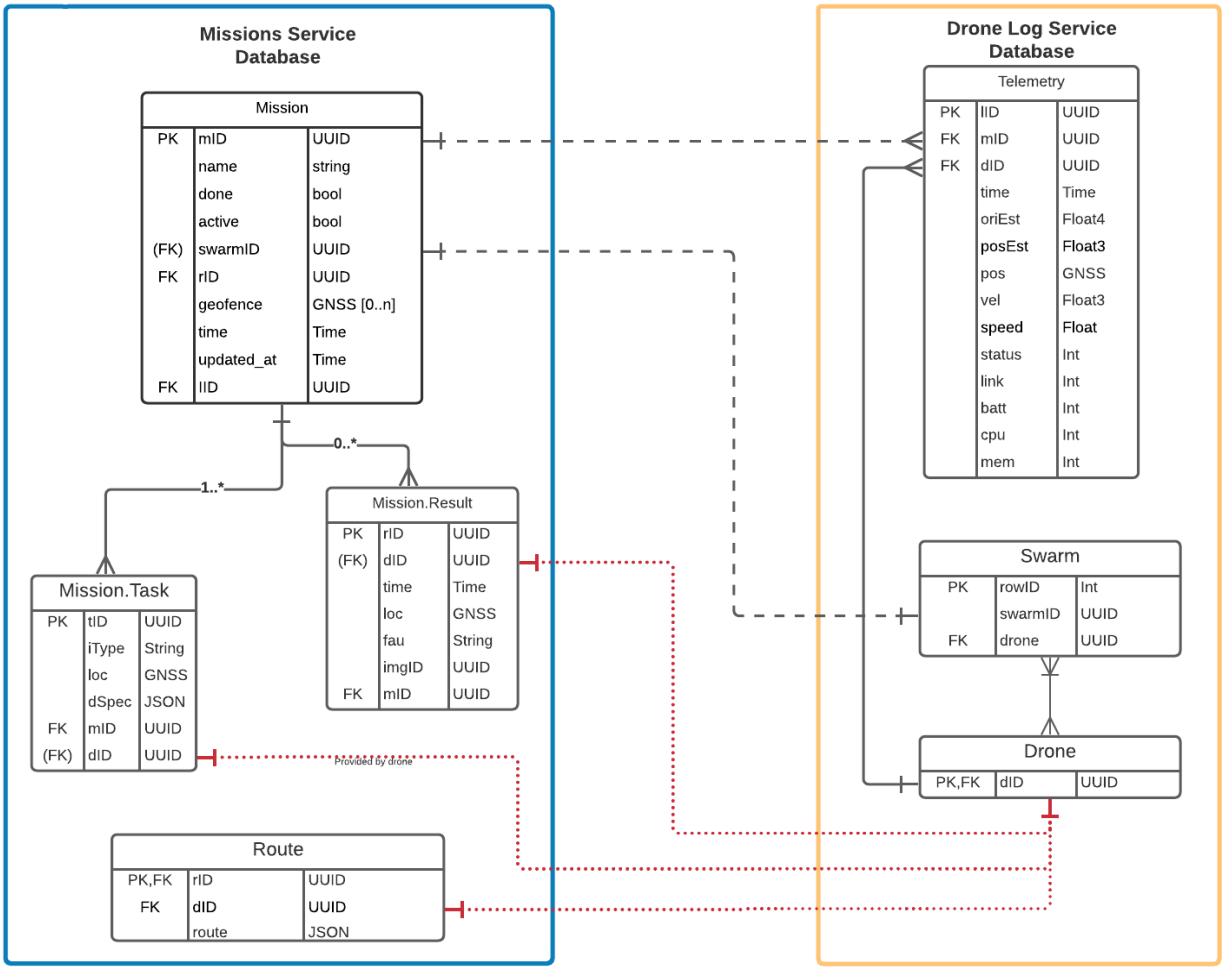

Figure 2 Database diagram

4.4 Drone Simulation as a Service

The drone simulation runs in a Docker container within the Kubernetes cluster. We created a Dockerfile for building the docker image and running the simulation in a container. The image is based on Ubuntu 20.04 LTS image with Xfce desktop environment and Virtual Network Computing (VNC) servers for headless use. Xfce is a lightweight desktop environment for UNIX-like operating systems. It aims to be fast and low on system resources while still being visually appealing and user-friendly. The image uses VNC to view the desktop environment from the outside of the container. We used it to visualize the building process and to verify that the simulation runs successfully. On the base image, we installed Gazebo to simulate the Iris Quadrotor 3D model. The vehicle's model is a dynamic model based on the body's mass and moments of inertia. For drone low-level control we use PX4 Autopilot. For offboard control, we installed ROS2 Foxy Fitzroy distribution

and implementation of drone point-to-point flight. To integrate PX4 and ROS2 and enable message exchange, we use the default middleware Fast DDS. The drone reports its location to the cloud every 10 seconds and requests route waypoints every 5 seconds. The Mission service provides route waypoints only when the drone unique identifier coincides with the unique identifier of the drone requesting the route. The simulation is exposed using Gzweb Gazebo Web client and can be reached through the web browser.

5 Software Deployment

In order to deploy the application automatically every time the changes are made, we set up a pipeline employing DevOps tools and technologies. We use GitLab to store the code, enable collaboration, and configure the delivery pipeline. Gitlab has built-in tools for software continuous integration and continuous deployment (CI/CD) which are used to run the pipeline. For automatizing deployment, scaling, and management of containerized applications we use Kubernetes. We set up the K3s Kubernetes cluster made of one master node and two agent nodes running on three Linux Ubuntu 18.04.5 LTS servers. Cluster decentralization improves the system's robustness assuring that the application will keep running even if one agent node fails. To automatize the deployment, we integrated the cluster with GitLab by providing the cluster API and token. To be able to run the CI/CD jobs, we installed the GitLab runner on the cluster. Each service contains a Dockerfile used for building docker images. The