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

Test and validation report

D6.4

Due date of deliverable: 31.05.2022.

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 testing and validation of a cloud platform developed in WP6 – Mission control and navigation. The deliverable presents validation of functional and non-functional requirements presented in D6.2. Functional requirements include users' interactions with the platform as well as interactions between the drones and the platform. Non-functional requirements describe the capabilities of the platform to scale and support the load. Furthermore, the platform should be easy to modify and extend with new features which is achieved by development in a microservice architecture. To validate the requirements, we conducted performance testing comparing the monolithic application to microservices, justifying the architecture choice. Moreover, we tested and validated all platform functionalities including mission planning, execution, monitoring, image transfer, upload to Alteia AI platform, satellite image download, as well as interactions with a real UAV, and presented it within this deliverable. This document includes extensive load testing results, enabling many concurrent platform users.

The document is structured in five sections. After an introduction, we present the performance test comparing the performance of the initially developed monolithic application to the microservices. Section 4 presents possible data flows within the system and their validation. Finally, section 5 presents load testing results providing the best deployment strategy to enable multiple concurrent users.

Contents

References

[1] Fowler M. Monolith first. [Online] Available: https://martinfowler.com/bliki/MonolithFirst.html

[2] Tilkov S. Don't start with a monolith. [Online] Available: https://martinfowler.com/articles/dont-start-monolith.html

[3] Richardson C. Pattern: Microservice Architecture. [Online] Available: https://microservices.io/patterns/microservices.html

[4] Fowler M. How to break a Monolith into Microservices. [Online] Available: https://martinfowler.com/articles/break-monolith-into-microservices.html

[5] Matlekovic L, Schneider-Kamp P. From Monolith to Microservices: Software Architecture for Autonomous UAV Infrastructure Inspection. arXiv preprint arXiv:2204.02342. 2022 Apr 5.

[6] Copernicus Open Access Hub. [Online] Available: https://scihub.copernicus.eu/

[7] Alteia. [Online] Available: https://alteia.readthedocs.io/en/latest/index.html

[8] Matlekovic, Lea, Filip Juric, and Peter Schneider-Kamp. "Microservices for autonomous UAV inspection with UAV simulation as a service." Simulation Modelling Practice and Theory (2022)

Acronyms

Acronym	Description
D4S	Drones 4 Safety
CPU	Central Processing Unit
API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
AI	Artificial Intelligence
REST	Representational State Transfer
CI	Continuous Integration
CD	Continuous Deployment
UAV	Unmanned Aerial Vehicle
OSM	Open Street Map
SDK	Software Development Kit

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, the software services, and the hardware drone system will offer railway and bridge operators the chance to inspect their transportation infrastructure accurately, frequently, and autonomously.

The main purpose of this document is to demonstrate the developed cloud platform as well as test its functionalities and capabilities based on functional and non-functional requirements. The development of the cloud platform for mission management began with planning and specification of the platform. It involved the determination of the requirements the platform must satisfy so that it can be used for autonomous inspections mission management. From the system specification and requirements analysis conducted in WP2, the cloud platform requirements have been determined and the initial platform design is described in D6.1 Specification of the Drone Inspection as a Service platform. The document presents the platform's high-level architecture and lists services that should be available to the user as well as provides suggestions for communication solutions between drones and the platform. As the project evolved, the platform is implemented and initially presented in D6.2 Platform implementation with modules for planning, monitoring, and control. The document presents functional and non-functional requirements and software architecture derived accordingly. It describes the implementation in detail and the tools used for development. Finally, it provides access to the platform and source code as well as a manual for users and developers on how to interact with the platform. Further extensions to the platform are presented in D6.3 Implementation of the platform with data flow and visualization modules where data storage services, as well as services for data analysis and visualization, are described.

In this document, the focus is on platform testing and validation of its functionalities. Multiple tests were conducted to validate the functional and non-functional requirements set in D6.2. Initially, the application had been developed in a monolithic architecture that did not comply with requirements. After migration to microservice architecture, we conducted tests to validate that the new architecture enabled faster processing time and flexibility. Individually, services can be tested by sending a request to the appropriate endpoint and validating the response. We tested services by validating functionalities as a whole, according to the requirements, but also individually where only one service provides complete functionality. Furthermore, we tested the interaction between the platform and a drone both in the simulation and in a real-world setting. Finally, we conducted load testing to determine the appropriate deployment setup and validate the user traffic that the platform is able to support.

3 Microservices

This section describes the initial development of the platform, i.e. migration from a monolithic architecture to microservices. Initial backend development, including routing solver, was performed in a monolithic style. After encountering difficulties in the early development and identifying function and non-functional platform requirements, it was clear that an alternative solution was required. Finally, as described in previous deliverables, the platform was developed in a microservice architecture. In order to compare and justify chosen microservice architecture, we describe both software architecture styles relevant to this document and present their benefits as well as challenges. Software architecture in general describes application organization and structure. Architectural decisions impact application quality, performance, maintainability, and usability. Software architecture chosen at the beginning of the application 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 application features grows fast. If the architecture is not reconsidered, the development slows down and application becomes difficult to maintain. That has been the main reason why, early into development, we decided to migrate the application to microservices. To test and compare the performance between the monolithic application and microservices, we designed test scenarios which results are presented in this section. The benefits of choosing microservice architecture for the mission management application have been clearly demonstrated. Not only was the development proven faster, but the route calculation performed better in the majority of scenarios.

3.1 Monolithic Architecture

Monolithic applications encompass several tightly coupled functions and tend to have a big codebase. The development does not require advanced architecture planning since the application is developed, packaged, and deployed as a standalone instance. The packaged application can be deployed to the server and scaled horizontally by running multiple instances behind a load balancer. The load balancer distributes the traffic across the instances deployed on the different servers. Testing the monolith applications can be performed end-to-end by launching the application and using an existing testing framework e.g., Selenium. However, when the code base grows and many developers work on the same application, monolith applications face challenges. The application can become too complex and difficult to understand. That prevents quick updates and development slows down. On each update, the entire application must redeploy and it can be difficult to track update impacts. It leads to extensive and time-consuming manual testing. A bug in any module can potentially bring the whole system down since all instances rely on the same code base. Even though the application can be scaled, the load is usually not equally distributed to all the modules i.e., the traffic is directed to only one application module, but they are all scaled equally. When application modules have different resource requirements, as they often do, it can lead to unnecessary CPU (Central Processing Unit) and memory consumption. Monoliths are not robust to changes and adopting a new framework or technology would lead to rewriting of the entire code base which is both expensive and time-consuming. Development in monolithic architecture could be a good choice if the application will not need to be further extended or as a starting point when the main goal is to have a simple, working end-product. However, when the development slows down as the codebase grows, it is recommended to reconsider the architecture.

3.2 Microservice Architecture

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 opinions suggesting starting an application development in monolith architecture first and then migrating to microservices [1]. As monolithic systems become too large to deal with, many enterprises are drawn to breaking them down into microservices. On the other side, others recommend starting with microservices if that architectural style is the goal [2]. However, most large-scale websites including Netflix, Amazon, and eBay have evolved from a monolithic architecture to microservices [3].

3.3 Migration Process and Platform Architecture

Migration from monolith to microservices is not an easy task [4]. There are many different approaches to splitting the monolith. The code should be studied before decoupling into the logical components. Dependencies should be identified and isolated. Even though our monolithic application was not deployed in the production and it did not have users depending on it, we encountered difficulties while refactoring, and parts of the code had to be rewritten instead of reused.

The main processes leading to application migration from monolith to microservices are as follows:

- Identification - identification of features in the monolithic application
- Isolation - isolation of features where each isolated feature is one meaningful and standalone unit
- Implementation - implementation of isolated features within the framework enabling the communication between them

For application deployment, we later developed a strategy and used DevOps tools to implement and automatize processes leading to deployment. By enabling continuous integration and continuous deployment, we tremendously benefited from the microservice architecture, i.e. better codebase organization, faster development, easier integration and deployment.

Final application architecture, identification of additional requirements, implementation, and deployment are described in detail in D6.2 and D6.3. Services are designed to enable scaling of specific processes e.g., route calculation. Figure 1 depicts the final architecture and communication flow. The drone operator interacts with the system using the web interface. The web interface communicates with the backend by sending requests and serving responses to the user. It is implemented and deployed as a microservice too. 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. Images from the drones are stored in the Object Storage and transfer is handled through the Message Broker. Satellite Imagery service accesses the satellite data from the Copernicus Open Access Hub [6] and provides the user with the download option. The Image Uploader uploads images to the Alteia AI platform [7] where the images are analyzed using deep learning techniques for fault detection and definition. The individual services are described in detail in D6.2 and D6.3 as well as databases storing mission data, drone telemetry, and images captured during the mission.

