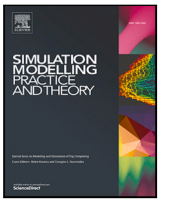
Simulation Modelling Practice and Theory 119 (2022) 102548

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat



Microservices for autonomous UAV inspection with UAV simulation as a service

Lea Matlekovic *, Filip Juric, Peter Schneider-Kamp

Department of Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark

ARTICLE INFO

Keywords:
Cloud robotics
Microservices
Containerization
Kubernetes
CI/CD
UAV simulation

ABSTRACT

Autonomous UAV systems are increasingly touted as the dominant future paradigm for inspecting civil infrastructure. Within this study, we have designed and developed a cloud system for high-level path planning, monitoring, and testing of autonomous UAV missions for inspecting infrastructures such as power lines, power towers, bridges, and railways. The software architecture is based on identified system's functional and non-functional requirements. The cloud system is intended for UAV inspection operators and therefore should support multiple concurrent users. The microservice architecture has assured independence between functionalities, allowing independent scaling resulting in the fast processing time of near-optimal route calculation for UAVs reaching inspection targets. Furthermore, the independence between the services facilitates feature addition and future development. The system robustness is assured by containerizing services and continuously deploying to the Kubernetes cluster distributed across multiple worker nodes. Kubernetes scaling properties have enabled multiple concurrent users regardless of heavy computations for inspection paths. Application load testing has resulted in low processing time when individual services are scaled. Calculated inspection paths are validated for real-world inspection by employing the UAV Gazebo simulation based on a 3D dynamic model with an onboard flight controller, leading the UAV through the waypoints provided by the cloud system. By containerizing the simulation and deploying it within the cluster, we have enabled developers and users to test paths before sending the real-world UAVs to the inspection.

1. Introduction

In recent years, we have seen a tremendous increase in the research and development of applications for autonomous UAVs and robotic systems in general. A recent survey [1] presents industries where UAV solutions are expected to gain a significant market share. According to the survey, a dominance of UAV solutions can be expected in the near future, especially in the civil infrastructure industry. Simultaneously, the European Commission has detected that the rapidly growing transport system in Europe imposes safety concerns and has decided to invest in sustainable solutions for civil infrastructure maintenance [2].

This article contributes to building a sustainable, energy-efficient, and low-cost solution for autonomous UAV infrastructure inspection envisioned in [3]. With that solution, we expect to save time and increase the safety of operators and construction workers during the infrastructure inspection. Enabling continuous infrastructure maintenance will prevent accidents caused by damaged constructions and will ensure public safety. Although many research articles concerning UAV infrastructure inspection have been

* Corresponding author.
E-mail addresses: matlekovic@imada.sdu.dk (L. Matlekovic), fijur20@student.sdu.dk (F. Juric), petersk@imada.sdu.dk (P. Schneider-Kamp).

https://doi.org/10.1016/j.simpat.2022.102548

Available online 21 April 2022

published, no one, to the best of our knowledge, has developed a microservices cloud-based application for infrastructure inspection mission planning, monitoring, and testing.

The previously mentioned survey states the lack of research attention to multi-UAV cooperation for construction and infrastructure inspection applications. Multi-UAV cooperation could provide wider inspection scope, higher error tolerance, and faster task completion time. This article contributes to multi-UAV cooperation research by presenting a high-level path planning and scheduling solution for multiple UAVs inspecting the infrastructure. The survey addresses several challenges in utilizing UAVs for construction and infrastructure inspection:

- Limited energy and short flight time
- Limited processing power
- Limited payload capacities
- Limited localization ability in GPS denied areas

To tackle the challenge of limited energy, our work provides path planning solution only in proximity of power lines, where the UAVs could charge as presented in [4,5]. We take into account limited processing power and provide a framework for offloading heavy computations from the UAVs onboard computer. Microservice architecture and deployment strategy facilitate integrating computation-extensive processes to the cloud, extending the processing power without mounting additional onboard computers on the UAV or consuming energy from the onboard battery. Such processes encompass inspection image analysis using machine learning, 3D structural model reconstruction from collected images, and local inspection path planning which would alternatively run on-board. The concept of using cloud resources for processing computation-extensive tasks is commonly known as cloud robotics. In literature and practice, the cloud robotics concept can often be found implemented in cyber–physical systems. A cyber–physical system (CPS) comprises a large computational system, communication system, and physical elements that interact with the physical world [6]. These systems are part of the broad concept of the Internet of Things (IoT) and represent a cutting-edge technology of today's world [7].

Research related to autonomous UAVs and integration with the cloud can often be found under the term Internet of Drones (IoD). A recent, extensive survey on IoD [8], in addition to previously mentioned challenges, addresses the challenges of cloud connectivity and communication as well as security and safety. Our work assures cloud safety and further contributes to cloud to UAV communication by providing a practical solution based on containerized simulation of a UAV using the ROS2 framework. We use the simulation as a testing and safety tool as it is not possible to observe the vehicle flying beyond the visual line of sight (BVLOS). It provides a safe testing environment for all current and future application developers and users.

Within this article, we analyze functional and non-functional requirements for the autonomous UAV inspection application and propose the software architecture accordingly. To test the viability of the architecture, we developed a proof-of-concept application and implemented a continuous integration and deployment strategy. Prior research concerning cloud architectures in IoD has thoroughly investigated service-oriented architecture (SOA) applications. In SOA, the services communicate through a common communication bus and share storage; they are deployed and operate together. To provide additional flexibility to our application development and deployment, we designed it as a microservice architecture. Unlike SOA, microservices can be containerized, deployed, and scaled independently. The scaling can adapt to the load increase on the specific service. Each service accomplishes only one task and communicates with other services using its own protocol. Service independence enables rapid development and makes the application robust to changes. Due to the heavy computation of inspection routes, we tested the application load for multiple concurrent users taking the advantage of independent scaling. We deployed UAV simulation as a microservice to test the application requirements in a realistic scenario and to monitor the UAV movement after the calculated flight path had been transmitted to the UAV. As a UAV simulator, we employ Gazebo [9], which provides a high-performance physics engine, advanced 3D graphics, sensors simulation, compatibility with ROS/ROS2, and satisfies all the requirements a state-of-the-art UAV simulator should fulfill [10].

This article is organized as follows. Section 2 provides an overview of previous research concerning cloud architecture for UAV mission planning and execution stating the differences of our approach. Requirements and the microservice application architecture are presented in Section 3. In Section 4 we describe the application implementation, deployment strategy, and DevOps tools used for continuous integration and application deployment. Section 5 analyzes the application performance qualitatively, as well as quantitatively through load test results. We conclude in Section 6 discussing possible future work.

2. Related work

There has been growing interest in research of IoD and UAVs in general. In the last years, some researchers have proposed IoD system architecture solutions as a whole [11] while others have focused on the IoD cloud architecture [12]. We conducted extensive research on literature focused on the cloud architecture of IoD systems as well as other robotics systems. A noticeable cloud architecture trend is SOA, although a few latest works show increasing interest in microservices. We have summarized related work's findings and have stated the differences.

A literature overview of the service-oriented architecture and Web services used for integrating robotics systems with the cloud is presented in [13]. The overview focuses on robotic systems' integration with the cloud and analyses solutions proposed since the trend emerged in 2010 until 2019. The author categorizes research in service-oriented robotics computing into two aspects: virtualization and computation offloading. Virtualization provides seamless access to robots through service interfaces. Computation offloading uses service interfaces to offload computation-extensive tasks to the cloud. Most of the solutions use ROS as a robot

operating system and UAV to cloud connection is established through middlewares like rosjs and rosbridge, rosPHP API, ROS Web services, or ROSLink. A user interacts with the cloud through the REST or SOAP Web services using the HTTP protocol or Websockets to receive real-time updates from the UAV. MAVLink seems to be the most common protocol for exchanging messages between UAV and the cloud. An interesting solution is proposed in [14]. The article presents SOA architecture, achieving constant cloud connection to the robots and enabling developers remote development and control of UAVs. The same author later published an article where the solution was implemented on the MyBot robot [15]. Although the platform serves a different purpose than ours, there are similarities in the approach to developing a modular, extensible cloud platform. However, the article neither provides a decentralized strategy for cloud services development nor continuous integration, containerization, and continuous deployment strategy. The same author with collaborators later developed Dronemap Planner [16]. Dronemap Planner (DP) is a service-oriented cloud-based drone management system. The system controls, monitors, and communicates with drones over the Internet. DP allows seamless communication with the UAVs enabling their control from anywhere without restriction on distance. In addition, DP provides access to cloud computing resources to offload heavy computations from UAVs. It virtualizes the access to UAVs through Web services (SOAP and REST), schedules their missions, and promotes collaboration between them. The article describes and states the usefulness of traveling salesmen (TSP) and multiple traveling salesmen problem (MTSP) solving for UAVs' high-level path planning. It defines Cognitive Engine (CE), a cloud component where Dynamic Mission Planner (DMP), solving traveling salesman problem, should be implemented. However, it neither provides details of implementation nor shows any experimental results using DMP. Also, the system is not developed in microservices and does not provide continuous integration and continuous deployment strategy. In our work, by implementing small services and automatizing integration and deployment, we expect to speed up any future upgrades to the application and take advantage of individual scaling to enable multiple concurrent users. Similar to our approach, the DP uses Gazebo simulation to test the application. The difference in our implementation is using the ROS2 for running our simulation model instead of ROS. The main advantage of ROS2 is distributed discovery system using the Data Distribution Service (DDS) standard. In comparison to ROS which uses centralized node discovery, DDS provides a decentralized approach eliminating the risk of communication failure [17]. Furthermore, DDS provides security enhancements by defining a Service Plugin Interface (SPI) architecture. SPIs enable authentication, access control, and encryption of data exchanged within the ROS2 system [18]. Our simulation is containerized and treated as a microservice. If the developers want to test new features with simulated UAVs, they do not have to run Gazebo locally. Instead, the Gazebo simulation is deployed in the cloud and can be reached through the web browser.