repository contains the configuration file gitlab-ci.yml defining the pipeline stages and CI/CD jobs. In the first stage, the GitLab runner runs the pipeline configuration file set up to build a docker image for each service when changes are pushed to the repository. Docker images are pushed to the Gitlab Container Registry. In the second pipeline stage, the images are pulled and services are containerized and deployed in the Kubernetes cluster where they run in pods. The deployment configuration file for each service specifies the number of pod instances or sets up the automatic pod scaling.

5.1 DevOps tools

5.1.1 GitLab

GitLab is an open source end-to-end software development platform with built-in version control, issue tracking, code review, CI/CD, and more. It is also a DevOps platform that combines the ability to develop, secure, and operate software in a single application. In GitLab, the project is kept in repository that all the developers have access to. The developers clone the repository on their local settings and collaborate with others. Each developer works on a dedicated branch and, after adding and testing new features in the code, merges his/hers work to the main branch. The development pipeline can vary depending on the organization and project goals. GitLab enables integration with Kubernetes cluster and provide tools to achieve automatic build and deployment to the cluster. Automatic processes are defined by setting up configuration files in the repository.

5.1.2 Docker

Docker is a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers. Containers offer a logical packaging mechanism in which applications or services can be

abstracted from the environment in which they run. It provides a consistent environment, including software dependencies, specific programming language runtimes, libraries, etc., no matter where the container is deployed. Containers virtualize CPU, GPU, memory, storage network resources. It provides isolated environments for running services and enables assignment of resources, e.g. CPU, GPU, with considering system and service requirements, e.g. reliability, real-time. Docker is popular, open-source container format that is supported on many cloud platforms and by Kubernetes system. Docker builds a containerized environment based on the Docker image. We use it for automated builds and container deployments in the Kubernetes cluster. In GitLab, the Docker images are created using GitLab runners and stored in the container registry. The image is taken from the registry whenever the deployment is triggered.