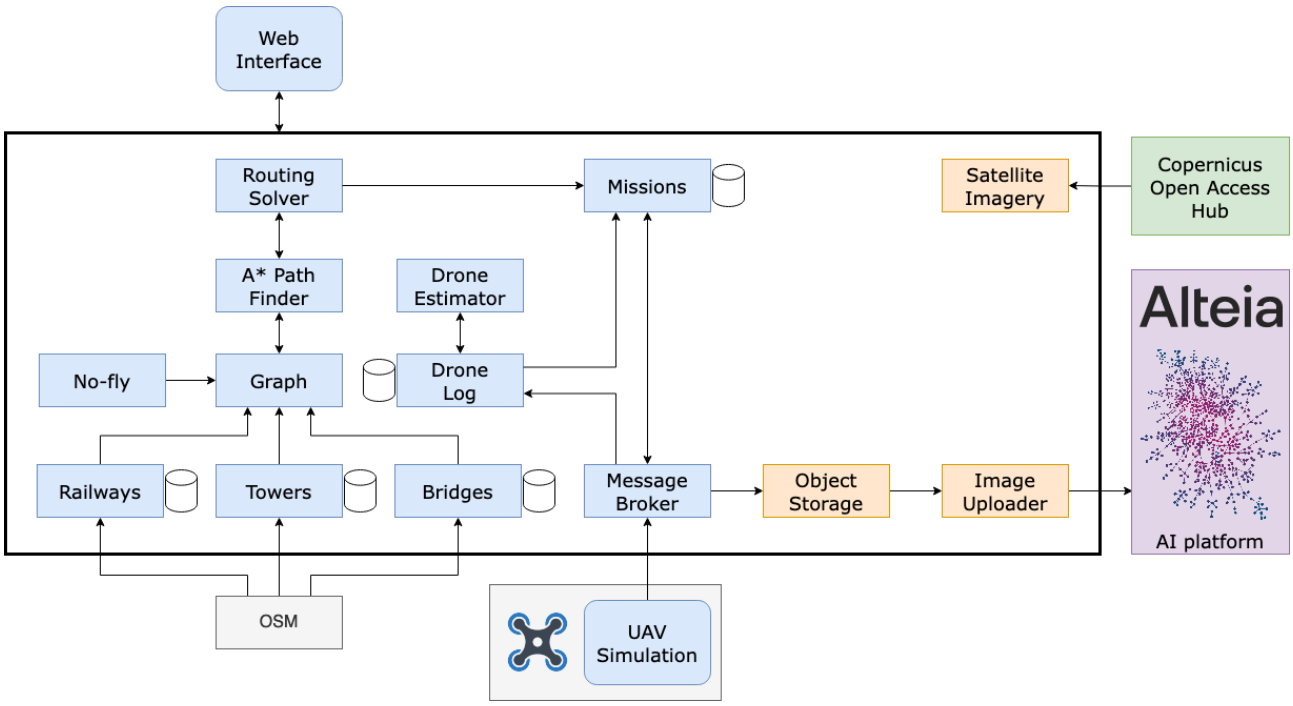

Figure 1 Microservices software architecture

3.4 Performance Evaluation

To evaluate our application performance, we deployed it automatically through the GitLab CI/CD to the Minikube Kubernetes cluster. We compared the performance with the monolithic version of the application. To assure the same running environment, we created a Docker image for our monolith application and deployed it to the same cluster within a new namespace. We tested the performance by sending requests with different sets of sources and targets and measuring the time to process the requests. Sources represent UAV locations and targets are power tower locations. We chose the locations contained in the graph randomly. The data contains power tower locations in Denmark and we suppose that UAVs are located on the power tower locations at their start. We exponentially increment the number of sources from 1 to 16, and the number of targets from 1 to 64. One hundred requests were generated for each combination of sources and targets, thus 3500 requests in total. The same requests were made to both systems. Requests to microservices application were made for a different number of pod deployments. First, we deployed only one pod per service and

measured the processing time. Then we scaled the deployment with ten A* pathfinder pod replicas to take the advantage of asynchronous communication between the vehicle routing solver and A* pathfinder services. The processing time comparison between monolithic and microservices application is shown in graphs in Figures 2 to 13. Processing time for microservices application without scaling is shown in green, with ten A* pathfinder pod replicas is shown in yellow, and the monolithic application processing time is shown in blue. Figures 2 to 6 show processing time for a different number of sources, i.e. UAVs, while altering the number of targets. Figures 7 to 13 show processing time for a different number of targets while altering the number of sources. Both options were visualized to provide a better analysis of the system's performance.

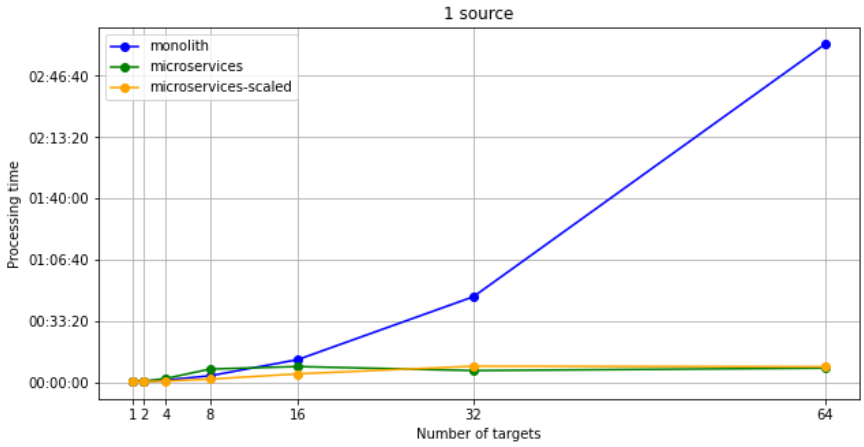

Figure 2 Processing time comparison between monolith and microservice application with 1 source and altering number of targets

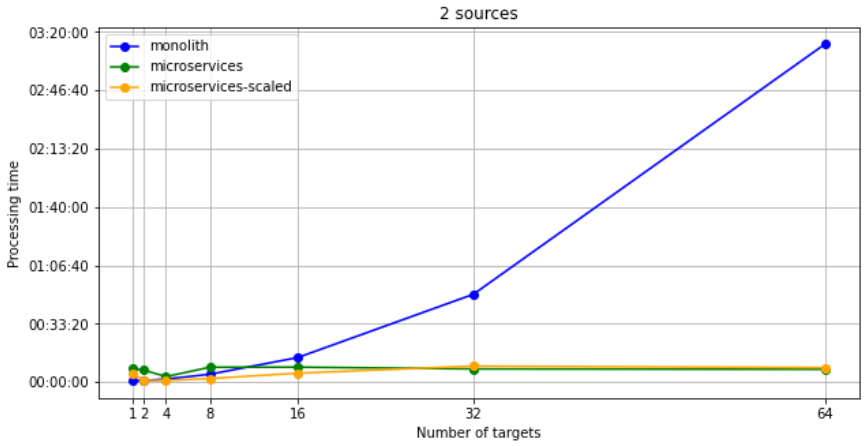

Figure 3 Processing time comparison between monolith and microservice application with 2 sources and altering number of targets

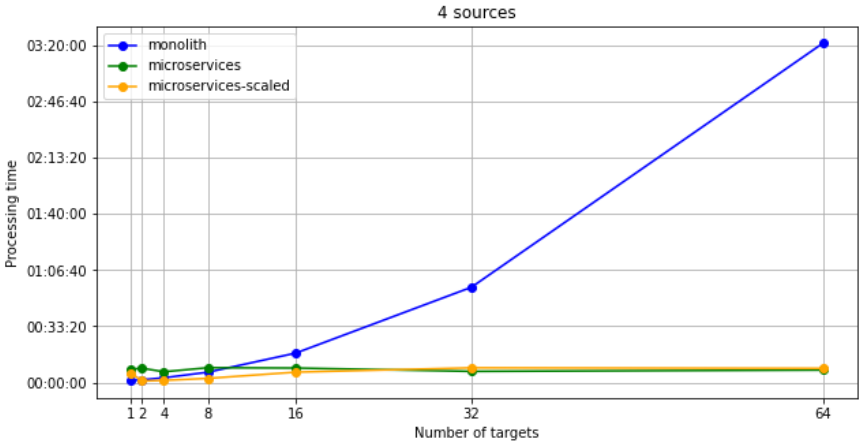

Figure 4 Processing time comparison between monolith and microservice application with 4 sources and altering number of targets

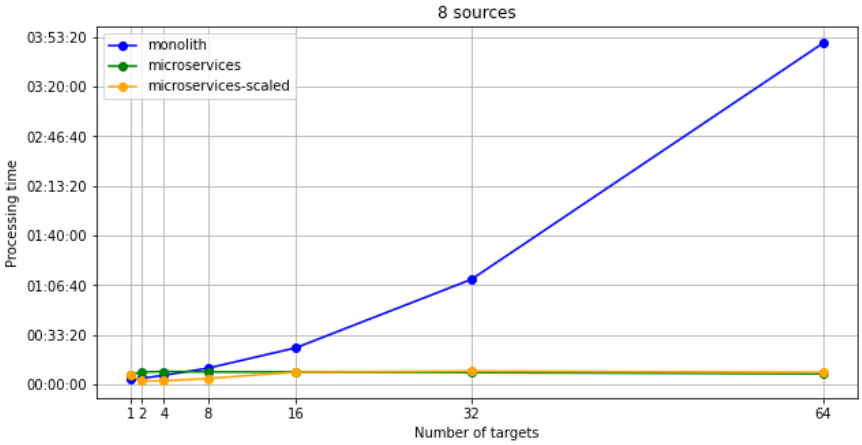

Figure 5 Processing time comparison between monolith and microservice application with 8 sources and altering number of targets

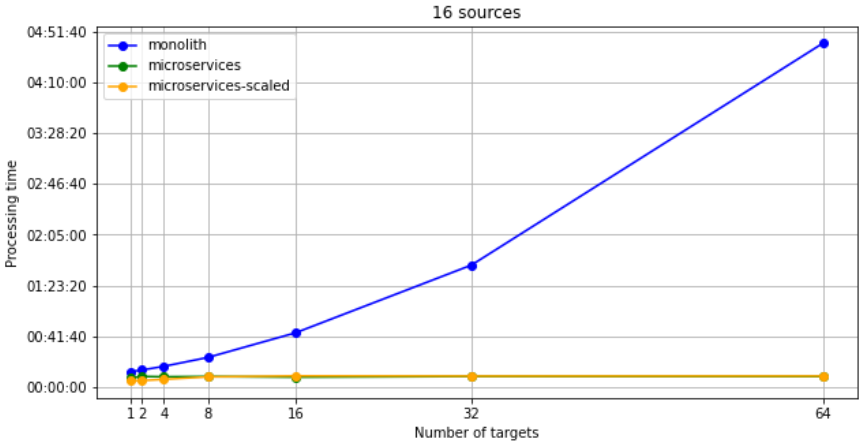

Figure 6 Processing time comparison between monolith and microservice application with 16 sources and altering number of targets

