# Fractus: Orchestration of Distributed Applications in the Drone-Edge-Cloud Continuum

Nasos Grigoropoulos
University of Thessaly
Email: athgrigo@uth.gr

Spyros Lalis
University of Thessaly
Email: lalis@uth.gr

2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC) | 978-1-6654-8810-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/COMPSAC54236.2022.00134

***Abstract*—Next-generation drone applications will be distributed, including tasks that need to run at the edge or in the cloud and interact with the drone in a smooth way. In this paper, we propose Fractus, an orchestration framework for the automated deployment of such applications in the drone-edge-cloud continuum. Fractus provides users with abstractions for describing the application's placement and communication requirements, allocates resources in a mission-aware fashion by considering the drone operation area, establishes and maintains connectivity between components by transparently leveraging different networking capabilities, and tackles safety and privacy issues via policy-based access to mobility and sensor resources. We present the design of Fractus and discuss an implementation based on mature software deployment technology. Further, we evaluate the resource requirements of our implementation, showing that it introduces an acceptable overhead, and illustrate its functionality via real field tests and a simulation setup.**

## I. Introduction

Drone platforms have increasingly more sensing, actuation, processing and autonomous operation capabilities; nevertheless, they are more affordable than ever. More importantly, recent regulations from aviation safety agencies, e.g., European Union's EASA [1], set the basic framework for the safe operation of drones, paving the way for their smooth integration even in urban environments. These developments promote the wider adoption of drones in several domains, from surveillance, agriculture and delivery to the monitoring of critical infrastructure and smart city applications.

However, to effectively operationalize drones in the context of next-generation applications, one needs to adopt a more structured application development and deployment approach, properly integrated with the edge and cloud computing paradigm. To achieve this, we envision drones as part of a shared infrastructure including edge and cloud resources, which can be used by different applications through a system that relieves the developer/user from the resource allocation, deployment and connectivity issues. However, drone platforms are highly mobile, rely on wireless communication, feature different sensors and are typically constrained in terms of computing resources. Thus, drones must be handled in a way that takes into account the geographic dimension of the applications, while dealing with the safety and privacy issues that arise due to their practically unlimited mobility.

In this work, we present *Fractus*[1], an orchestration framework for the automated deployment and connectivity management of distributed, component-based applications across the entire system continuum, including drones, edge nodes and the cloud. The main contributions of our work are: (i) We present the design of a complete orchestration framework that deploys and interconnects application components across the drone-edge-cloud continuum. (ii) Resource allocation and component deployment is mission-aware, based on the geographical area where the application will use the drone. (iii) Connectivity between application components is managed transparently, exploiting ad hoc local networking opportunities between drone and edge nodes that host interacting components. (iv) Critical drone functions, such as mobility and sensor operations, are accessed by the application in a controlled way, through interfaces that offer safety- and privacy-preserving operations, driven by policies that can be specified and updated by the relevant stakeholders in a flexible way. (v) We discuss a concrete implementation of Fractus, based on Kubernetes [2], the most widely used container orchestration platform, which we extend in several ways to achieve the desired functionality. (vi) We provide an extensive evaluation, showing that our implementation has an acceptable resource footprint even for the constrained platforms found on drones and edge nodes, and we illustrate the provided functionality via real field tests and a suitable simulation environment.

In the following, Section II motivates our work and gives an application example that illustrates the concept of Fractus. Section III presents the design and key functionality of Fractus. Section IV discusses the main aspects of our implementation, and Section V presents its evaluation. Section VI, gives an overview of related work, pointing out our key differentiation points. Finally, Section VII concludes the paper.

## II. Motivation and Concept

Next-generation drone applications will not be just about navigating the drone and collecting data via its onboard sensors but will include heavyweight data processing that cannot be performed solely by the drone. For instance, surveillance and monitoring applications may involve different image and big data processing tasks, while last-mile delivery applications may need to run different signal and image processing jobs in order to land and/or place cargo in a receptor with high accuracy. Even though it is possible to run some of these tasks in the cloud, in some cases it can be more beneficial to

[1]Fractus are small cloud fragments forming and dissipating rapidly.

978-1-6654-8810-5/22/$31.00 ©2022 IEEE
DOI 10.1109/COMPSAC54236.2022.00134

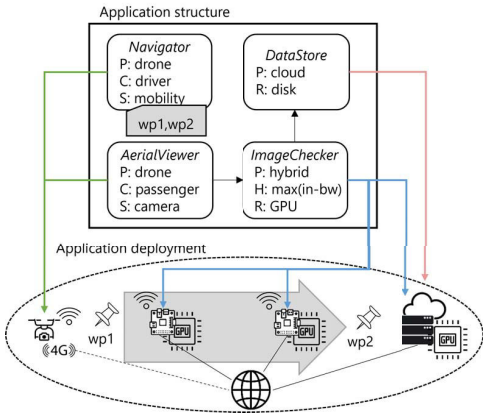


Fig. 1: Application structure and indicative deployment based on the requirements of each component

offload them to the edge, e.g., to minimize latency or reduce the amount of data transfers over metered 4/5G connections.