Authors in [19] recognized the gap in cloud robotic systems development and proposed a microservices-based cloud robotics system. As the authors state, microservices enable faster development and feature addition, especially when continuously deployed to production. In conclusion, they express microservices importance in the future cloud robotics systems and predict an increase in research on microservices for the development of intelligent robots. Their cloud platform performs Simultaneous Localization and Mapping (SLAM) and it is tested in the simulation environment as well as in real-life using Turtlebot mobile robot. The article presents a similar approach to our work; however, the application is not suitable for autonomous inspection purposes as it limits the distance between the cloud and the robot by connecting both on the same WIFI network. Since the implementation considers an indoor robot, it does not employ Simulation as a Service. The implemented solution has not been validated with load testing. Similar is proposed in [20], where authors use Docker for service isolation and Kubernetes for service deployment and management. The work also does not include Simulation as a Service. The article fails to present load testing results, although it mentions that microservices architecture is originally a solution for high-availability and high-performance demands.

A system for UAV monitoring and management is proposed in [21] presenting application performance results. The application is developed as a monolith and deployed on the Heroku cloud platform. The system is implemented in a client–server architecture. The client provides a user interface for UAV monitoring and requests UAV sensors updates from the server. Modules implemented on the server-side provide HTTP connection listening, connection to the database, and collision avoidance. The authors use UAV sensor data stored in the database to simulate UAV flight. Their solution neither implements flight planning nor tests flight simulation; it only provides UAV monitoring.

UAV mission planning web application communicating with UAV simulation is presented in [22]. The mission is planned using Mission Planner open source software where the user selects waypoints generating the flight path. DroneKit library is used for communication between the web server and drone server. The drone server runs software-in-the-loop (SITL) simulation, and the movement is visible on the leaflet-based web interface. Although the architecture is depicted in a modular way, there is no mention of microservices, and it seems the modules are not entirely independent. The system is implemented locally using physical and virtual machines and it is missing the deployment strategy. Therefore, the solution does not support multiple concurrent users using the application as a service. Another notable advantage of our work is the implementation of the mission planner for route optimization and determination of the inspection target visiting order. The UAV simulation is not only SITL, but it includes the vehicle dynamics by simulating the UAV model in the Gazebo simulator.

The authors in [23] implemented a mission definition system for automated infrastructure inspection UAV flights. The system calculates flight paths for specifically shaped infrastructure, e.g., high voltage towers and telecommunication antennas. From the system design presented, the architecture reminds of microservices, but it is not specified. The article neither considers deployment to the cloud nor deployment strategy. Later the same authors propose a microservice architecture for drone-as-a-service [24]. They present the requirements for the architecture and design but do not describe the implementation in detail. Once the mission is defined, the system calculates a set of possible options to perform it. It generates flight plans and determines the resources needed for execution. Although the article mentions calculating optimal flight plans, it does not consider determining the order of visits for multiple UAVs. The application is developed for mission planning and automated flights, but it does not assume continuous

Table 1
Comparison of our approach to similar implementations of cloud UAV applications.

	[16]	[19]	[20]	[23,25]	Proposed solution
Microservices	–	✓	✓	✓	✓
CI/CD	–	✓	✓	–	✓
Cloud deployment	✓	✓	✓	✓	✓
Docker	–	✓	✓	✓	✓
Kubernetes	–	✓	✓	–	✓
Simulation	✓	✓	✓	✓	✓
ROS2	–	–	–	–	✓
MAVLink	✓	–	–	✓	✓
Inspection purpose	–	–	–	✓	✓
High-level path planning	✓	–	–	✓	✓
Route calculation	–	–	–	–	✓
Load test	–	–	–	–	✓

autonomous flying. The pilot is in charge of executing the flight plan either manually or by sending the MAVLink application-generated message to the UAV. The authors' most recent publication [25] upgrades the system with simulation and evaluation services. The UAV simulation consists of a series of differential equations modeling UAV dynamics, and it is used for platform evaluation. Our implementation uses Gazebo simulation as a service to evaluate the platform and to test the UAV behavior. The UAV model is based on the real vehicle and can be chosen depending on the vehicle used for inspection. Simulated UAV runs all the same software we expect to deploy on the real UAV, i.e., PX4 low-level controller and ROS2 communication solutions. A solution similar to drone-as-a-service is proposed in [26] with a focus on providing a monitoring solution by enabling telemetry and video streaming from the UAV to the cloud. However, the solution is not well adapted for autonomous inspection purposes. The paper proposes a high-level overview of microservices for communication with UAVs and does not further provide implementation details.

Service-oriented architecture in cloud robotics has been thoroughly investigated. Recently, the interest in microservices has been growing. We identified the lack of research concerning continuous integration and continuous deployment of microservices in cloud robotics. Using these strategies, the microservice architecture can reach its full potential. The related work also lacks automated scheduling solution implementations for multiple UAVs as well as a testing environment representing the real world with UAV simulation based on the UAV dynamic model. Previously proposed solutions have not provided an application suitable for multiple concurrent users calculating routes and have failed to evaluate the cloud system's load support. A comparison of our approach to similar, previously mentioned implementations is provided in Table 1.

3. Requirements and software architecture

In this section, we present functional and non-functional requirements for autonomous UAV mission planning, monitoring, and testing application as well as accordingly designed software architecture. Functional requirements specify application desired behavior and user interactions. Non-functional requirements specify criteria that assure seamless operation of the system during the development and user interaction.

3.1. Functional requirements

The main actors interacting with the application are developers and application users such as UAV operators. The developers should be able to easily add features; without spending time on understanding the codebase. They should not be concerned with application deployment. Therefore, the deployment should be automatized. However, it should be easy to test new features by interacting with previously implemented functionalities and UAVs. 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 UAV participating in the mission. The routes should be visualized on the 2D map and sent to the UAVs as waypoints used for navigation. The operator should be able to monitor the inspection progress and receive alerts in real-time. Inspection data and UAV 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.

3.2. Non-functional requirements

We identified scalability, interoperability, modifiability, security, and performance as the application's non-functional requirements. When load increases, the application should scale to adapt to the load and uninterruptedly continue working. Interoperability describes the application's ability to interact and exchange data between components within the system as well as with the outside systems. For autonomous UAV inspection mission planning, the application should be able to exchange information between components and communicate with the UAV systems without compromising data. 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 application is robust to changes. The application should perform to satisfy the requirements and to do so efficiently. We expect the application's growth in features and increased traffic in the near future. These requirements, when met, will facilitate the application's growth.

3.3. Software architecture

Based on functional and non-functional requirements, we designed the application in microservice software architecture and automatized the delivery process. Microservice architecture, also known as Microservices, 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. Monolithic applications encompass several tightly coupled functions and tend to have a big codebase. The application is developed, packaged, and deployed as a standalone instance; therefore, the development does not require advanced architecture planning. 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 improving the code readability, searchability, and facilitating the development and maintenance. Services can be scaled independently and automatically, depending on the load. They can use different technology stacks, including programming language and data storage. However, service independence brings more complexity in application deployment compared with monolithic applications. Numerous practices and tools have been developed to facilitate the testing, integration, and deployment of microservices. DevOps is a combination of practices and tools designed to facilitate the applications' delivery. It aims to increase the ability to deploy applications faster by merging the development and operation tasks and automating the delivery process. 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 builds and tests run there before proceeding to production. Remote repository platforms with built-in version control facilitate the collaboration between developers and enable continuous integration. These platforms also integrate with various DevOps tools to enable continuous delivery and deployment. 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 processes can vary in complexity depending on the number of services, test requirements, and in general, the deployment strategy.

The services are designed as follows:

- *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 UAV. The user can store and retrieve mission plans or execute them by sending the calculated navigation data to the UAVs.
- *Towers service* - the service extracts power towers and power lines locations as points and lines from Open Street Map [27] 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.
- *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.
- *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.
- *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 UAVs. The new graph creation should run only when new locations are added to either towers, railways, or bridges databases.
- *Routing Solver service* - the service receives inspection targets from the user via Missions service and UAV locations from the vehicles. It determines which UAV should inspect which target based on the shortest paths requested from the A* Pathfinder service. The requests to the A* pathfinder are sent asynchronously to enable the faster calculation and take advantage of pod scaling in Kubernetes. The service returns the path for each UAV 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.
- *A* Pathfinder service* - the service requests a graph from the Graph service and performs the A* algorithm on the targets and UAVs locations received from the Routing Solver service. It returns the shortest path between a target and a UAV 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.
- *No-fly service* - the service contains areas where UAV 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.
- *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 UAV. Upon the user's request, the mission routes and tasks are sent to the UAVs 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 UAV conducts an inspection.
- *Drone Log service* - the service receives data from the UAVs and stores it in the database. The data consists of telemetry from UAV sensors. For monitoring purposes, the service uses Drone Estimator to estimate the UAV location and reports it to the Web interface. The location is updated with UAV measurements every time they are received.