5.1.3 Kubernetes

Kubernetes is an open-source container-orchestration system for automating computer application deployment, scaling, and management. Kubernetes groups containers that make up an application into logical units for easy management and discovery. In case of the increased traffic, the Kubernetes is responsible to make replicas of the service and distribute the traffic to all of them. Many cloud services offer a Kubernetes-based platform or infrastructure as a service (PaaS or IaaS) on which Kubernetes can be deployed as a platform-providing service. Many vendors also provide their own branded Kubernetes distributions.

During the development and deployment of platform for mission planning, monitoring, and control of autonomous drone inspections, we set up two different Kubernetes distributions. The first one in Minikube. Minikube runs single-node cluster in a Virtual Machine. Kubernetes cluster, in general, consists of master and worker machines. Worker machines, also called nodes, run containerized applications in pods while the master manages workers and makes sure that the cluster is working in a configured way. In Minikube the master process and the worker process run on one node. The master node contains an API server which is the cluster entry point. The initial deployment and Minikube architecture are depicted in Figure 3. API server (1) runs a process that enables communication with the cluster. To visualize and manage all the cluster applications we can use the dashboard, as a web user interface, communicating with the API server. Otherwise, we can use API to communicate with the API server from the script or simply we can manage the cluster by writing commands in the terminal using a command-line tool. The master node keeps track of a cluster by running a controller manager. Controller manager (2) is responsible for checking if all the nodes and pods are running and, if one goes down, the controller manager replaces it with a new one. Scheduler (3) decides on which node a newly created pod will run, based on the available and required resources. Etcd key-value storage (4) in the master node keeps status data about the nodes. Master nodes and worker nodes communicate through a virtual network. The virtual network assigns an internal IP address to each pod. Pods communicate through the services (5) which contain permanent IP addresses. In case a pod restarts it will keep the service with the same address. Each service also provides a load balancing. Network rules for allowing the communication through the services are maintained by Kube proxy (6) running on each worker node. Kubelet (7) agent runs on each node assuring that containers run in pods as described in pods specifications. Container runtime is responsible for running the containers.