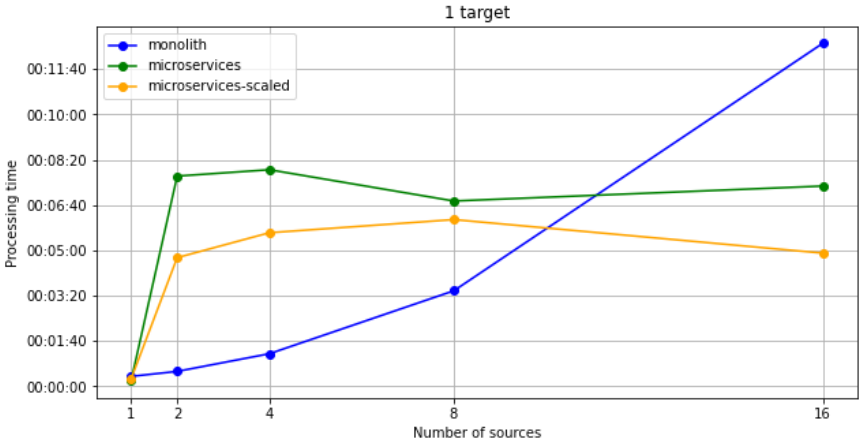

Figure 7 Processing time comparison between monolith and microservice application with 1 target and altering number of sources

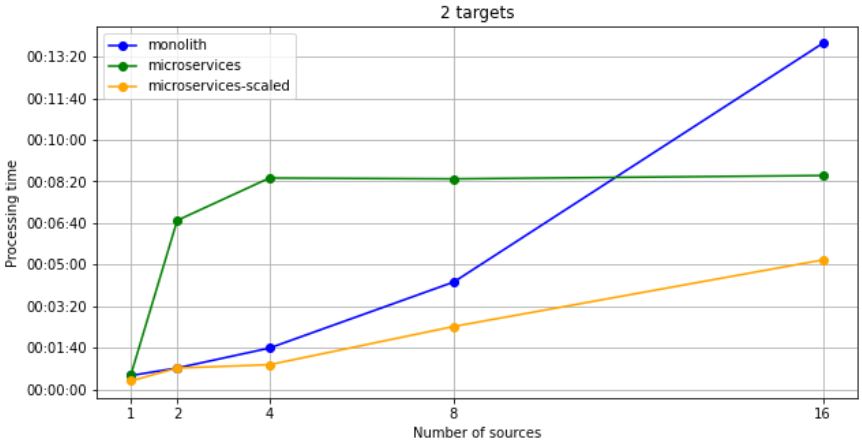

Figure 8 Processing time comparison between monolith and microservice application with 2 targets and altering number of sources

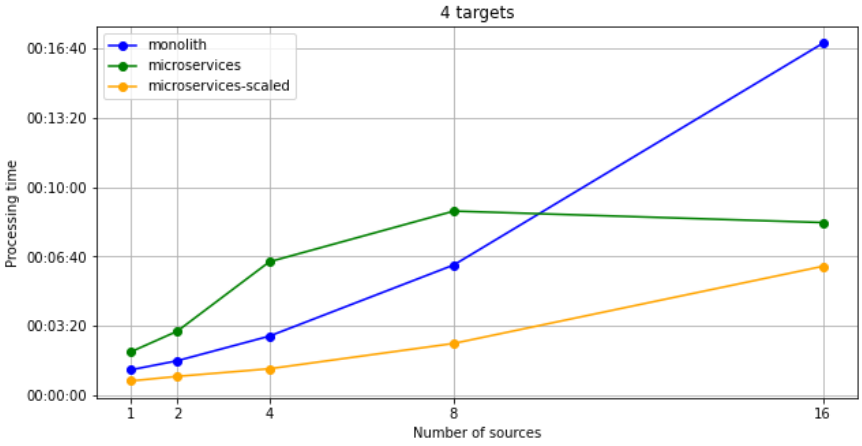

Figure 9 Processing time comparison between monolith and microservice application with 4 targets and altering number of sources

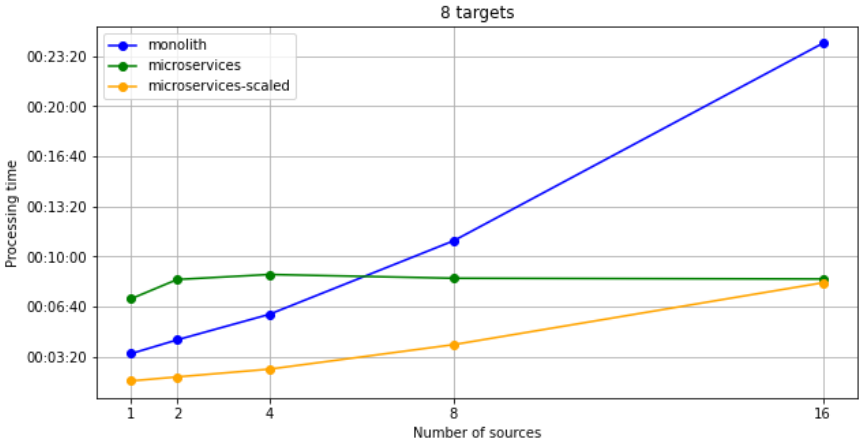

Figure 10 Processing time comparison between monolith and microservice application with 8 targets and altering number of sources

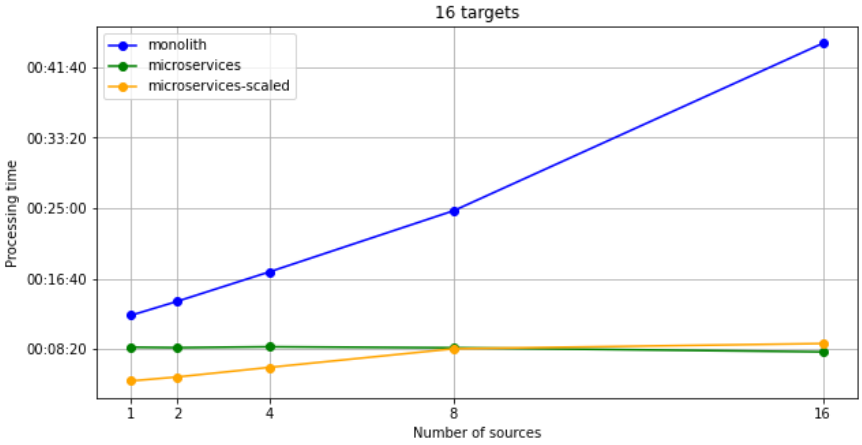

Figure 11 Processing time comparison between monolith and microservice application with 16 targets and altering number of sources

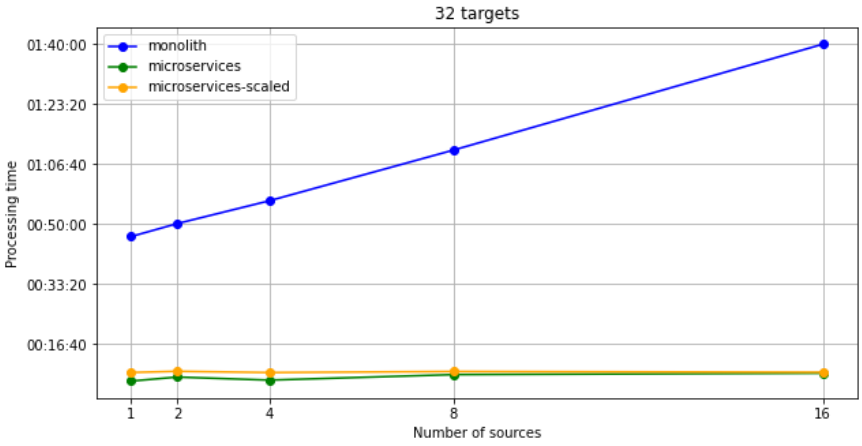

Figure 12 Processing time comparison between monolith and microservice application with 32 targets and altering number of sources

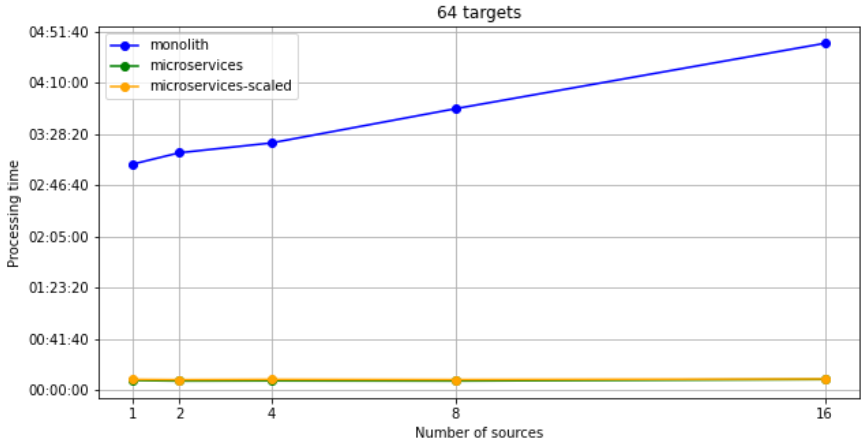

Figure 13 Processing time comparison between monolith and microservice application with 64 targets and altering number of sources