- *Drone Estimator service* - the service estimates the UAVs' location in periods between the location data is received from the Drone Log service. Using the estimations, the user can monitor the UAV estimated movements on the Web interface while the mission is in progress. Without the estimator, UAV movements would be visualized only when received directly from the UAVs, which can be every 10 s or more. Estimations are sent back and stored in the Drone Log database.
- *Message Broker service* - the service enables communication with UAVs and manages the data distribution from the cloud to the UAVs as well as from the UAVs to the cloud. It keeps track of the successful delivery of messages and assures the distribution based on priorities.
- *UAV Simulation* - the service runs a UAV 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 UAVs.

Services are containerized and deployed in the Kubernetes cluster which provides service management and independent scaling to achieve faster route calculations and support for multiple concurrent users. The communication within the cluster is secure by default, using the Transport Layer Security (TLS) protocol. Necessary certificates are created and distributed to the cluster components during the installation. Access to the cluster is possible through the Kubernetes API and exposed components. By configuring role-based access control (RBAC) policies, we have secured the Kubernetes API to respond only to requests that it can properly authenticate and authorize. Exposed components only communicate through HTTPS, where the traffic is secured via SSL/TLS encryption. Any incoming HTTP requests are automatically upgraded to HTTPS. The SSL certificates are automatically issued and updated using Let us Encrypt [28]. The web interface is secured using HTTP basic authentication, while all other services rely on a standard HTTP bearer token authentication scheme. This prevents any unauthorized use of our system.

Fig. 1 depicts Microservice architecture and communication flow. The user selects inspection targets on the Web interface, which creates a POST request to the Missions service. The request contains target locations. Mission service has UAV 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 UAV location. Calculated routes are stored in the Missions service, visualized on the Web interface, and fetched by the UAV Simulation through the Message Broker. While the mission is in progress, the Message Broker receives UAV telemetry data and sends it to the Drone Log service, where the data is stored. If the communication between a UAV and the Message Broker is temporarily interrupted or a UAV is in the GPS denied area, the Drone Estimator service provides estimations for the UAV location and status. The UAV will continue its mission and regularly poll for further instructions until the communication has been re-established. Estimation data is stored in the Drone Log service database for mission simulation after the mission finishes. The Message Broker updates the Missions service database with inspection results.

4. Implementation

In this section, we describe simulation implementation details and pipeline stages for application deployment in Kubernetes.

The UAV simulation runs in a Docker container within the Kubernetes cluster. We have 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 [29]. The image uses VNC to view the desktop environment from the outside of the container. We have used it to visualize the building process and to verify that the simulation runs successfully. On the base image, we have installed Gazebo to simulate the Iris Quadrotor 3D model. The vehicle model is a dynamic model based on the body's mass and moments of inertia. For UAV low-level control we use PX4 Autopilot [30]. For on-board control, we have installed ROS2 Foxy Fitzroy distribution and a UAV point-to-point flight controller. To integrate PX4 and ROS2 and enable message exchange, we use the default middleware Fast DDS. The UAV reports its location to the cloud every 10 s and requests route waypoints every 5 s. The Mission service provides route waypoints only when the UAV unique identifier coincides with the unique identifier of the UAV requesting the route. The UAV receives flight waypoints through a MAVLink message. The simulation is exposed using the Gzweb Gazebo Web client and can be reached through the web browser.

In order to deploy the application automatically every time the changes are made, we have set up a pipeline employing DevOps tools and technologies. We use GitLab [31] 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 [32]. We have set up the K3s Kubernetes cluster [33] 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 have integrated the cluster with GitLab by providing the cluster API and token. To be able to run the CI/CD jobs, we have 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 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.

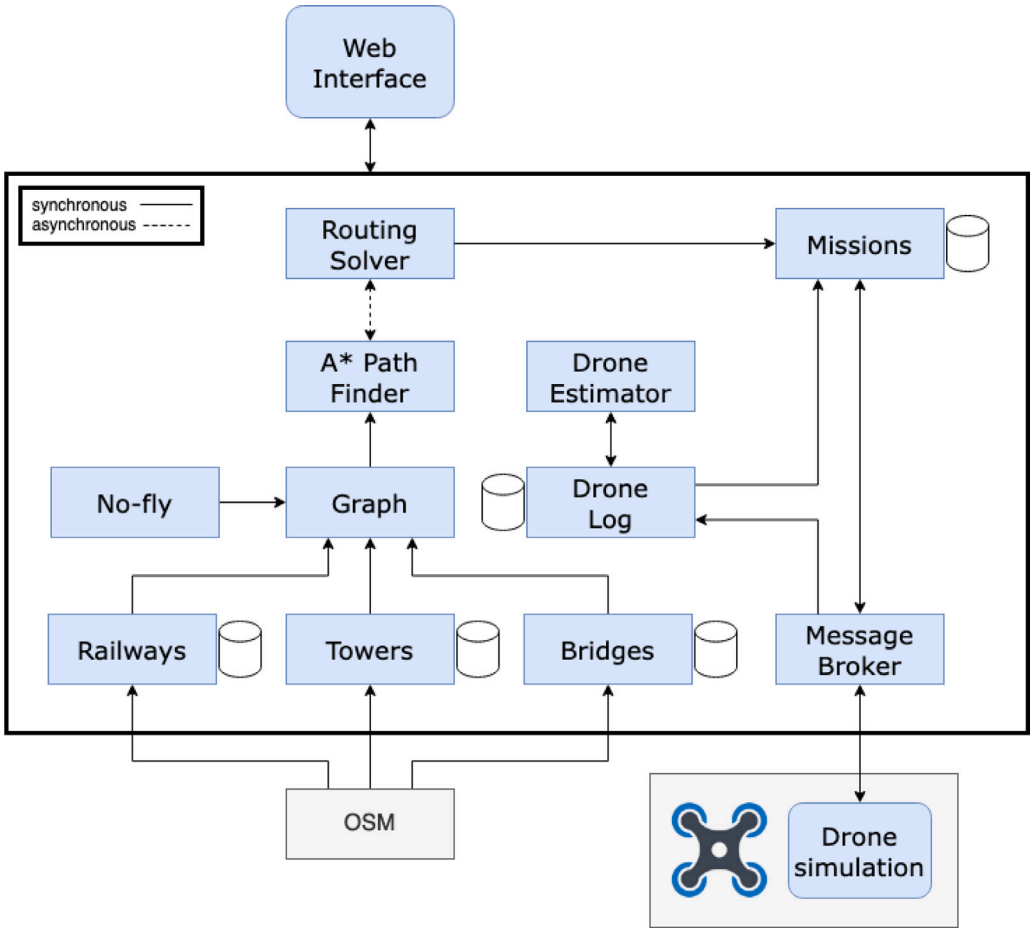


Fig. 1. Microservice application architecture. The arrows show the data flow within the system. Dashed line represents request sent asynchronously while solid lines represent requests sent synchronously. All the requests within the cluster are HTTP, relying on cluster's default TLS protocol while external requests are secured by HTTPS.

5. Performance evaluation

In this section, we have conducted the application's performance evaluation by validating functional and non-functional requirements identified in Section 2. We obtain qualitative and quantitative results demonstrating the application's functionalities, performance under the load and scaling properties as well as UAV simulation validity.

5.1. Qualitative performance

We present the qualitative results which satisfy the application's functional requirements. Fig. 2 shows a 2D map on the Web interface where the user selects target infrastructure for inspection. Blue circles represent power towers and green circles are selected towers for inspection. Orange circles show UAVs waiting for the routes to start with an inspection. When the user initiates route calculation, cloud services receive targets and UAV locations and return calculated routes as described in Section 3. Fig. 3 shows calculated routes on the Web interface. The result of the route calculation is a set of waypoints the UAVs have to traverse to reach the targets. For clear visualization, the results are shown on the Web interface as solid red lines passing through the calculated waypoints. The user can name the mission and store the waypoints in the database.

The UAV simulation service requests the route data with a coinciding unique identifier from the Mission service database and sends it to a UAV. The UAV starts flying from point to point, based on the waypoints calculated in the cloud. Fig. 4 shows UAV simulation deployed as a service and visualized in the Web browser.

We have validated the simulation by selecting a target, calculating the route, and sending it to the simulated UAV. The message fetched by the UAV contains calculated waypoints as an instruction for the onboard computer to follow. As a target, we have selected a bridge around the city of Odense, Denmark. Fig. 5 shows the calculated route for a UAV to reach a bridge both on a map view and a satellite view. The algorithm assures that the UAV flies in the proximity of the power lines for as long as possible,

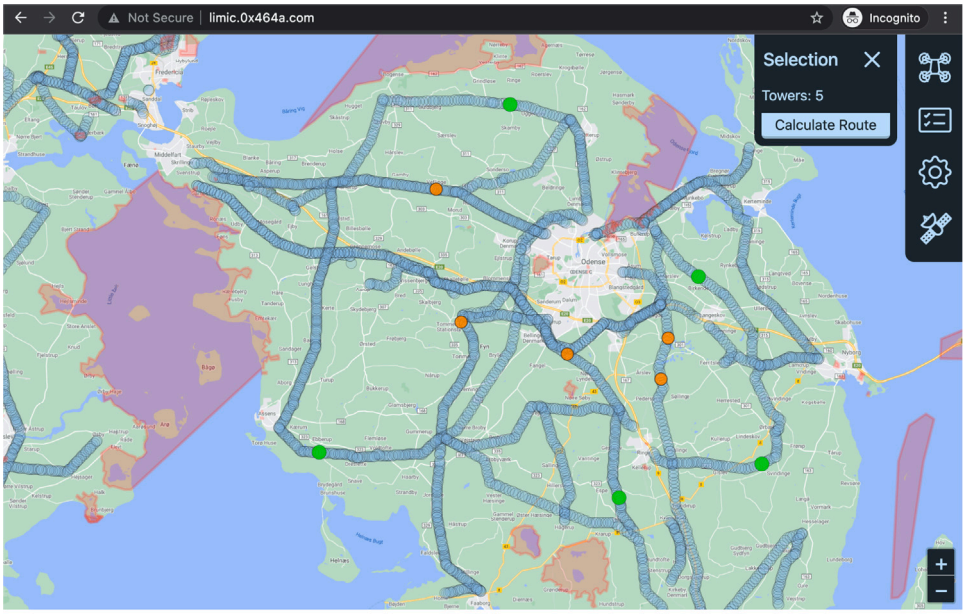


Fig. 2. Web interface showing inspection targets (blue), selected targets (green), and UAVs (orange). The red areas represent no-fly zones where UAV flight is not permitted.