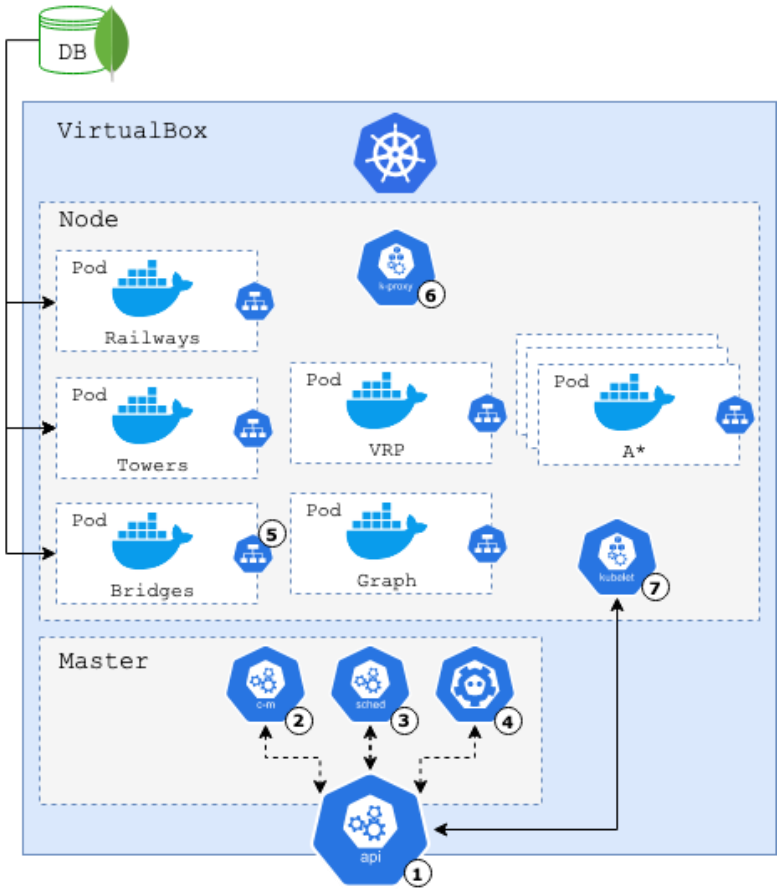

Figure 3 Platform deployed in Minikube

The final Kubernetes cluster we set up to deploy the platform is K3S. K3S is a multi-nodes Kubernetes cluster where nodes are distributed through multiple servers. We set up one master node and two worker nodes each on different server. In comparison with Minikube, K3S provides more reliability by distributing pods through multiple machines. The load balancer manages incoming requests and distributes the traffic to available pods running on worker nodes. Pod multiplication and asynchronous communication assures that route calculation provides results faster, even when many users are sending requests to the backend cloud system.

5.2 Continuous integration and deployment (CI/CD)

Microservices architectural style is suitable for continuous integration and deployment. Using DevOps tools, we enabled continuous integration of new features and services into the platform and deployment to the cloud. Collaboration between developers through GitLab facilitates integration from local environment into the repository. GitLab's ability to integrate with Kubernetes cluster enables continuous deployment. CI and CD are set of operating principles, and collection of practices that enable application development teams to deliver code changes more frequently and reliably. The implementation is known as the CI/CD pipeline. Since these procedures automatize deployment steps, developers can focus on meeting the project requirements, code quality and security. Continuous integration drives developers to implement minor changes and check in code

to GitLab frequently. It is expected that, by implementing the continuous integration, the platform development will speed up since all the developers will be able to see and work with the updated code. It leads to the better collaboration between the developers and better software quality. Continuous delivery automates the applications delivery to the selected infrastructure environment, in our case to University servers. We set up two different clusters and compared their performance. The final choice was the 3-node K3S cluster distributed through three servers. In comparison with one node cluster we set up in the beginning, the cluster with three nodes has an ability to continue working and distribute resources to available nodes. In the case when one of the servers stops running or the node crashes, the Kubernetes system will distribute the resources to rest of the working nodes and the system will continue running interruptedly.

6 Developer Manual

Developers are highly encouraged to familiarize themselves with the content of the deliverable to gain the understanding of the software architecture and software functionalities. The codebase can be found in the GitLab repository with URL given in Table 2. At this stage, the repository is private, and the access can be given upon request.

Table 2 Working repository URL

URL	Description
https://gitlab.sdu.dk/matlekovic/D4S	Working repository

Each developer, when contributing to the system, is required to create their own branch and clone the existing code from the master branch. The code for each service is situated in separated folders. If additional service is added, it has to follow the current organization and be developed in a new folder. Required files in the folder are given in Table 3. Additional files, implementing service features, can freely be added.

Table 3 Files in each microservice folder

File name	Description
__init__.py	Python module definition
__main__.py	Runs service as a Python module
Dockerfile	Used for creating Docker image; contains instructions for installation of dependencies and running the service in containerized environment
***.yml**	Service configuration file for deployment in Kubernetes
router.py	Defines service endpoints and data objects

For newly created services, locally developed features can be tested by sending requests to existing services. All the endpoints and associated data object types can be visualized through the URL given in the Table 4.

Table 4 Interactive documentation URL

URL	Description
http://limic.drones4safety.eu/dev.html	Interactive documentation for all services

7 User Manual

The user can interact with the platform using Web interfaces through URLs given in Table 5.