In all figures, we see that with the monolithic application, the processing time extends proportionally with the number of sources and targets while with microservices the processing time stabilizes and stops raising. For the microservices application, the overhead of internal communication between the services adds in processing time for two to eight sources visiting one target, which is best visible in Figure 7, where both scaled and not scaled microservice application performs slower than the monolith. The same is visible in Figures 3 to 5, where the path is calculated for sources to reach only one target. For a high number of sources, as in Figure 6, microservices perform faster for each number of targets. As the number of targets is raising, monolithic application processing time slows down, and microservices start outperforming the monolith, especially when scaled, as seen in Figures 8, 9, and 10. For 16 targets, microservices application performs significantly better as shown in 11. By increasing the number of targets even more the processing time becomes near constant as seen in Figures 12 and 13. If we take the highest number of sources and targets and calculate the average time it takes to find near-optimal paths for 16 UAVs to visit 64 towers, we calculate that microservices, when scaled, can process the request and serve the solution in about 5,32 seconds. Compared with monolith, which takes 1 minute and 13 seconds on average. When scaled, microservices are slower than the monolith only when two, four, or eight UAVs are chosen for inspection of one target. However, it is unlikely for a user to pick either of the combinations mentioned, since most of the time it is intuitive when visualized, which UAV is the most suitable for the mission inspecting only one target. We also have to take into account that sources and targets are chosen randomly over the country of Denmark. Therefore, they do not always represent a realistic set and sometimes it was not even possible for the solver to find a solution. We expect that the average processing time for a realistic set of sources and targets is slightly better. However, this experiment provided an insight into applications' performance and showed a positive impact of migrating the route calculation algorithm to microservices. The performance of the microservices application was further improved by deploying more pod replicas to a more powerful cluster [5].

4 Functional Validation

This section describes possible interactions within the system and functionally validates them. Implemented interactions are result of the analysis of the system's functional requirements presented in D6.2. There, we described the system's architecture and deployment strategy that facilitated platform development. Here, we demonstrate and validate user interactions with the system through the web interface as well as receiving data from the drones and mission monitoring.

4.1 Data flow

This section presents possible data flows within the system which are to be validated in this section. A user interacts within the backend system through the exposed web interface. The drones report data to the cloud where the data is stored and can be visualized on the interface. High-level interactions are visualized in Figure 14. The user interacts with the drones by setting the mission's objectives on the web interface, sending instructions to the drones, and monitoring the data they send back. Additionally, the user can directly control the drones through the QGroundControl station deployed in the cloud.

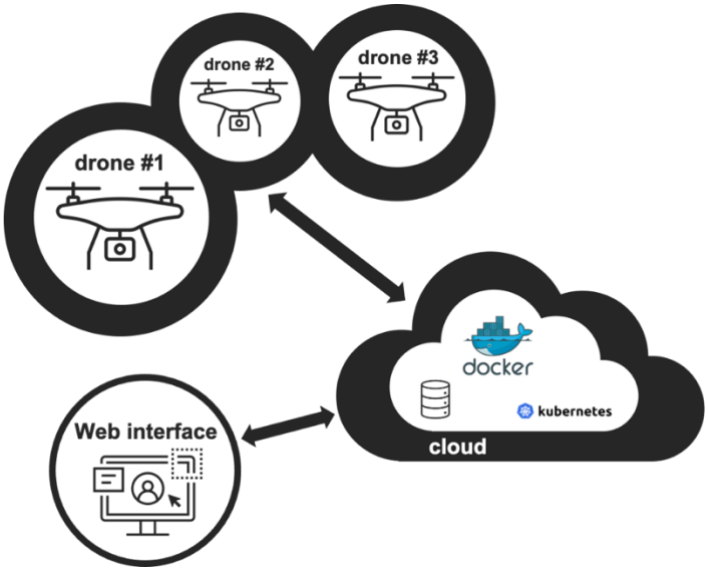

Figure 14 System's high-level architecture

4.1.1 Drone to cloud interactions

In order to monitor the mission's progress, it is essential to enable data transfer from drones to the cloud and visualize it on the user interface. Drones use sensors to measure the telemetry data and send it to the cloud. The drone's location data is stored in the Drone Log database. The data is supplemented with estimation values for the periods when the drones do not report the telemetry to the cloud. The Drone Log reports the telemetry to the web interface and saves the mission status to the Missions database. Image flow begins with taking the image using a camera mounted onboard the drone. Images are sent to the cloud and stored in the Object Storage notifying the user through the web interface that the image has been received. The data flow of drone data transfer to the cloud is depicted in Figure 15.

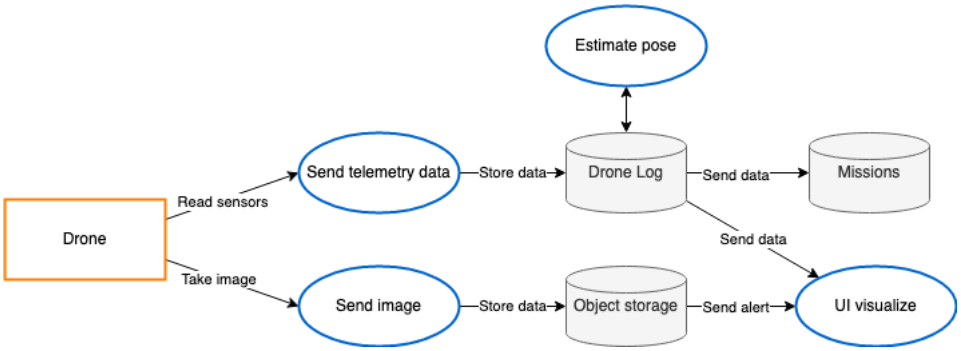

Figure 15 Data flow of transferring the data from the drone to the cloud.

4.1.2 User to cloud interactions

The user interacts with the system through the web interface where different actions are possible. To calculate mission routes, the user selects the target infrastructure on the web interface and requests route calculation. Route calculation is described in detail in D6.3. Routes are stored in the Missions database and visualized on the map-based web interface. The map can also be visualized in the satellite view and the user can have a closer look at the infrastructure from aerial images. For sending the waypoints to the drone, the user selects a mission from the possible options stored in the Missions database and starts the mission. Inspection paths are sent to the drones. Before selecting the target infrastructure, the user has the possibility to visually validate the infrastructure location by downloading an up-to-date satellite image of the infrastructure location. To download the image, the user selects a location on the map. Images are retrieved from the Satellite Imagery service which accesses the data from the Copernicus Open Access Hub. To analyze images collected by the drones, the user selects the corresponding mission and uploads images gathered during that mission to the Alteia AI platform. Images are retrieved from the Object Storage. The data flow of users' interactions with the system is depicted in Figure 16.

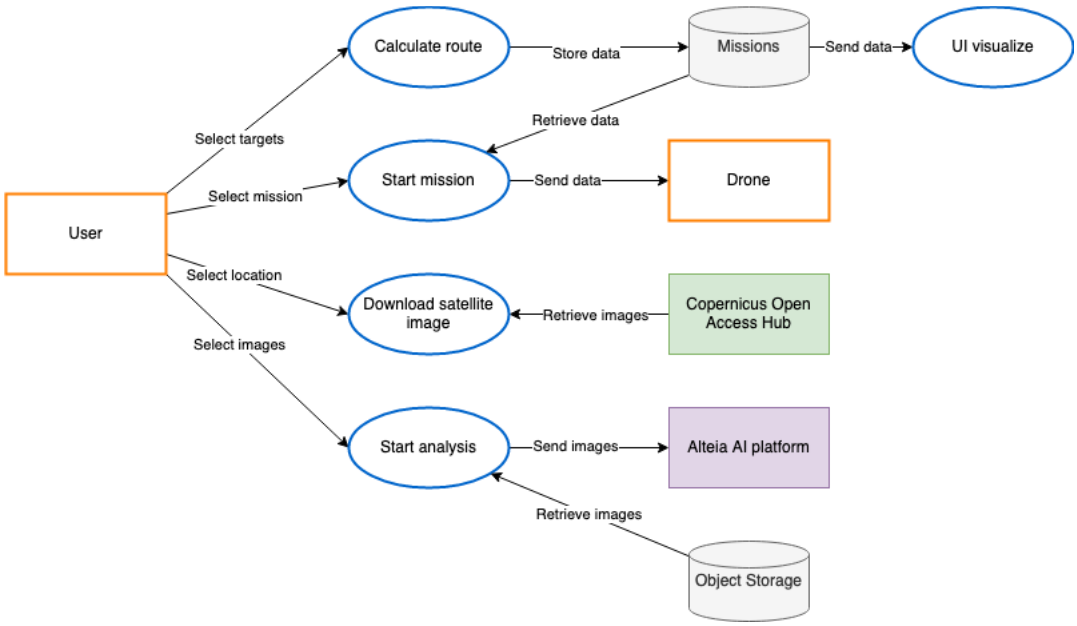

Figure 16 Data flow of user interactions with the cloud platform and drones.

4.2 Demonstrations

In this section, we demonstrate platform functionalities and interactions with the drone. Therefore, we validate mission planning, execution and monitoring, drone simulation, image transfer, image upload to the Alteia AI platform, and satellite image download. Also, we set up a real-world experiment to validate interactions between a real drone and the platform. Successful validation of mission planning, execution, monitoring, and image transfer is presented in the upcoming subsections.

4.2.1 Mission planning

Route calculation for multiple drones visiting multiple targets has been previously demonstrated in D6.2. Figure 17 shows a 2D map on the Web interface with a successful mission plan visualized in a map and satellite view. The user selects the target infrastructure for inspection. Blue circles represent power towers, purple circles are bridges, 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. 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.