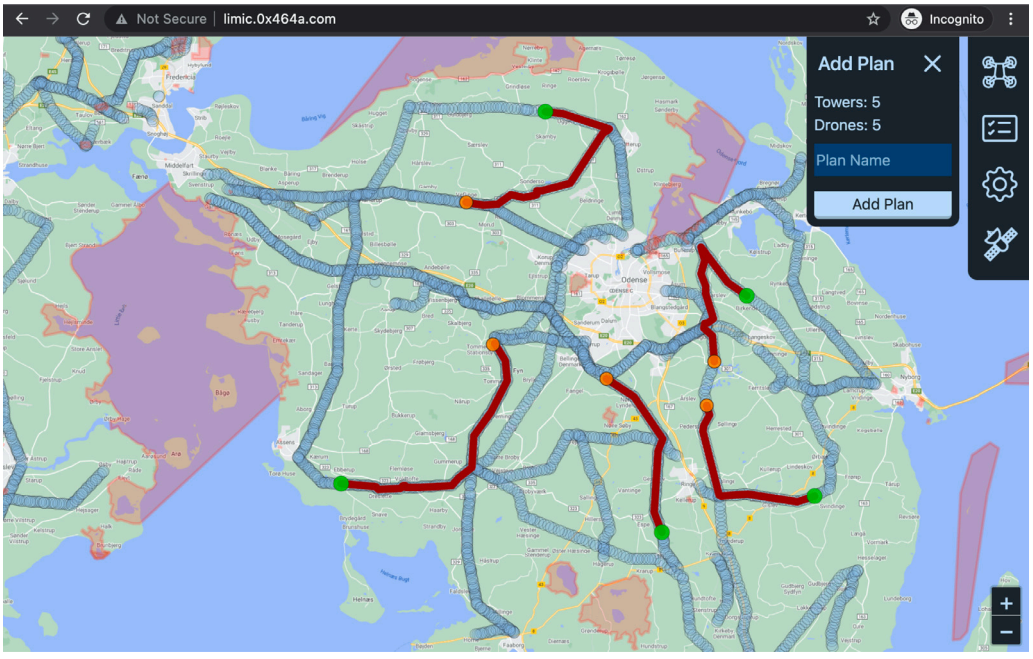


Fig. 3. Web interface showing calculated inspection routes in red, for each UAV reaching a target.

therefore resulting in a non-optimal path in terms of flight distance and time. Such a decision has been made to facilitate obtaining permission for autonomous UAV inspection from regulation bodies as it assures that UAVs will stay near the infrastructure with minimal interruption of residential areas resulting in the elimination of unpredictable flying. In the graph and on the Web interface, the bridges are represented as approximate center points of the bridge polygon extracted from the Open Street Map to facilitate the target selection and route calculation. Each bridge node is connected to the closest power tower in a radius of 3 kilometers and other bridges in a radius of 5 kilometers. This choice has been made as an alternative to connecting each node in the bridge polygon to the closest power tower nodes which would increase the processing time for graph creation. We only use approximate center nodes

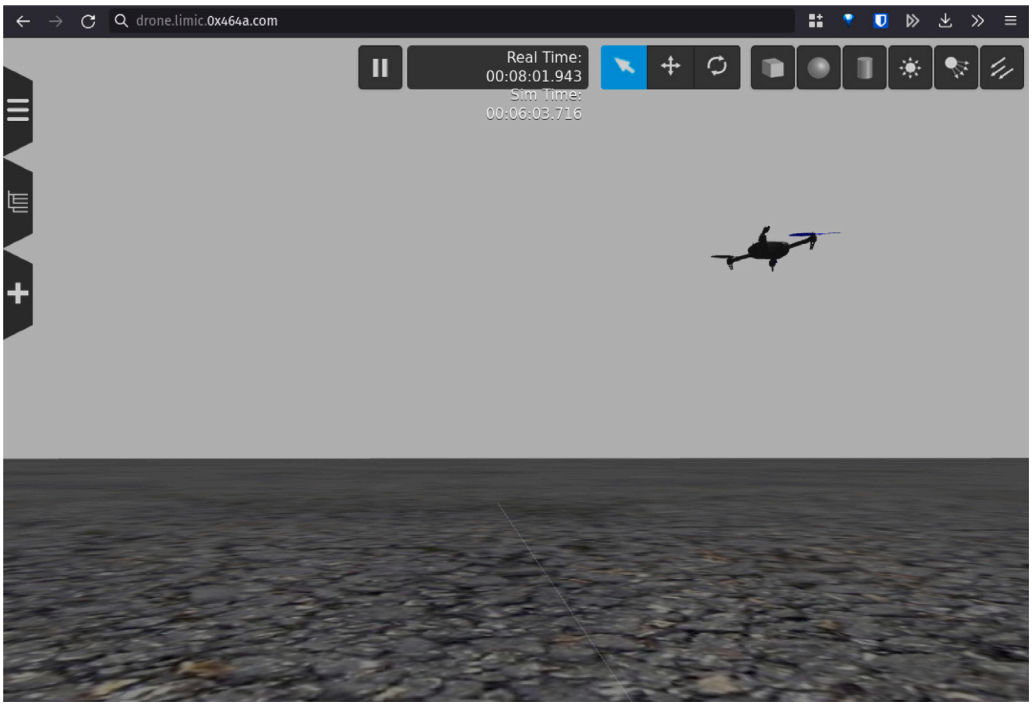
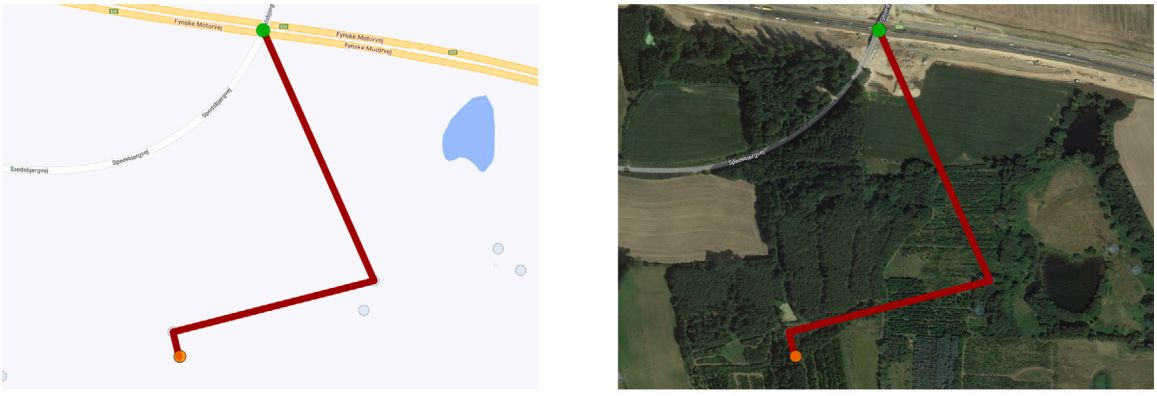


Fig. 4. UAV simulation deployed as a service and visualized in the Web browser.



(a) Map view (b) Satellite view

Fig. 5. Route calculation for a UAV reaching a bridge. The UAV is shown as an orange circle and the bridge is shown as a green circle. Calculated route contains four waypoints, including the UAV's starting position and bridge location.

for high-level route calculation to reach the target, while detailed bridge inspection requires low-level path planning. Therefore, we store the bridge polygon locations which can support low-level inspection path planning and pass it to the UAV. As the UAV receives the flight instructions, it starts flying in simulation and the movement is visualized on the web interface. We have recorded the flight data and have used it to visualize the executed route. The waypoints sent to the UAV and executed flight path are shown in Fig. 6. By recording and visualizing the flight data from the simulation, we have validated the communication between the cloud and the UAV as well as the onboard flight controller as the UAV reached all the waypoints calculated in the Routing Solver service.