Table 5 Platform URLs

URL	Description
http://web.limic.drones4safety.eu/	Mission planning, control, and monitoring interface
http://drone1.limic.drones4safety.eu/	Drone simulation
http://vnc.drone1.limic.drones4safety.eu/	QGroundControl station for direct drone control

The first URL shows map-based interface for mission planning, control, and monitoring showed in Figure 4. Blue circles represent infrastructure elements which can be inspected. Blue circles turn green when the user selects an element for inspections. Orange circles show drones waiting for the routes to start with an inspection. Red circles show example of infrastructure defects and how it would be represented on the Web interface when drones find faulty infrastructure during the inspection. The user selects infrastructure elements for inspection and presses “Calculate Route” in the upper right corner to determine the optimal inspection schedule and routes. When the user initiates route calculation, cloud services receive targets and drone locations and return inspection routes for each drone. Figure 5 shows calculated routes on the Web interface. The user can name the mission plan and store it. When the plan is stored, each drone receives the mission waypoints and starts executing the global mission by flying point-to-point.

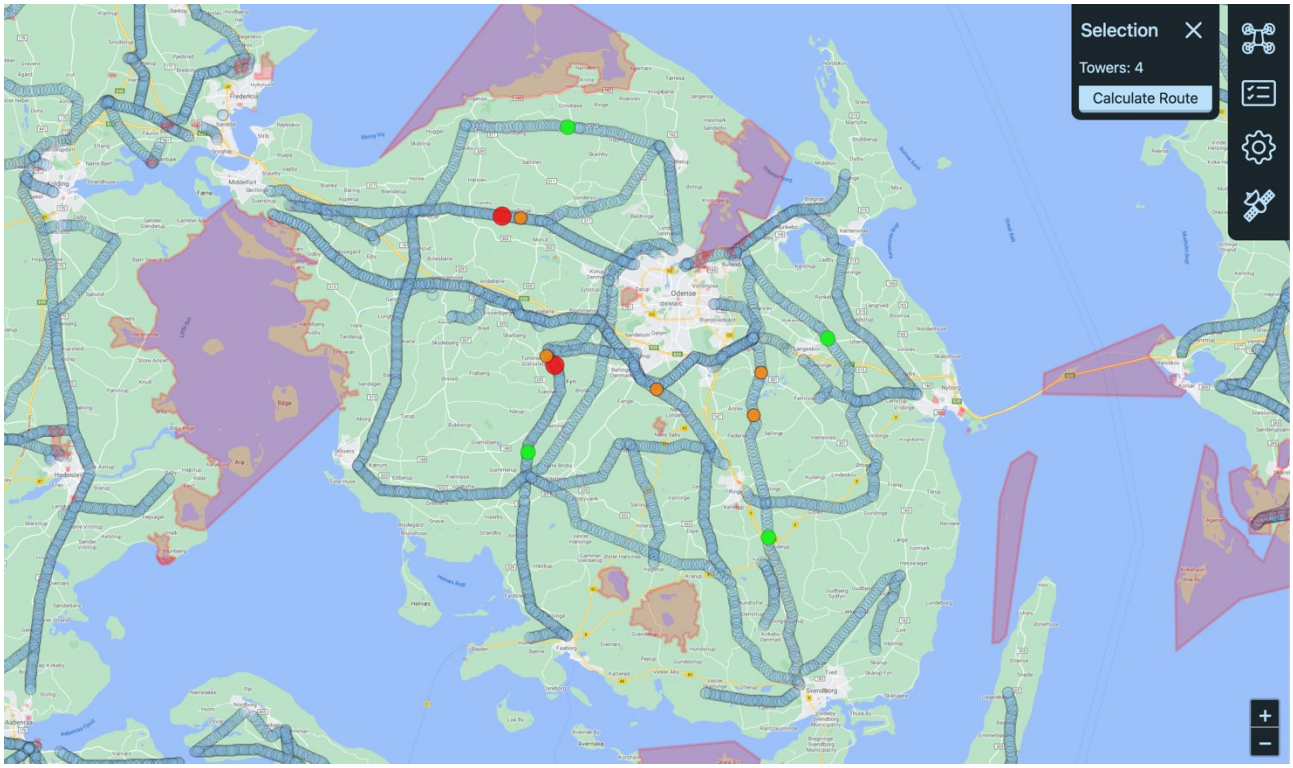

Figure 4 Web interface for mission planning, control, and monitoring

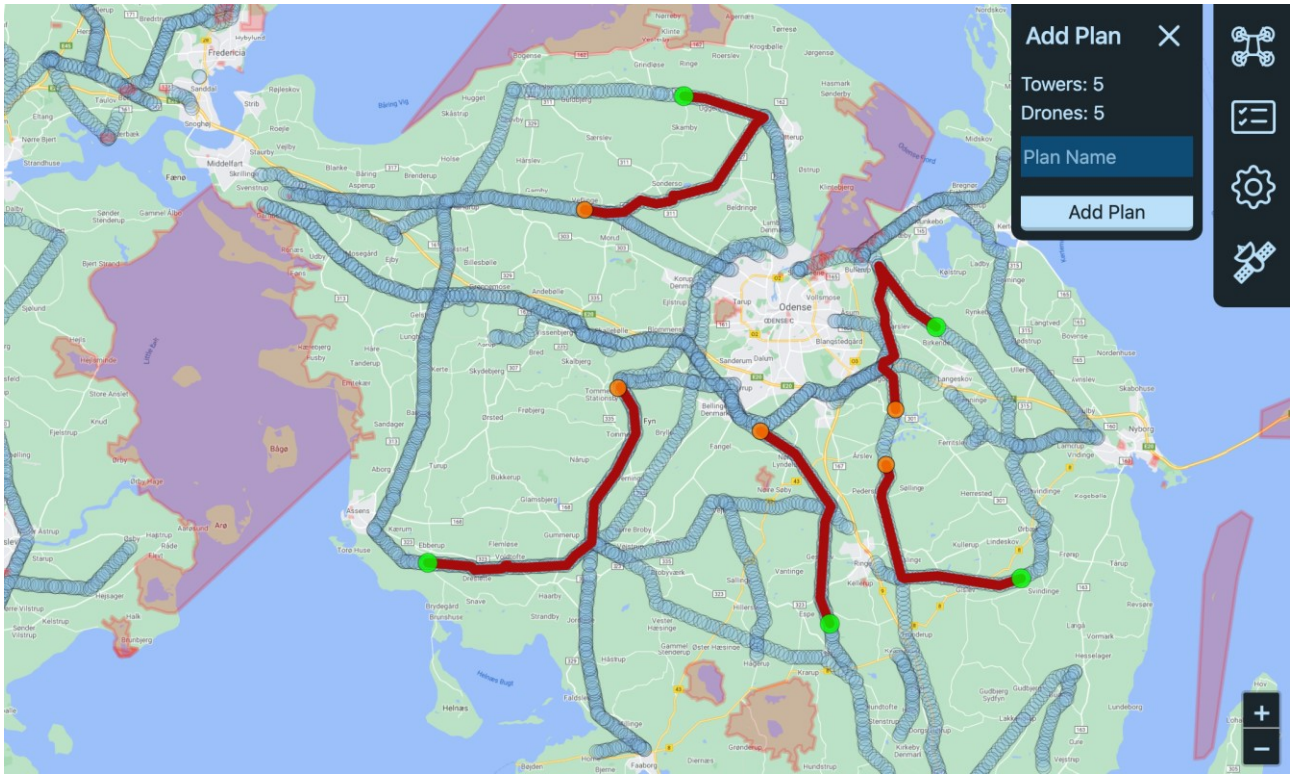

Figure 5 Inspection routes calculated and showed on the Web interface

Drone flight is tested with simulation that can be reached on the second link in the Table 5. The drone simulation is shown in Figure 6. The simulation can be deployed for each drone participating in the mission. For this proof-of-concept, we deployed one drone with URL prefix drone1. Additional drones would contain prefix depending on the number of drones participating in the mission, e.g., drone2, drone3, etc. After the drone receives mission waypoints, it starts executing the mission by flying in the simulation. The drone movement is visualized on the Web interface along the route which the drone is pursuing.