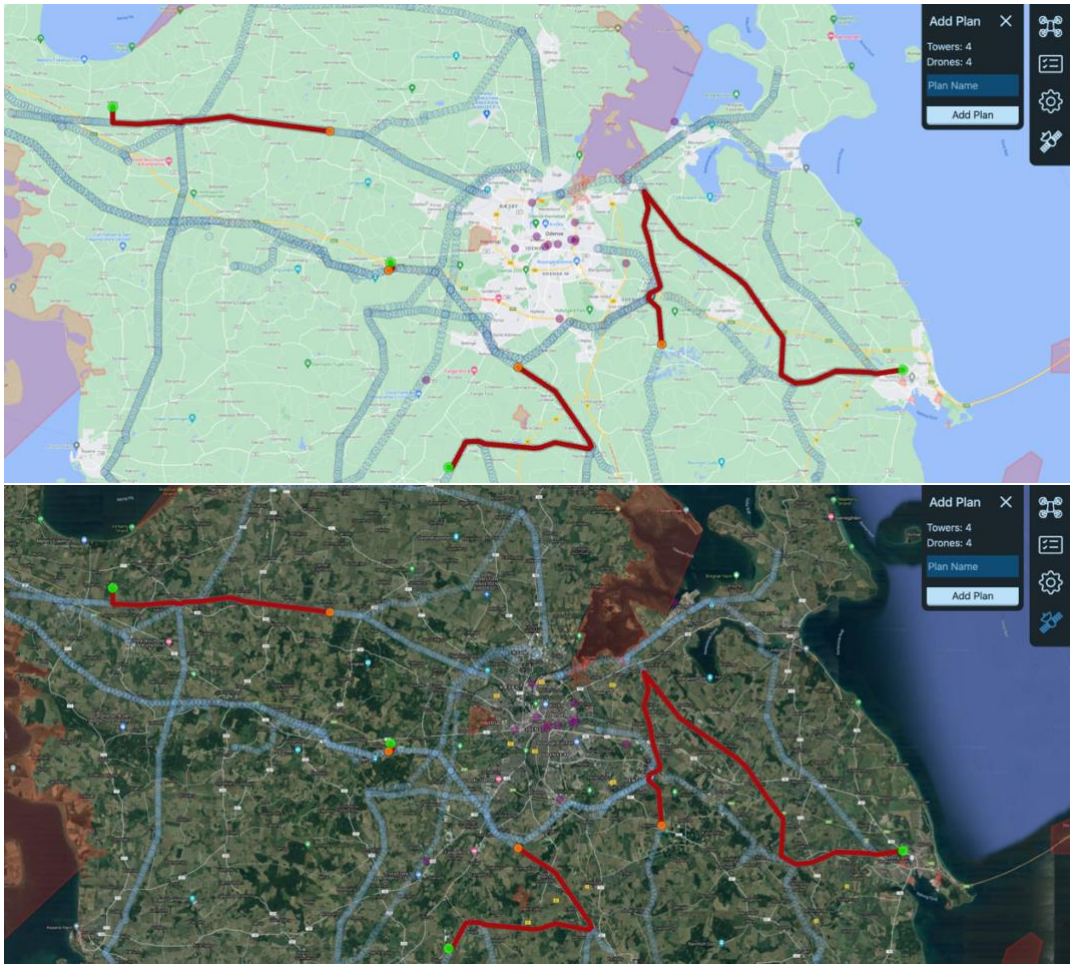

Figure 17 Map-based web interface showing calculated routes for four drones inspecting four selected bridges. Drones are shown in orange, selected bridges are green, power towers are blue, and bridges are purple. Result routes are shown in red.

We demonstrate a step-by-step mission planning process on a simple example of one drone visiting a bridge. Figure 18 shows a planning process in three steps. The drone is visualized as an orange circle on the web interface and a purple circle represents a bridge. When a user selects the bridge as an inspection target, the visualization color is changed to green representing a selected target, and a pop-up window shows with the "Calculate Route" button. When the button is pressed, route calculation services are engaged, and the calculated route is visualized in red. The user has an option to give a name to the mission and store it by pressing the "Add plan" button. As a target, we have selected a bridge around the city of Odense, Denmark. The algorithm assures that the UAV flight plan is in the proximity of the power lines for as long as possible, 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 (OSM) to facilitate the target selection and route calculation. Each bridge node is connected to the closest power tower within a radius of 3 kilometers and other bridges within 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 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.

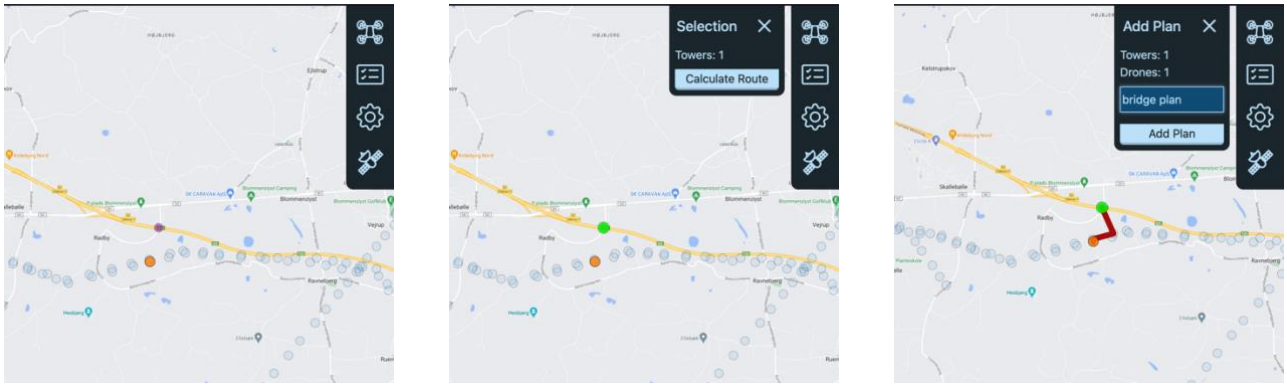

Figure 18 Step-by-step mission planning for a drone to reach a bridge.

4.2.2 Mission execution and monitoring

Stored missions can be found under the "Plan" card shown in Figure 19. Plans are divided by activity. A plan is marked as "Active" when the mission is in progress and "Finished" after the mission has been completed. "Idle Plans" shows missions that are planned but not executed. When an idle plan is chosen, a pop-up window appears with possibilities to either remove the plan or execute it. When "Start" is pressed, the plan is sent to the drone and the drone starts executing the mission.

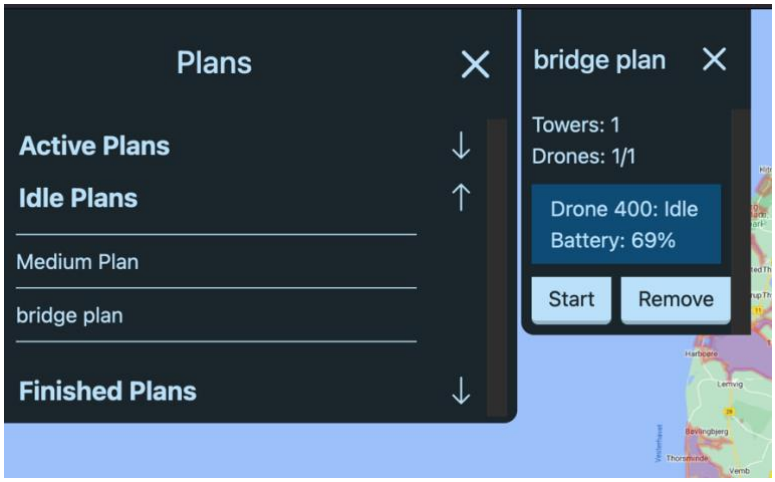

Figure 19 Mission plans grouped in Active, Idle, and Finished. To execute a mission, a user has to start a plan shown in Idle Plans.

For this demonstration, we have used the drone simulation as a service to validate the mission planning, execution, and mission monitoring. When a drone in the Gazebo simulation environment receives flight waypoints, it flies to the predefined height and an onboard computer follows waypoints point-to-point. A simulation environment showing a drone executing the mission is exposed to the web and visualized in Figure 20.

The drone reports its location to the web interface in order for a user to monitor the mission's progress. The movements from the beginning of the mission until the drone reaches the bridge can be seen in Figure 21.

We recorded the simulation flight data and used it to visualize the executed route. The waypoints sent to the UAV and executed flight path are shown in Figure 22. 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.

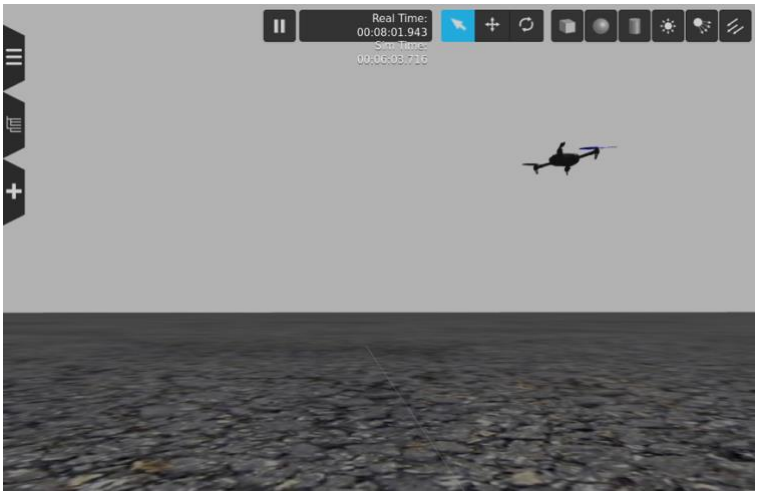

Figure 20 Simulated drone executing the inspection mission. The simulation is exposed to the web and the user can monitor the drone's movement both in simulation and on the web interface.