5.2. Quantitative performance

We present the quantitative results which satisfy the application's non-functional requirements. We have set up experiments to test the application's ability to support increasing and decreasing demand for route calculation. The experiments suppose that the application will be used by many concurrent users, i.e. inspection operators, when planning autonomous infrastructure inspection

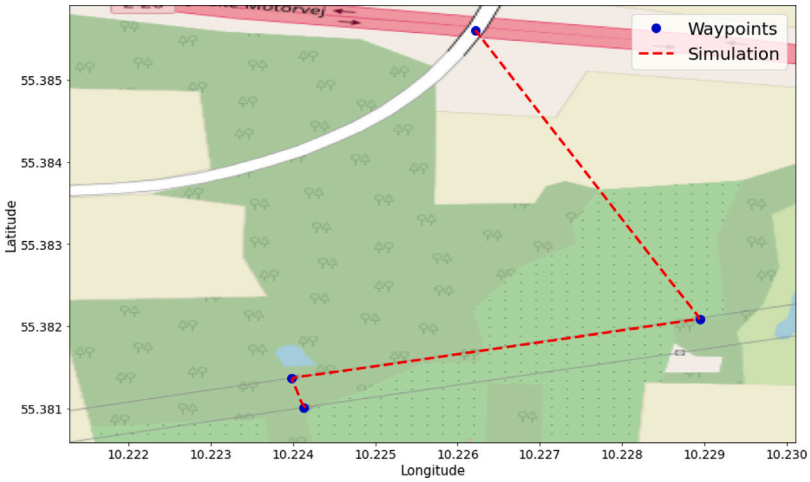


Fig. 6. The route visualized from the data collected from the UAV simulated flight. Waypoints which the UAV received from the cloud are shown as blue circles. A red dashed line shows the path traversed by the simulated UAV.

missions. When a user selects inspection targets on the Web interface, a request for route calculation is sent to the Routing Solver service. For simulating users sending POST requests to the Routing Solver service, we use Locust Python load testing framework [34]. Locust framework generates requests based on the desired number of users interacting with the application and increases/decreases that number with a predetermined rate. The POST requests to the Routing Solver service contain UAV and inspection target locations.

We have conducted load testing to investigate how microservice architecture and deployment strategy impact the system's performance while increasing and decreasing the demand for the route calculation. Four different test scenarios are evaluated:

First scenario - the scenario is designed to compare the system's performance depending on the number of pod replicas of Vehicle Routing and A* Pathfinder services. The first simulation mimics the system's performance as if it was not deployed in Kubernetes and scaled. We deploy only one pod of each service and simulate the increasing demand of up to 300 users after which the demand decreases. In the second simulation, we increase the number of Vehicle Routing and A* Pathfinder pod replicas to 20 and simulate the same user behavior. The waiting time between requests from the individual user varies from 1 to 5 s. The user behavior proceeds as follows:

- 10 users per second are added until the load reaches 100 users over a period of 1 min;
- 20 users per second are added reaching 200 virtual users over a period of 1 min;
- 20 users per second are added reaching 300 virtual users over a period of 1 min;
- 20 users per second are removed until the load reaches 200 virtual users over a period of 1 min;
- 50 users per second are removed until the load reaches 100 virtual users over a period of 1 min;

Fig. 7 shows the load test results. We monitor the number of requests the system is able to receive in a second and the median response time it takes to calculate a route and return it to the user. The objective is to assure a low response time even when the demand is high. That can be achieved only when the system supports a high number of requests per second. It is visible from the figure that for 1 pod deployment requests per second do not increase as the system does not have resources to support them. Instead, the unresponded requests accumulate, waiting to be served and causing long response times. That is the behavior we would also expect if the application was developed in monolithic architecture and deployed without scaling. Although the monolith applications can be scaled, the architecture does not allow the scaling of the specific parts for which the expected demand is high. They can only be scaled as a whole, resulting in unnecessary resource usage which increases costs, especially when using third-party cloud platforms. By increasing the number of pods in the cluster, the system is able to support more requests per second, keeping the response time low and stable.

Second scenario - the scenario is designed to compare the system's performance for an increasing load of up to 1000 active users. We test the supporting load when the system is deployed with 20 Routing Solver and A* Pathfinder pod replicas as well as 40 pod replicas. The waiting time between the individual user's requests is 1 s. The user waits 1 s after the response to the previous request has been received to send another request. The user behavior proceeds as follows:

- 10 users per second are added until the load reaches 100 users over a period of 1 min;
- 20 users per second are added reaching 300 virtual users over a period of 1 min;
- 20 users per second are added reaching 500 virtual users over a period of 1 min;
- 20 users per second are added reaching 700 virtual users over a period of 1 min;
- 50 users per second are added reaching 1000 virtual users over a period of 1 min;

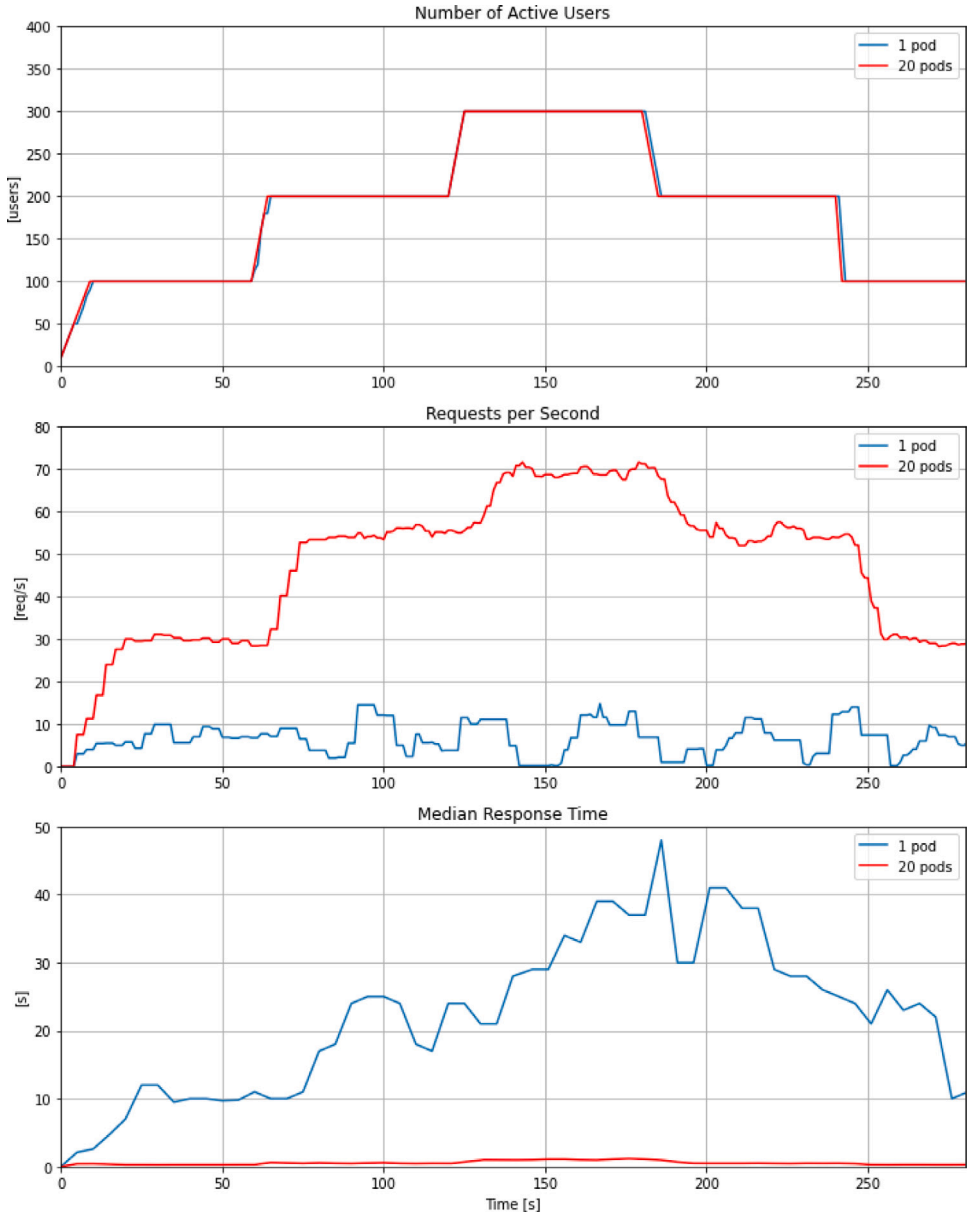


Fig. 7. Load test results for simulation of increasing and decreasing demand up to 300 users with only 1 pod replica of each service deployed (blue) and with 20 pod replicas of Routing Solver and A* Pathfinder services deployed (red).

Fig. 8 shows the load test results. As the waiting time is constant and short, the number of requests the system with 20 pod replicas receives per second is higher compared to the first scenario with 20 pod replicas deployed. The simulated scenario does not represent a real user behavior as it assumes that each user sends requests with high frequency. However, the scenario shows the dependence of the supported number of requests per second and median response time to the number of deployed pods. As we increase the number of pods from 20 to 40, the system response time stabilizes faster when more users are added. By sending requests with high frequency and from the increasing number of users, the system reaches its limit of requests sent per second. For 20 pods deployment, the system is able to support around 75 requests per second while for 40 pod deployments that number is increased to around 95. To increase the support for even more requests, more pod replicas should be deployed. As the cluster CPU resources are limited, there is a limit of pod replicas that can be deployed. To increase that limit, it is possible to add more nodes to the cluster.