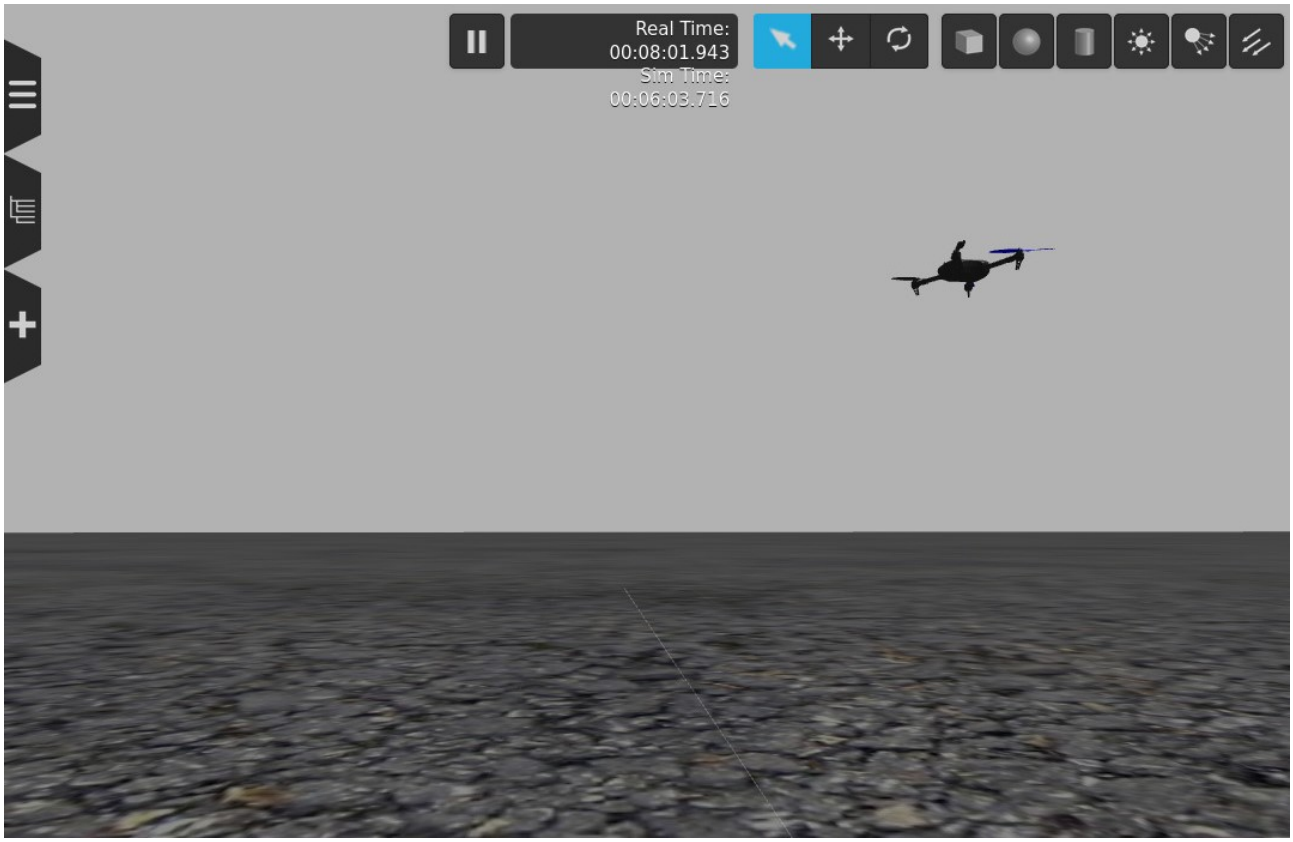

Figure 6 Drone simulation as a Service exposed through the Web interface

The third URL in Table 5 exposes QGroundControl station connected to the drone in the simulation. In case of emergency, the user can override the mission received from the mission planner by taking a direct control over the drones using QGroundControl. The user can send the drone commands for landing, take-off, or directly control the drone using