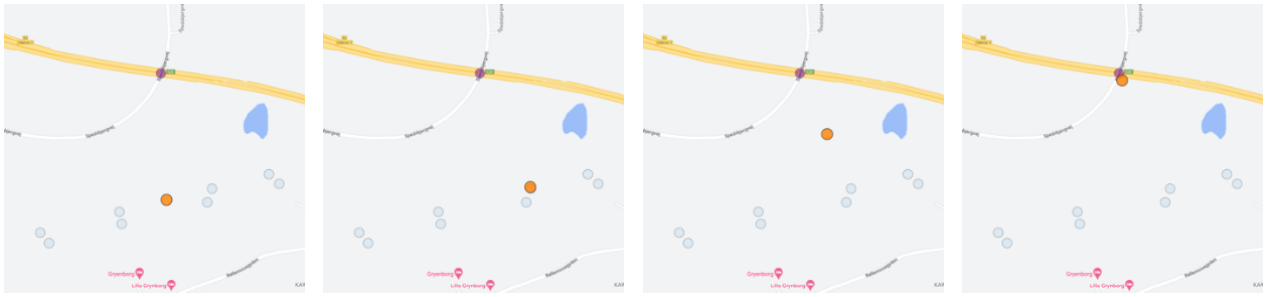

Figure 21 Drone movement monitoring during the mission on the web interface

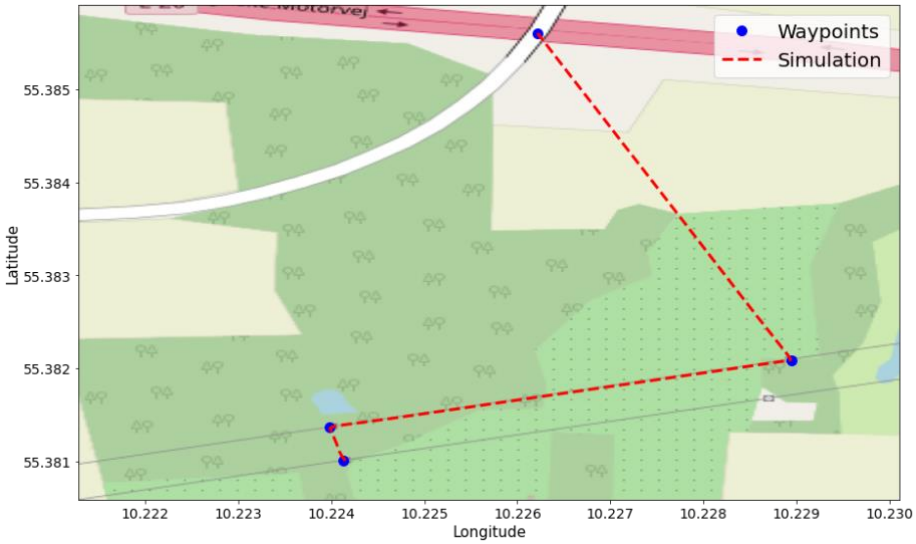

Figure 22 The route visualized from the data collected from the UAV simulated flight. Waypoints which the UAV received from the cloud are shown as blue circles. A red dashed line shows the path traversed by the simulated UAV.

4.2.3 Image transfer

During the inspection mission, a drone is required to capture infrastructure images in order to detect possible damage to the infrastructure. Captured images could be pre-analyzed onboard the drone to detect infrastructure faults that require immediate attention. The images could also be collected and sent to the cloud where they can be further analyzed in the Alteia platform under development in WP4. As a simple demonstration of receiving infrastructure image capturing an infrastructure fault, we developed an alert pop-up visible on the web interface when an image requiring attention arrives from the drone. The image is received as a POST request from the onboard computer mounted on the drone, stored in a database, and visualized on the interface. When a pop-up alert appears, a user can visualize the image by pressing the alert icon. The pop-up alert icon is shown in Figure 23 and an example image is shown in Figure 24.

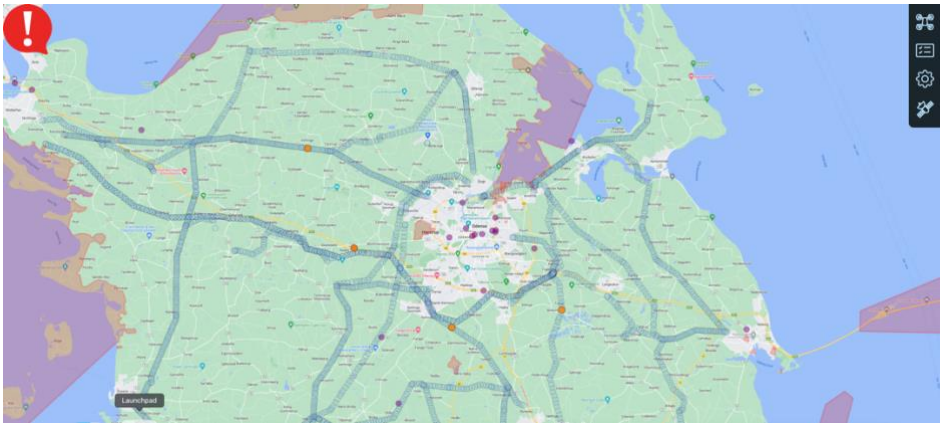

Figure 23 Pop-up alert shown on the web interface when an image is received from the drone.

Figure 24 Inspection image received from the drone and visualized on the web interface.

4.2.4 Alteia AI upload

The uploader service uploads mission images to the Alteia AI platform, currently under development in WP4. To upload mission images, Delair provided us login credentials for using Alteia Python SDK. Image Uploader service creates a new project on the Alteia platform if one with the same name does not already exist. The project can contain multiple missions, providing information about drone routes stored in the Missions service. When drones collect images, the images are stored in the Object Storage and can be provided to the Image Uploader at the users' request. The Image Uploader creates an image dataset on the Alteia platform where each image has the metadata with dimension values and the location where they were taken. The Alteia SDK provides a method for uploading images to the previously created image dataset. For each location where drones gathered images, the Image Uploader creates a dataset where an image is uploaded. The final upload is visible through the Alteia platform and it is shown in Figure 25.

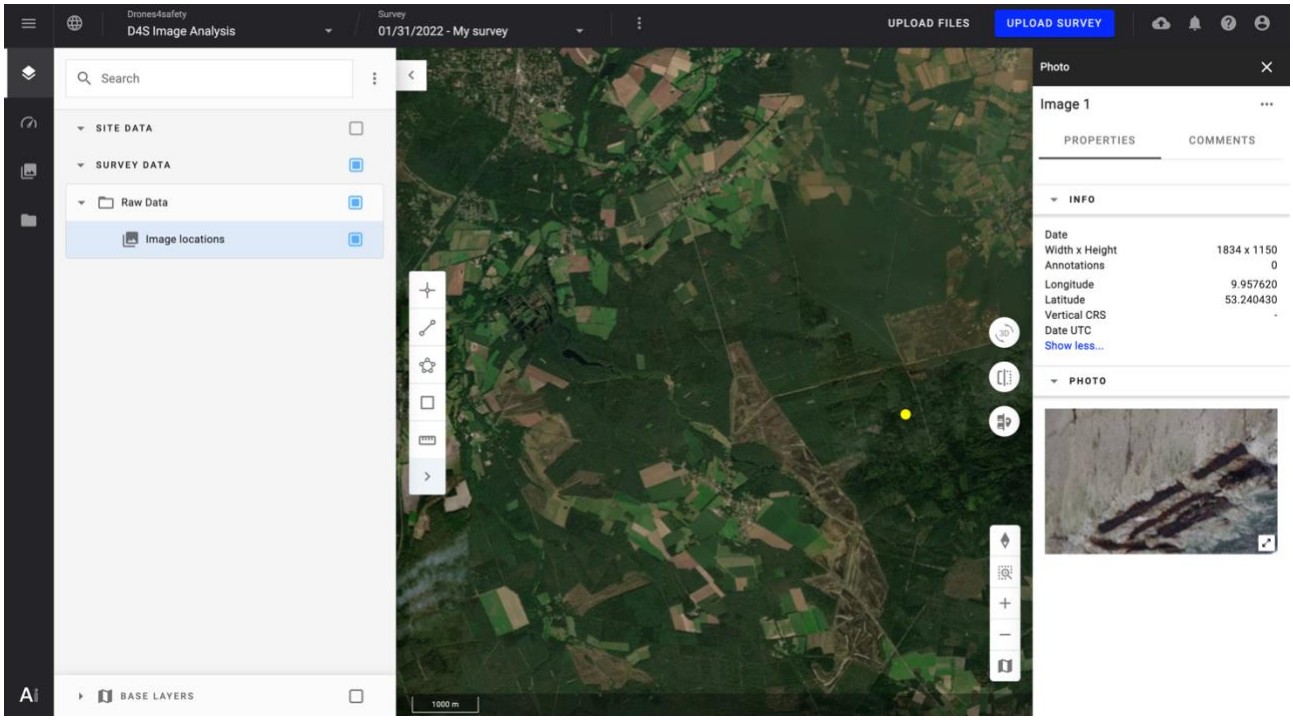

Figure 25 Infrastructure image uploaded to the Alteia platform through the Image Uploader Service. The map shows the location where the image was captured. The image is visible on the right side, including the corresponding metadata.