Third scenario - the scenario tests the system support for increasing the number of users up to 1000 after which the demand decreases. The request frequency is decreased to represent a more realistic scenario. Each user sends requests 10 s after the previous

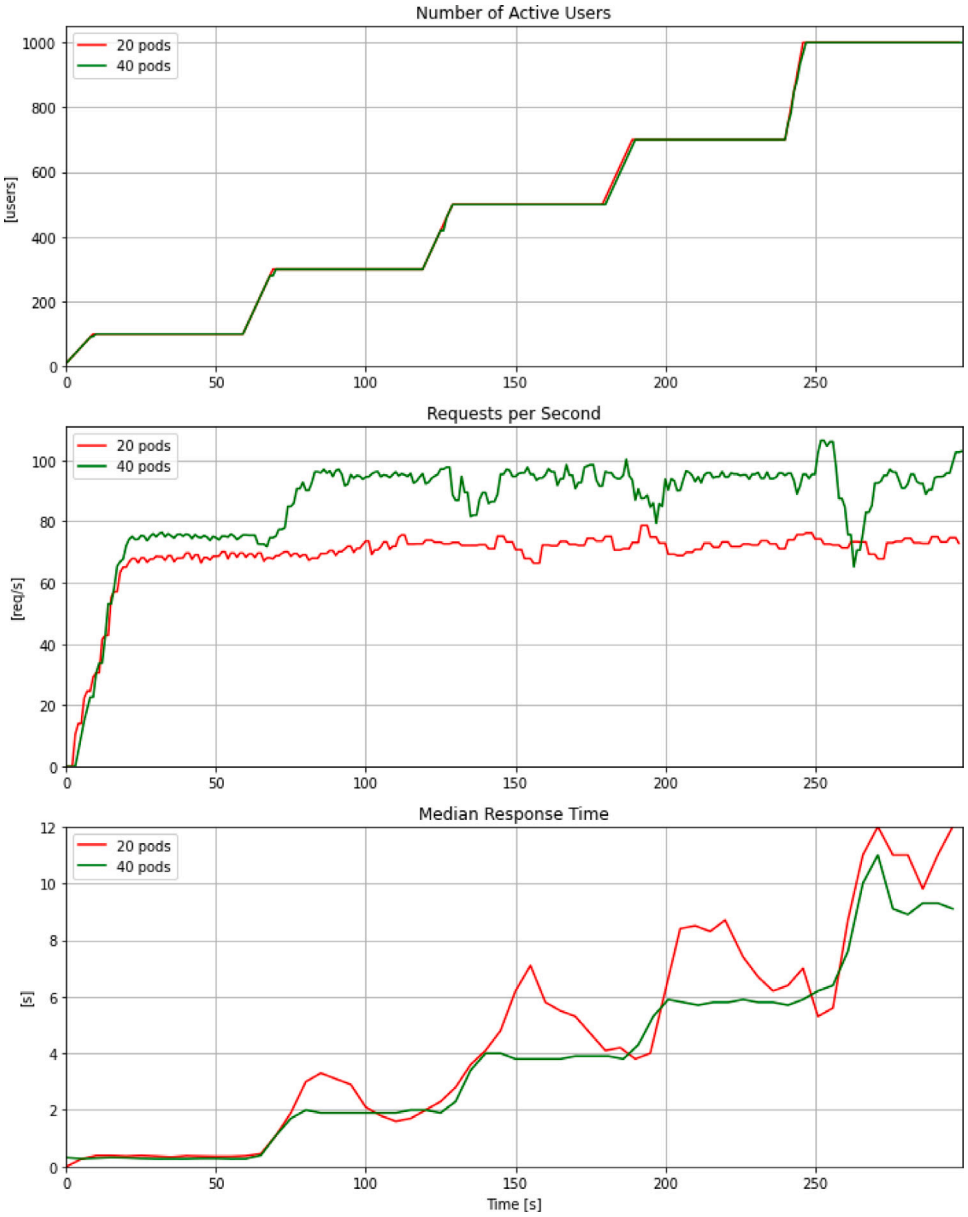


Fig. 8. Load test results for simulation of increasing demand up to 1000 users with 20 pod replicas of Routing Solver and A* Pathfinder services deployed (red) and with 40 replicas deployed (green).

request has been responded to. We deploy 20 pod replicas of Routing Solver and A* Pathfinder services and compare them to 100 pod replicas. The scenario starts with adding users as in the second scenario, after which the users are removed as follows:

- 50 users per second are removed until the load reaches 700 users over a period of 1 min;
- 20 users per second are removed until the load reaches 500 users over a period of 1 min;
- 20 users per second are removed until the load reaches 300 users over a period of 1 min;
- 20 users per second are removed until the load reaches 100 users over a period of 1 min;
- 10 users per second are removed until the load reaches 0 users over a period of 1 min;

Fig. 9 shows the load test results. As the waiting time between requests is increased, the pressure on the system is lower than in the second scenario. This results in the faster stabilization of the response time for 20 pods deployed when the load increases. With only 20 pods deployed, the system begins experiencing difficulties responding to requests when the number of users is rising to 700 and 1000, resulting in longer response times. The same peak is visible every time there is an increase in demand, but with 100 pods

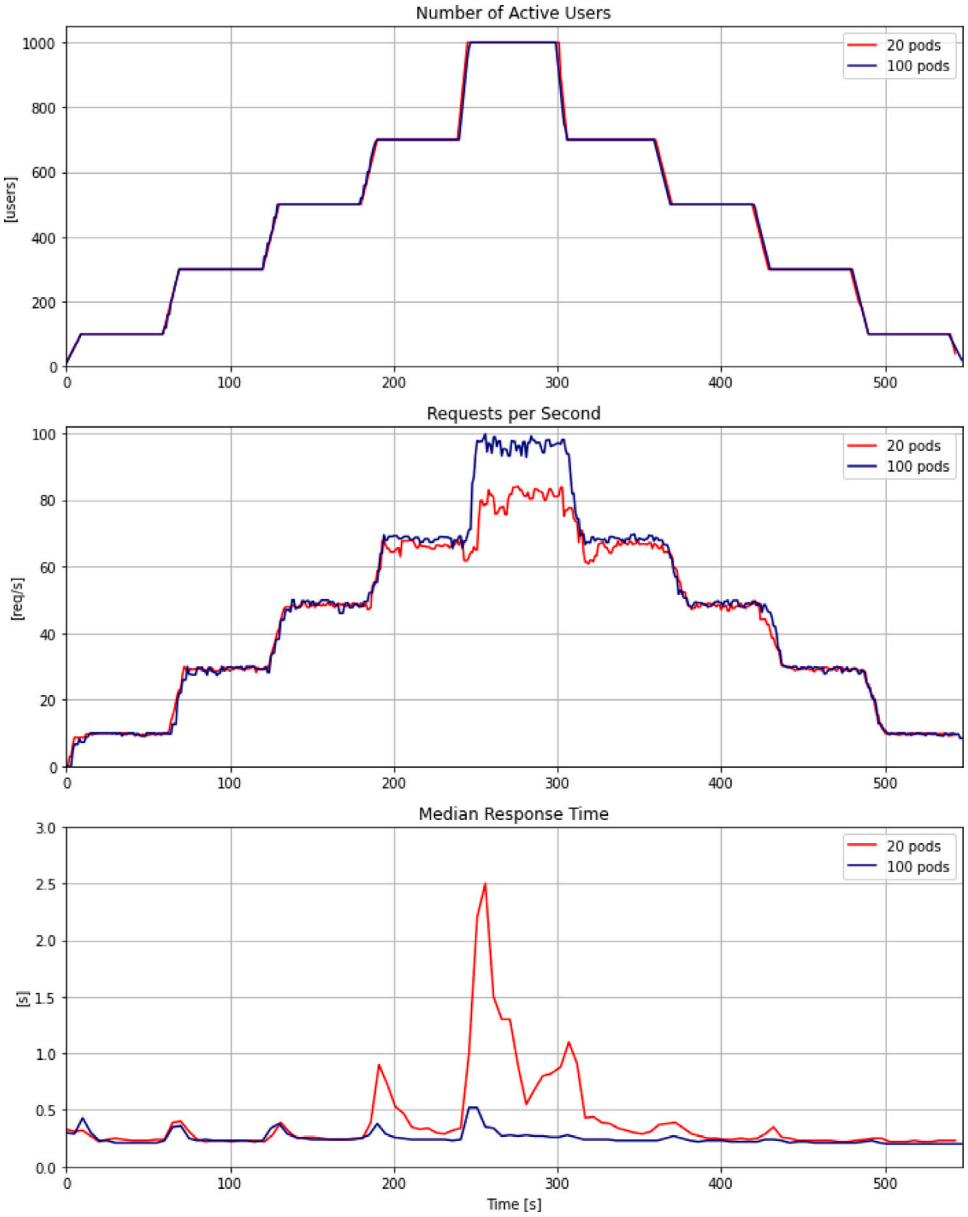


Fig. 9. Load test results for simulation of increasing and decreasing demand with up to 1000 users with 20 pod replicas of Routing Solver and A* Pathfinder services deployed (red) and with 100 replicas deployed (navy).

deployed the system is able to serve the requests faster so the response time is not drastically increased. As expected, the more pods we deploy, the system is able to respond to more requests per second without compromising the response time.