virtual or physical joystick. Web interface with QGroundControl is showed in Figure 7. It is also possible to plan custom missions, independent of infrastructure locations, which can be useful for sending the drones to the start locations and end locations after the mission is finished. Detailed instructions for using QGroundControl can be found on the official website.

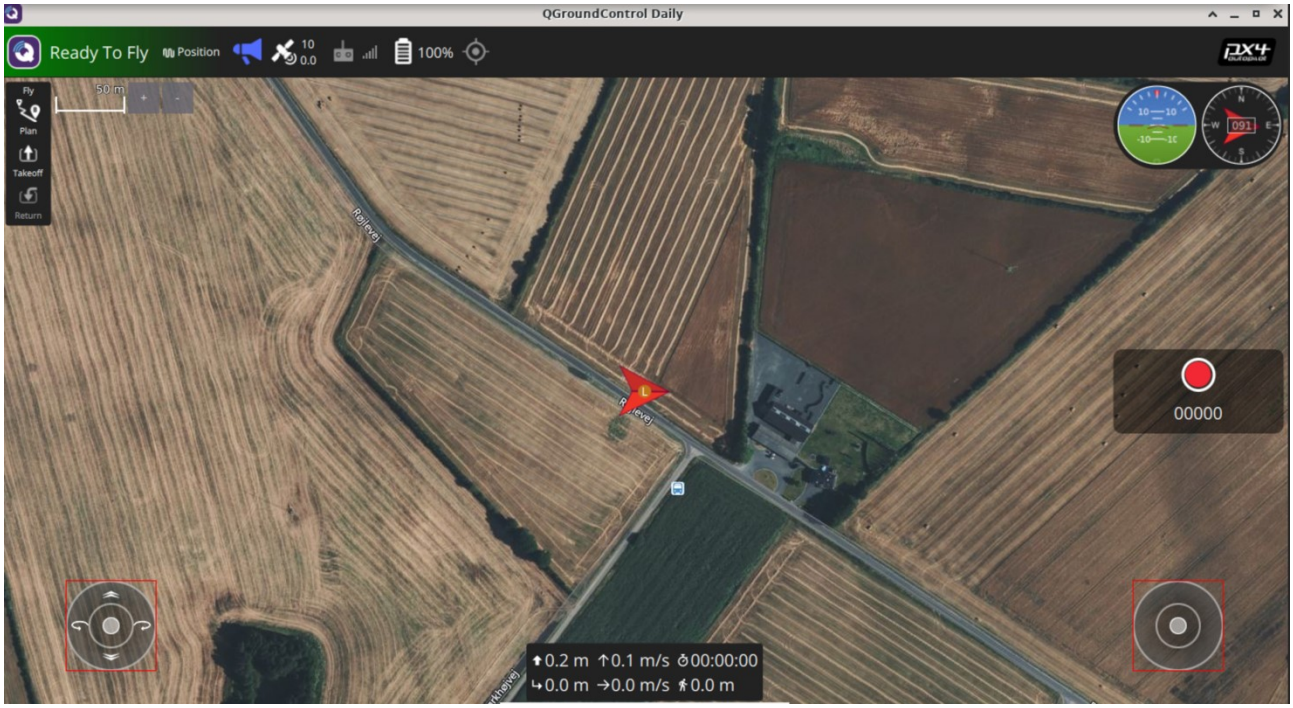

Figure 7 QGroundControl exposed through the Web interface