4.2.5 Downloading satellite images

In order to validate infrastructure locations and check if safety landing zones are in proximity, we have implemented the satellite imagery service providing the latest images. Images are retrieved from the open-source Copernicus Open Access Hub. When retrieving the images, we need to specify the bounding box which is created using the location as an input. When writing a query to request images, we specify a time when the images were taken and make sure we are retrieving recently collected data. The Satellite Imagery service retrieves data from the previous two months where the cloud coverage is only up to 20%, assuring image visibility. The API returns an ordered dictionary containing satellite images bounding boxes where our bounding box is included or partially included. The algorithm for choosing the most appropriate image is implemented, selecting and downloading only the image with the best characteristics and visibility. The image is cropped to include only the relevant location and the user can download it. The result of the described process is a satellite image showing relevant infrastructure according to the limits of the satellite image resolution. The user can validate if the infrastructure coordinates are correct and if there are flat landing zones where the drones could land in the case of an emergency. An example of a downloaded bridge satellite image is shown in Figure 26.

Figure 26 Satellite image of a bridge downloaded from the Copernicus Open Access Hub using the Satellite Imagery service.

4.2.6 Airport Test

We have conducted platform tests in a realistic setting at the Hans Christian Andersen Airport in Odense, Denmark where the infrastructure setup is located. The setup includes two power towers connected with power lines and the test is conducted using a UAV accompanied by custom software running on an onboard computer. The test setup is shown in Figure 27. We have tested the interaction between the cloud platform and the drone in form of data transfer. The test involves mission planning, execution, and monitoring as well as an image transfer.

Figure 27 Drone testing setup at Hans Christian Andersen Airport in Odense, Denmark.

Prior to the test, power towers located at the airport have been included in the graph for mission planning and visualized on the web interface. The drone reports its location to the cloud and it is also visualized on the interface. The test begins with the mission planning where one of the towers has been selected as an inspection target on the interface and the inspection route has been calculated. Inspection route stores locations of waypoints enabling the drone to reach the inspection target. The mission plan is shown in Figure 28.

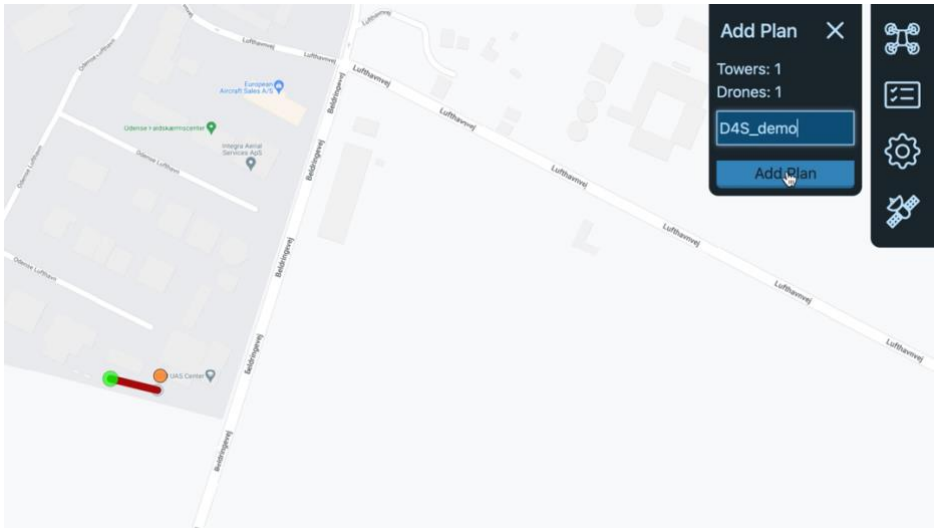

Figure 28 Mission planning for the drone to reach both power towers.

At the beginning of the mission, the drone is located next to one of the towers and reports its location to the cloud. The mission is sent to the drone from the web interface and the drone begins executing the mission. It is important to mention that the platform user is not necessarily required to be present at the site of the drone operation, as the communication is established over the internet and requires only that the drone has access to the internet. For safety reasons, we monitored the drone on site and enabled overriding of autonomous flight with manual control in case unexpected events occur. The drone waiting for the mission is shown in Figure 29.

Figure 29 The drone waiting to receive mission flight waypoints.

As the drone receives the mission, it starts flying. The drone's onboard controller is developed to lead the drone close to the mission's waypoints, but to keep a safe distance from the waypoints as they represent locations where power towers are located. To fully capture the infrastructure with the camera, drones never fly precisely from point to point as in the mission plan. The drone executing the mission is shown in Figure 30.

Figure 30 The drone executing the mission.

During the mission, the drone reports its location to the cloud and we can see the movement on the web interface. The orange circle representing the drone is moving as the mission progresses. Visualizations on the web interface are shown in Figure 31.

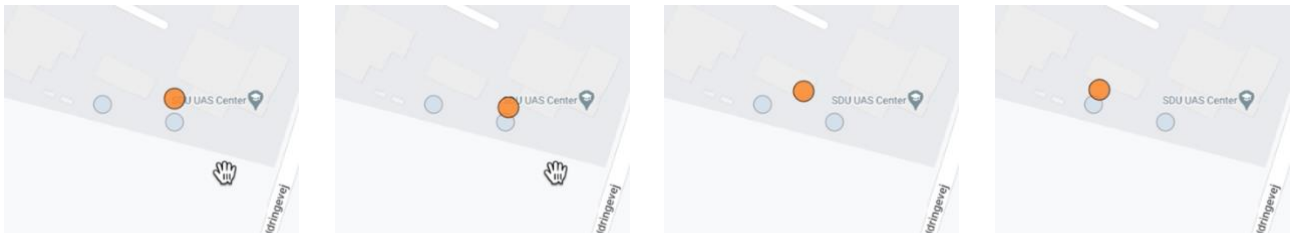

Figure 31 Mission progress monitoring on the web interface

The drone telemetry readings during the mission have been recorded to validate the movement. As instructed by the onboard controller, the drone kept a safe distance from the infrastructure resulting in a slightly shifted flight path. However, the drone used waypoints received from the cloud to gain knowledge of power towers' location and align accordingly. As a result, infrastructure images were successfully captured and transferred to the cloud. When an image is received, a pop-up alert appears on the web interface as shown in Figure 32. When pressed, the image received from the drone reveals to the user as shown in Figure 33.

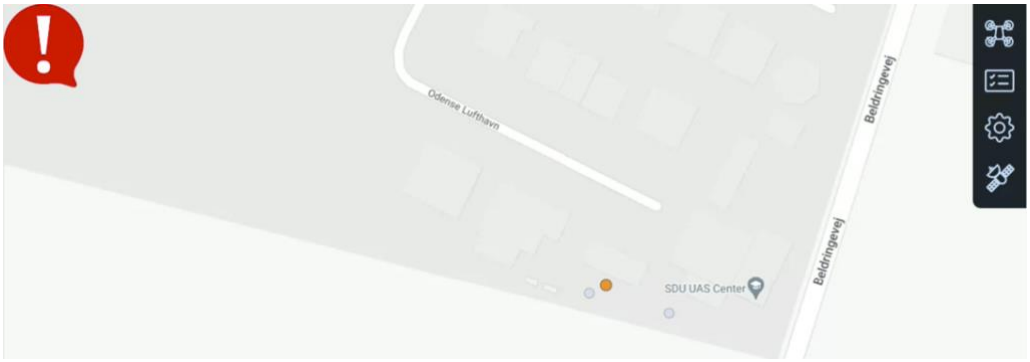

Figure 32 Pop-up alert informing the user that an image is received from the drone.

Figure 33 Image captured by the drone and received in the cloud

5 Load testing

In this section, the quantitative results which satisfy the application's non-functional requirements are presented. 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 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. Locust framework generates requests based on the desired number of users interacting with the application and increases/decreases that number at 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 behaviour. The waiting time between requests from the individual user varies from 1 to 5 seconds. The user behaviour proceeds as follows:

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

Figure 34 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 behaviour 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.

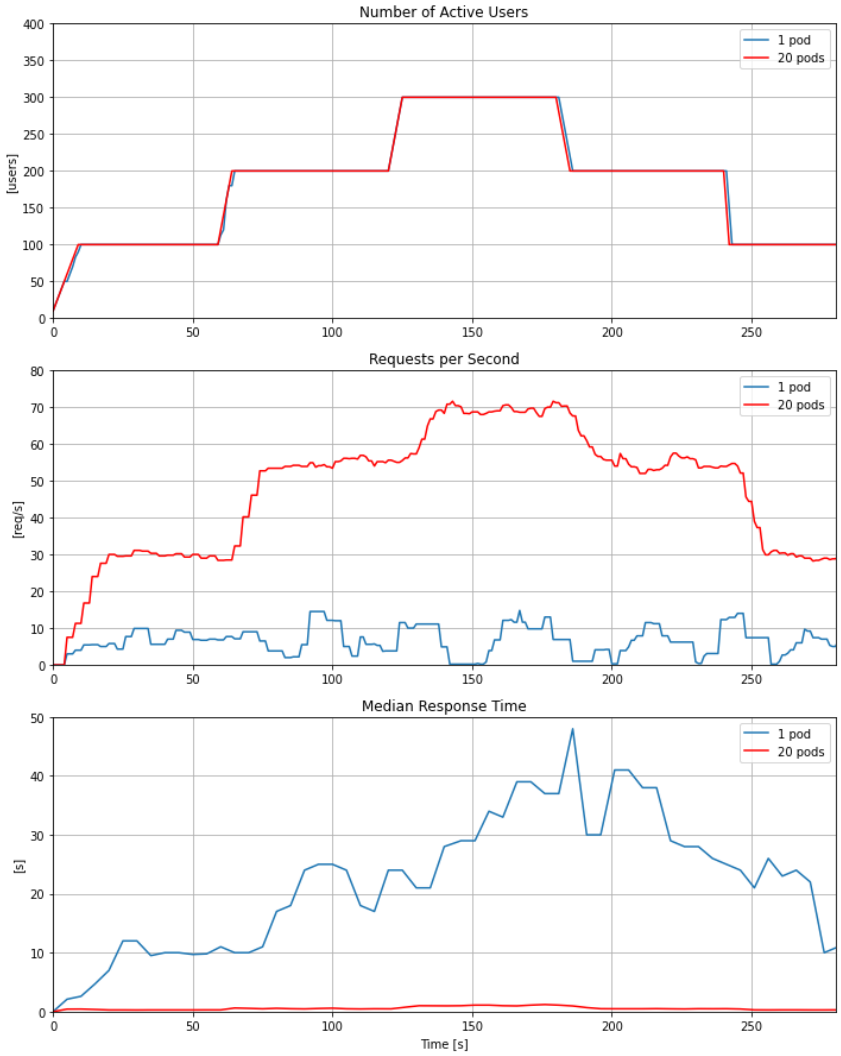

Figure 34 Load test results for simulation of increasing and decreasing demand up to 300 users with only 1 pod replica of each service deployed (blue) and with 20 pod replicas of Routing Solver and A Pathfinder services deployed (red)*

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 second. The user waits 1 second after the response to the previous request has been received to send another request. The user behaviour proceeds as follows:

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

Figure 35 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 behaviour 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.