However, to achieve this flexibility one has to: (i) locate edge nodes in the drone's operational area capable of hosting the desired application components; (ii) schedule application components on such nodes according to the application's needs; (iii) establish direct connectivity between drone and edge nodes whenever it is deemed beneficial. Currently, all these issues are tackled manually, targeting specific setups, which makes application deployment inflexible in adapting to different environments or changing conditions. Our vision is to provide a framework where the user only provides the application's components and requirements, and all the above issues are taken care of in a transparent way. This allows developers to focus on the core application logic, while the application components can run unmodified in different setups.

Take as a concrete example a road traffic monitoring application that uses a camera-equipped drone. The upper part of Figure 1 shows an indicative component structure: the *Navigator* guides the drone so that it follows the desired path along specific waypoints to provide the required aerial coverage; the *AerialViewer* captures and filters the images of the drone camera; the *ImageChecker* is responsible for the core image processing part of the application; finally, the results are forwarded to the *DataStore* that saves them in persistent storage, from where they can be made available to other applications. The lower part of Figure 1 shows the system infrastructure. Note that the drone and edge nodes have ad hoc wireless networking interfaces. In addition, the drone is assumed to have stable 4/5G Internet connectivity, whereas all edge nodes have a wired connection to the Internet.

What Fractus proposes is that the deployment is driven by placement (P), class (C), heuristic (H), service (S) and resource (R) specifications that accompany each component (see Section III-B). The arrows directed from the application structure in Figure 1 to the nodes of the system infrastructure illustrate such an indicative deployment. More specifically, the *Navigator* and *AerialViewer* run on a drone where they access the mobility and camera service, respectively. The *ImageChecker* is instantiated multiple times on nodes that have GPU computing resources. Apart from creating a base instance in the cloud, additional instances of the *ImageChecker* may be hosted on edge nodes along the path of the drone, which can interact with the *AerialViewer* component in a direct way via WiFi instead of using a 4/5G Internet connection. During application execution, the selection of the active instance of *ImageChecker* component is based on the heuristic targeting the maximization of the bandwidth for ingress traffic, as specified by the application. Finally, the *DataStore* component is placed in a cloud node with ample storage.

To make such a model of operation attractive, we follow the cloud's paradigm and embrace multitenancy to improve resource usage and lower costs of operation. However, even though the efficient sharing of computing resources like CPU and memory can be achieved using well-known virtualization and resource-limiting practices, the multitenant usage of drones along with their sensors is more complicated as applications from different users may have different access rights to sensor data and to the drone's mobility. To this end, Fractus regulates access to such critical resources through the combination of the specified application requirements and external safety and privacy restrictions provided by the relevant stakeholders. Also, the resulting fine-grained access rights are enforced in a hard way by the system itself, without depending on the goodwill or skills of the application developer.

To continue the above example, assume a different application for monitoring the air pollution due to car traffic, which includes a *GasMeasure* drone component for measuring gas emissions and a *GasStore* cloud component for storing the measured concentration of these gases. In this case, Fractus may deploy the *GasMeasure* component on the drone that is already being used by the road traffic monitoring application, providing access to the CO/CO2 on-board sensors. In a similar vein, the component of yet another application that analyses the movement of pedestrians can be deployed on the same drone too and be given access to the on-board camera that is being used by the *AerialViewer* component of the road traffic application. However, for these components, the system might activate (possibly different) privacy-preserving policies for the images retrieved or even block image retrieval in certain areas. In addition, Fractus can restrict the *Navigator* component of the road traffic application so that it cannot direct the drone into a no-fly zone or descent below a certain altitude.

## III. Design and Functionality

Fractus is designed to support the orchestrated deployment of drone applications that seamlessly integrate drone-edge-cloud resources according to specified needs while ensuring safety and privacy. It targets applications that adopt a microservice approach, consisting of a collection of relatively small, independently deployable and loosely coupled components that communicate with each other through well-defined interfaces. The individual application components are packaged as separate containers, providing isolation and resource limiting

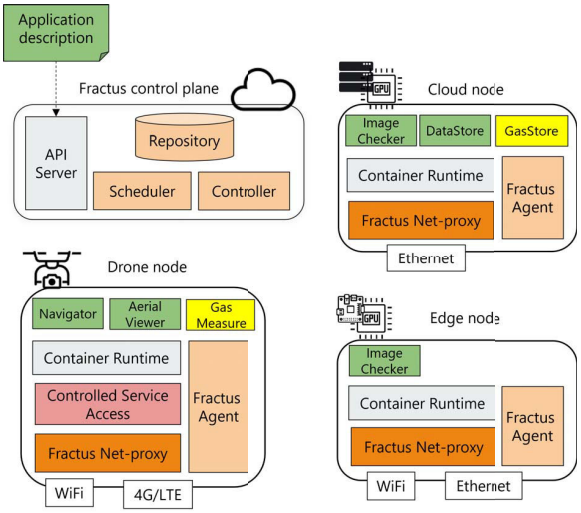


Fig. 2: Fractus architecture and indicative deployment

in an efficient and lightweight manner, which is important for the resource constrained platforms typically found on drones.