Fourth scenario - the scenario tests the system's limits with high user demand. The number of users is increased to 5000 with waiting time between the requests of 10 s. The scenario proceeds as follows:

- 50 users per second are added until the load reaches 1000 users over a period of 100 s;
- 50 users per second are added until the load reaches 2000 users over a period of 100 s;
- 50 users per second are added until the load reaches 3000 users over a period of 100 s;
- 50 users per second are added until the load reaches 4000 users over a period of 100 s;
- 50 users per second are added until the load reaches 5000 users over a period of 100 s;

We test the scenario with 100 Vehicle Routing and A* Pathfinder pods deployed. Although the cluster can deploy more than 100 pods, using all the resources impacts the performance and raises response time. Increasing the frequency of user addition to 50 has

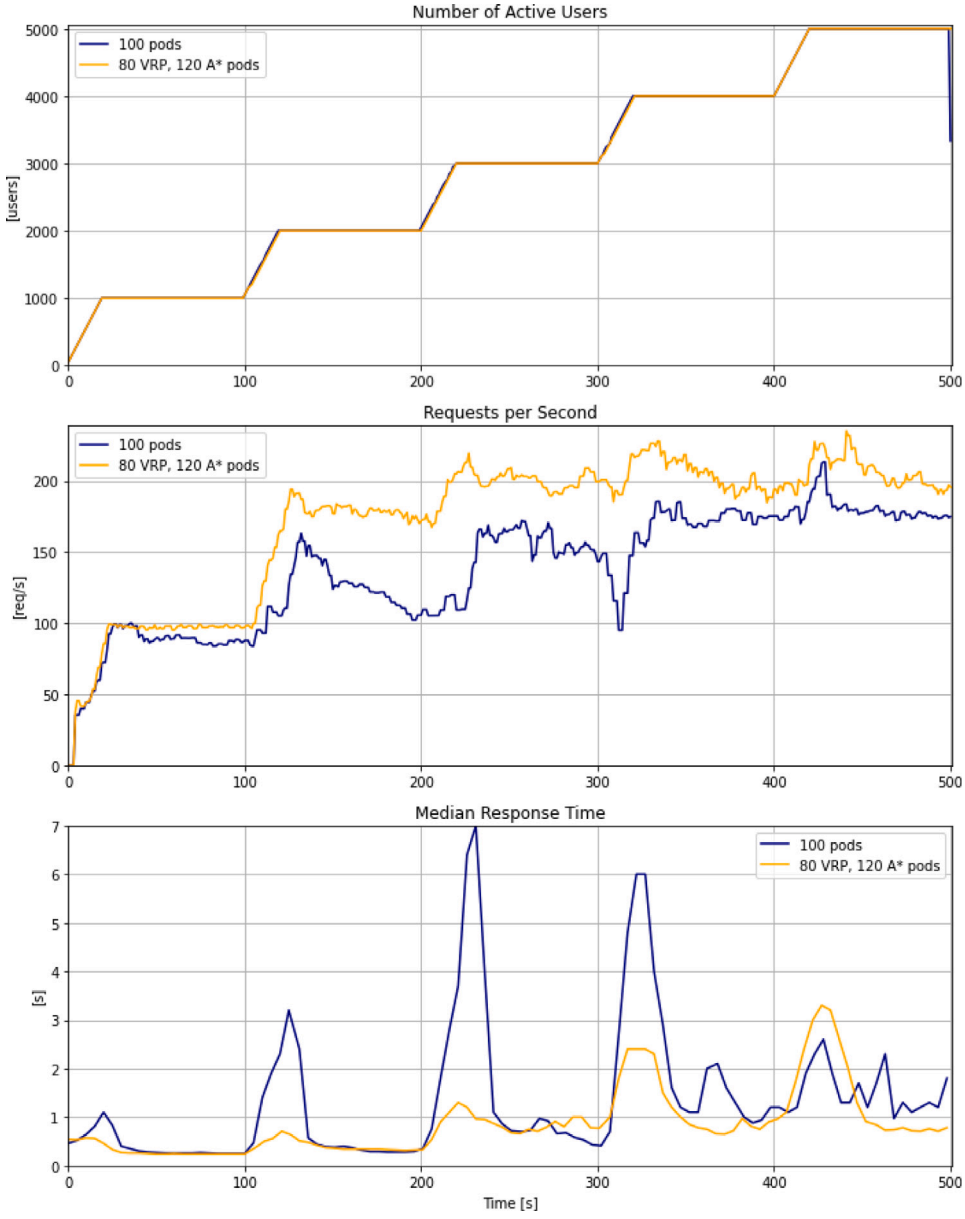


Fig. 10. Load test results for simulation of increasing demand with up to 5000 users with 100 pod replicas of Routing Solver and A* Pathfinder services deployed (navy) and with 80 Routing Solver and 120 A* Pathfinder replicas deployed (orange).

caused higher peaks in response time for 100 pods deployed as it is shown in Fig. 10. The response time stabilizes after the peak when the number of users is up to 3000. When more users are added, the response time is oscillating after the peak meaning that the system reached its capacity of requests per second. We have noticed that 5% of requests have failed and left unresponded. After analyzing the issue, we have found that A* Pathfinder pods are not able to respond to Vehicle Routing pods with equal frequency the Vehicle Routing responds to the user. While Vehicle Routing pods have not reached the request number they can support, A* Pathfinder pods have reached their limit when receiving requests from Vehicle Routing pods. To address the issue, we have deployed 80 pods of Vehicle Routing service and 120 pods of A* Pathfinder service assuring the response for all the requests. The system behavior is visualized in the same figure showing improved response time with lower peaks when the number of users is increasing. Also, the system is able to support around 200 requests per second. That is more than enough for the purpose of route calculation for autonomous infrastructure inspection planning assuming that the users are inspection operators.

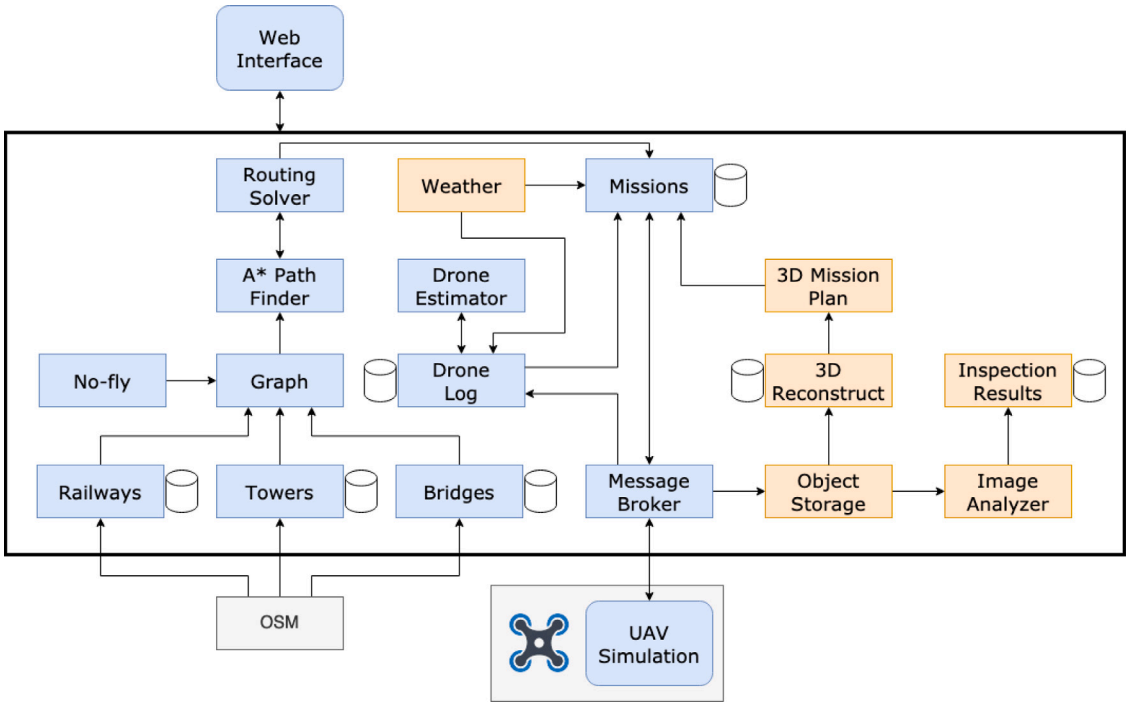


Fig. 11. Microservice application architecture showing services for future implementation in orange.

6. Conclusion and future work

This paper deals with the design and implementation of a microservices application for autonomous UAV infrastructure inspection. Using containers, services are isolated and deployed in a Kubernetes cluster where they run in pods. Pods are scaled for better application performance, and services communicate through HTTP requests. To enable faster feature addition and reduce the time from development to deployment, we set up a continuous delivery pipeline through GitLab assuring the application flexibility. We tested the application qualitatively and quantitatively and presented performance results. Mission planning and communication are validated employing containerized UAV Simulation as a Service. The calculated path waypoints are sent to the UAV which performs the route accordingly and reports its status to the user. Simulation is based on a 3D UAV dynamic model in the Gazebo environment.

Microservices architecture itself did not impact the application's performance. However, it provided better codebase organization enabling faster development. Only when deployed within the Kubernetes cluster, significant performance improvements are made by scaling individual services. The application response time can be improved and stabilized by increasing the number of pods that perform heavy computations accordingly to the load. In case the cluster has no resources to run more pods and the number of requests the system supports per second is not satisfying, a more powerful cluster is required in terms of node numbers and CPU power. It is important not to use the cluster resources to their limits, as it can negatively impact the application's performance. This application contributes to the growing field of autonomous UAVs and provides mission planning, monitoring, and management for infrastructure inspection.

In the ongoing research concerning cloud robotics, it is noticeable the lack of solutions for the fast expansion and flexible development of cloud applications. Robotics systems have limited energy, processing power, and network connectivity, especially when deployed BVLOS. With the ongoing revolution in 5G technology, robots are expected to communicate and transfer large amounts of data over long distances. In collaboration with cloud systems, robots can offload a substantial amount of computation from the onboard computers and receive flight instructions calculated in the cloud.

Microservices application, when deployed to the cloud, enables offloading computationally-expensive processes from the UAV on-board computer. We have demonstrated it with collaboration between the UAV and the cloud system. UAV sends its location to the cloud and receives navigation waypoints. UAV onboard computer is not used for heavy computations, but only to pass necessary data to the cloud where the routes are computed. The same approach can be used for local path planning. Our application provides the solution for global path planning where the objective is to reach the inspection targets. However, entirely autonomous inspection requires detailed inspection path planning, taking into account the UAV sensor readings and data gathered from the environment. Instead of performing computationally-expensive SLAM on the UAV onboard computer to reconstruct the environment and plan paths, the environment reconstruction can be processed in the cloud, based on locations and images received from the UAVs. Therefore, local inspection path planning could also be processed in the cloud and navigation instructions sent to the UAVs.