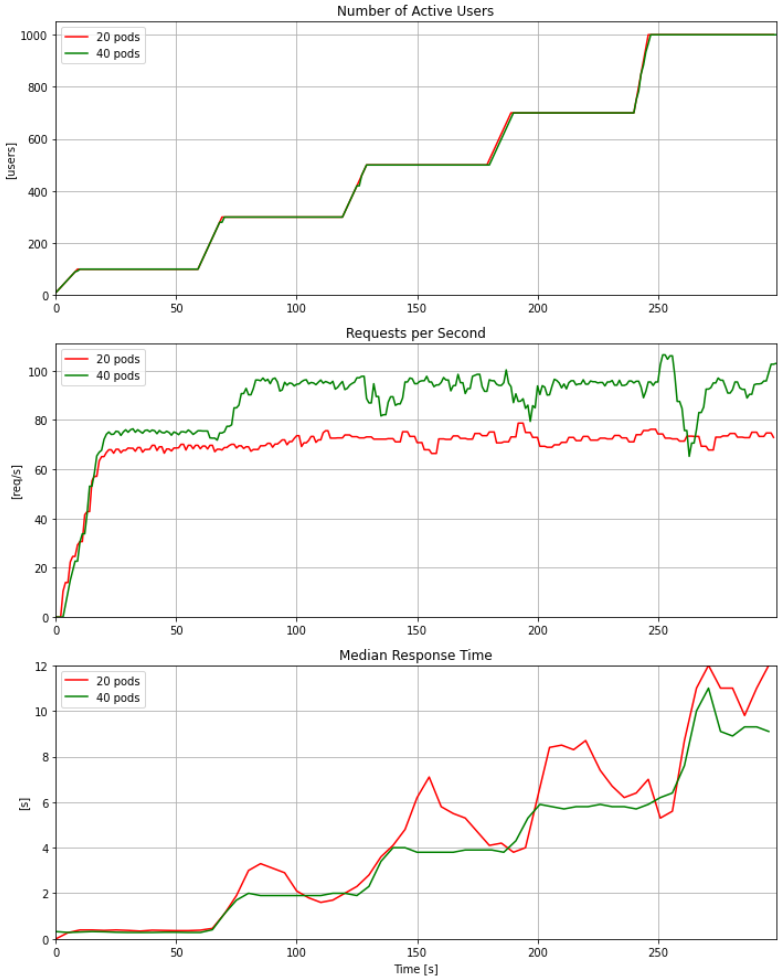

Figure 35 Load test results for simulation of increasing demand up to 1000 users with 20 pod replicas of Routing Solver and A Pathfinder services deployed (red) and with 40 replicas deployed (green)*

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 seconds after the previous 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 minute;
- 20 users per second are removed until the load reaches 500 users over a period of 1 minute;
- 20 users per second are removed until the load reaches 300 users over a period of 1 minute;
- 20 users per second are removed until the load reaches 100 users over a period of 1 minute;
- 10 users per second are removed until the load reaches 0 users over a period of 1 minute;

Figure 36 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 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.

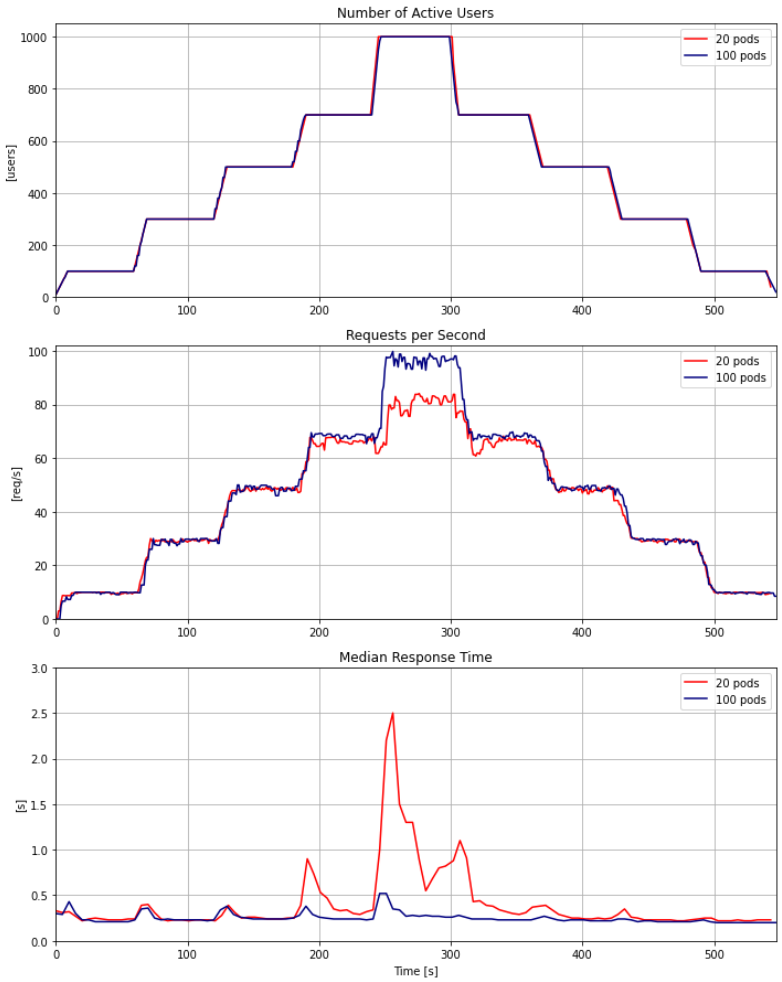

Figure 36 Load test results for simulation of increasing and decreasing demand with up to 1000 users with 20 pod replicas of Routing Solver and A Pathfinder services deployed (red) and with 100 replicas deployed (navy)*

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 seconds. The scenario proceeds as follows:

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

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 caused higher peaks in response time for 100 pods deployed as it is shown in Figure 37. 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 behaviour 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 [8].

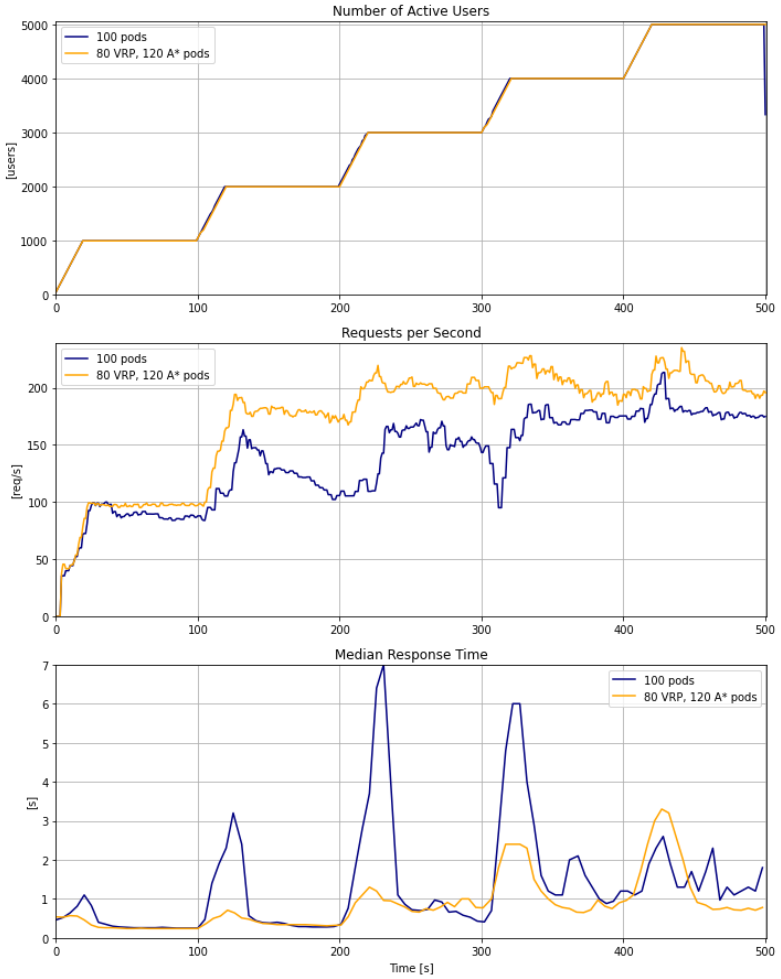

Figure 37 Load test results for simulation of increasing demand with up to 5000 users with 100 pod replicas of Routing Solver and A Pathfinder services deployed (navy) and with 80 Routing Solver and 120 A* Pathfinder replicas deployed (orange)*