To expand the application functionalities, we aim at implementing additional services in the future. These services will provide storage, analysis, and reconstruction of images and videos collected by the UAVs on the inspection field. Reconstruction of gathered visuals could lead to the creation of infrastructure 3D models used for generating previously mentioned UAV path plans for detailed inspections. Also, visuals could be used for running machine learning models for fault detection and classification. Another idea worth considering for the future is implementing a Weather service to determine the weather in the flight area and assure the weather allows a safe flight. Furthermore, it could be used for weather prediction and route optimization, e.g., to determine the routes downwind and avoid rainy areas. Fig. 11 shows the services proposed for future implementation, as well as the integration plan and data flow with services we developed. The microservice architecture and deployment strategy we presented will assure the effortless integration of additional services.

Acknowledgment

This project has received funding from European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement No 861111, Drones4Safety.

References

[1] H. Shakhatreh, A.H. Sawalmeh, A. Al-Fuqaha, Z. Dou, E. Almaita, I. Khalil, N.S. Othman, A. Khreishah, M. Guizani, Unmanned aerial vehicles (UAVs): A survey on civil applications and key research challenges, IEEE Access 7 (2019) 48572–48634, http://dx.doi.org/10.1109/ACCESS.2019.2909530.

[2] SOCIETAL CHALLENGES - Smart, green and integrated transport, 2021, URL https://cordis.europa.eu/programme/id/H2020-EU.3.4. (Accessed 20 September 2021).

[3] L. Shi, N.J.H. Marcano, R.H. Jacobsen, G. Mehrooz, E.S.M. Ebeid, P. Schneider-Kamp, Software architecture for autonomous UAVs for power line inspection, 2019.

[4] G. Vom Bögel, L. Cousin, N. Iversen, E. Ebeid, A. Henning, Drones for inspection of overhead power lines with recharge function, in: 2020 23rd Euromicro Conference on Digital System Design, DSD, IEEE, United States, 2020, pp. 497–502, http://dx.doi.org/10.1109/DSD51259.2020.00084, 2020 23rd Euromicro Symposium on Digital System Design, DSD ; Conference date: 26-08-2020 Through 28-08-2020.

[5] N. Iversen, A. Kramberger, O. Schofield, E. Ebeid, Novel power line grasping mechanism with integrated energy harvester for UAV applications, in: 2021 IEEE International Symposium on Safety, Security, and Rescue Robotics, SSRR, in: IEEE International Symposium on Safety, Security and Rescue Robotics, IEEE, United States, 2021, pp. 34–39, http://dx.doi.org/10.1109/SSRR53300.2021.9597692, IEEE International Workshop on Safety, Security, and Rescue Robotics (SSRR) ; Conference date: 26-10-2021 Through 27-10-2021.

[6] Y. Chen, H. Hu, Internet of intelligent things and robot as a service, Simul. Model. Pract. Theory 34 (2013) 159–171, http://dx.doi.org/10.1016/j.simpat.2012.03.006, URL https://www.sciencedirect.com/science/article/pii/S1569190X12000469.

[7] Y. Chen, Iot, cloud, big data and AI in interdisciplinary domains, Simul. Model. Pract. Theory 102 (2020) 102070, http://dx.doi.org/10.1016/j.simpat.2020.102070, URL https://www.sciencedirect.com/science/article/pii/S1569190X20300083, Special Issue on IoT, Cloud, Big Data and AI in Interdisciplinary Domains.

[8] P. Boccadoro, D. Striccoli, L.A. Grieco, An extensive survey on the internet of drones, Ad Hoc Netw. 122 (2021) 102600, http://dx.doi.org/10.1016/j.adhoc.2021.102600, URL https://www.sciencedirect.com/science/article/pii/S1570870521001335.

[9] Gazebo, 2021, URL http://gazebosim.org/. (Accessed 27 October 2021).

[10] A. Mairaj, A.I. Baba, A.Y. Javaid, Application specific drone simulators: Recent advances and challenges, Simul. Model. Pract. Theory 94 (2019) 100–117, http://dx.doi.org/10.1016/j.simpat.2019.01.004, URL https://www.sciencedirect.com/science/article/pii/S1569190X19300048.

[11] M. Gharibi, R. Boutaba, S.L. Waslander, Internet of drones, IEEE Access 4 (2016) 1148–1162, http://dx.doi.org/10.1109/ACCESS.2016.2537208.

[12] S. Mahmoud, N. Mohamed, Collaborative UAVs cloud, in: 2014 International Conference on Unmanned Aircraft Systems, ICUAS, 2014, pp. 365–373, http://dx.doi.org/10.1109/ICUAS.2014.6842275.

[13] A. Koubaa, Service-oriented software architecture for cloud robotics, 2019, arXiv preprint arXiv:1901.08173.

[14] A. Koubaa, A service-oriented architecture for virtualizing robots in robot-as-a-service clouds, in: International Conference on Architecture of Computing Systems, Springer, 2014, pp. 196–208.

[15] A. Koubaa, M.-F. Sriti, Y. Javed, M. Alajlan, B. Qureshi, F. Ellouze, A. Mahmoud, MyBot: Cloud-based service robot using service-oriented architecture, Robotica 107 (2017) 8–13.

[16] A. Koubâa, B. Qureshi, M.-F. Sriti, A. Allouch, Y. Javed, M. Alajlan, O. Cheikhrouhou, M. Khalgui, E. Tovar, Dronemap planner: A service-oriented cloud-based management system for the internet-of-drones, Ad Hoc Netw. 86 (2019) 46–62, http://dx.doi.org/10.1016/j.adhoc.2018.09.013, URL https://www.sciencedirect.com/science/article/pii/S1570870518306814.

[17] W. Woodall, ROS On DDS, 2022, URL http://design.ros2.org/articles/ros_on_dds.html. (Accessed 24 January 2022).

[18] K. Fazzari, ROS 2 DDS-security integration, 2022, URL https://design.ros2.org/articles/ros2_dds_security.html. (Accessed 24 January 2022).

[19] C. Xia, Y. Zhang, L. Wang, S. Coleman, Y. Liu, Microservice-based cloud robotics system for intelligent space, Robot. Auton. Syst. 110 (2018) 139–150, http://dx.doi.org/10.1016/j.robot.2018.10.001, URL https://www.sciencedirect.com/science/article/pii/S092188901830040X.

[20] B. Xu, J. Bian, A cloud robotic application platform design based on the microservices architecture, in: 2020 International Conference on Control, Robotics and Intelligent System, in: CCRIS 2020, Association for Computing Machinery, New York, NY, USA, 2020, pp. 13–18, http://dx.doi.org/10.1145/3437802.3437805.

[21] M. Itkin, M. Kim, Y. Park, Development of cloud-based UAV monitoring and management system, Sensors 16 (11) (2016) http://dx.doi.org/10.3390/s16111913, URL https://www.mdpi.com/1424-8220/16/11/1913.

[22] M.A. Murillo, J.E. Alvia, M. Realpe, Beyond visual and radio line of sight UAVs monitoring system through open software in a simulated environment, in: M. Botto-Tobar, S. Montes León, O. Camacho, D. Chávez, P. Torres-Carrión, M. Zambrano Vizuete (Eds.), Applied Technologies, Springer International Publishing, Cham, 2021, pp. 629–642.

[23] J.A. Besada, L. Bergesio, I. Campaña, D. Vaquero-Melchor, J. López-Araquistain, A.M. Bernardos, J.R. Casar, Drone mission definition and implementation for automated infrastructure inspection using airborne sensors, Sensors 18 (4) (2018) 1170.

[24] J.A. Besada, A.M. Bernardos, L. Bergesio, D. Vaquero, I. Campaña, J.R. Casar, Drones-as-a-service: A management architecture to provide mission planning, resource brokerage and operation support for fleets of drones, in: 2019 IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops, 2019, pp. 931–936, http://dx.doi.org/10.1109/PERCOMW.2019.8730838.

[25] D. Carramiñana, I. Campaña, L. Bergesio, A.M. Bernardos, J.A. Besada, Sensors and communication simulation for unmanned traffic management, Sensors 21 (3) (2021) http://dx.doi.org/10.3390/s21030927, URL https://www.mdpi.com/1424-8220/21/3/927.

[26] S.K. Datta, J.-L. Dugelay, C. Bonnet, IoT based UAV platform for emergency services, in: 2018 International Conference on Information and Communication Technology Convergence, ICTC, 2018, pp. 144–147, http://dx.doi.org/10.1109/ICTC.2018.8539671.
[27] OpenStreetMap, 2021, URL https://www.openstreetmap.org/#map=7/56.188/11.617. (Accessed 27 October 2021).
[28] I.S.R.G. (ISRG), Let's encrypt, 2022, URL https://letsencrypt.org/. (Accessed 27 February 2022).
[29] Xfce, 2021, URL https://www.xfce.org/. (Accessed 27 October 2021).
[30] PX4 Autopilot, 2021, URL https://docs.px4.io. (Accessed 27 October 2021).
[31] GitLab, 2021, URL https://about.gitlab.com/. (Accessed 27 October 2021).
[32] Kubernetes, 2021, URL https://kubernetes.io/. (Accessed 27 October 2021).
[33] K3s: Lightweight kubernetes, 2021, URL https://k3s.io/. (Accessed 27 October 2021).
[34] Locust, 2021, URL https://locust.io/. (Accessed 27 October 2021).