Figure 2 shows the main entities of Fractus, along with an indicative deployment of the road traffic and pollution monitoring applications discussed in Section II. In the following, we present the key aspects of Fractus in detail. Due to space limitations, we do not discuss all algorithmic aspects in detail; we note that the respective parts are implemented as plug-ins that can be replaced with more optimized versions, if desired.

### A. Resource and policy descriptions

For each drone and edge node, the Fractus Repository keeps record of both its static properties and the dynamic operation parameters. Static properties include physical features (e.g., size and weight of drones, location of edge nodes), the mobility and sensing resources offered to applications through corresponding services, service-specific capabilities (e.g., maximum speed for mobility) and the available ad hoc networking interfaces with the pairing information needed to establish communication links. Dynamic parameters include the available computing resources (e.g., CPU, memory, storage). In addition, for drones, the descriptions include status information (inactive, charging, taking-off, flying, landing), battery characteristics and current position. Further, the Repository includes information about the location of depot stations where drones are kept when idle or when they need to charge their batteries. The Agent sends the full node description to the Repository when it registers with Fractus. It also updates the values of dynamic parameters in response to requests of the Controller or periodically during application execution.

In a similar manner, the Fractus Repository stores descriptions for so-called access policies. These are submitted by relevant authorities (e.g., civil aviation, municipalities) and drone/edge node providers, to regulate the degree to which application components can access a sensing/mobility resource through the corresponding service, as well as to filter the replies returned to the application. The policies currently supported are for general access control (all services), geofencing and flight control limitations (mobility service) and privacy (camera service). A policy can be generic or be associated with specific areas, drones and applications.

### B. Application descriptions

Application descriptions specify for each component the desired placement, the required resources/services and the ingress/egress interactions with other components. Next, we focus on the parts that drive the most important functionality of Fractus for the drone and edge nodes. As an example, Listing 1 provides snippets of the component descriptions for the traffic monitoring application discussed in Section II.

**Listing 1** Partial (simplified) description for the drone and hybrid components of the traffic monitoring application

```
- component: Navigator
  placement: drone
  class: driver
  services:
  - service: mobility
    methods: [Arm, Takeoff, Goto, SetSpeed, Land]
  control: full
  controlPoints:
  - point: homeLocation
    navigation: pathBased
    path: [wp1, wp2, ...]
- component: AerialViewer
  placement: drone
  class: passenger
  services:
  - service: camera
    methods: [CaptureImage, RetrieveImage]
  egress: [ImageChecker]
- component: ImageChecker
  placement: hybrid
  heuristic:
    kind: max
    metric: ingress-bandwidth
  resources: [GPU]
  ingress: [AerialViewer]
  egress: [DataStore]
```

The component placement options are “drone” as for the *Navigator* and *AerialViewer*, “hybrid cloud-edge” as for the *ImageChecker* meaning that multiple instances can be deployed on different edge nodes and the cloud, or just “cloud” as for the *DataStore* (not shown in the listing). More specialized information needs to be provided depending on the placement. For instance, cloud components can specify the desired number of replicas, hybrid components have to specify the heuristic for selecting the currently active instance (discussed below), while drone components need to specify the mobility and sensing methods they will invoke at runtime.

Drone components declare the required degree of mobility/navigation control on the drone, by classifying themselves as a “driver” or “passenger”. A driver component may request full control for the entire flight procedure or partial control at specific locations. Respectively, it may specify a single point as the start location (which can also be left open) or several different control points. Each control point specifies the type

of navigation that will be used by the driver, which can be a path defined as a sequence of waypoints, or a region defined as a bounding polygon where the driver may do arbitrary movements decided at runtime. For instance, the *Navigator* component of the traffic monitoring application is a driver that requires full control and applies path-based navigation. In contrast, passenger components (*AerialViewer*) take advantage of a driver and can use the drone's sensor services without having to perform any explicit navigation operations. If the application does not come with its own driver, its passenger components have to specify the points of interest and optionally the order these need to be visited, and Fractus will try to find a suitable driver for it (see Section III-C). Notably, Fractus assumes that drones have obstacle avoidance capability as part of the basic autopilot stack and does not deal with such low-level issues.

Hybrid cloud-edge components (*ImageChecker*) are instantiated at least once in the cloud. Furthermore, additional instances can be created on edge nodes located in the area where the drone will operate to achieve good coverage that maximizes direct communication opportunities (e.g., over WiFi) with the interacting component on the drone (*AerialViewer*). In this case, Fractus transparently directs application traffic to or from only one so-called *target* instance at any point in time (see Section III-D). The selection is based on corresponding declarations in the application description. In the current prototype, the application may opt for maximum bandwidth or minimum latency for ingress and/or egress flows, or always prefer direct links vs communication over 4/5G Internet connections. Fractus also provides suitable hooks so that new heuristics can be easily integrated as plugins into the platform. We assume that hybrid components are stateless, which is common practice in microservice architectures.

### C. Application deployment

When an application description is submitted, Fractus performs several steps to deploy its components in the drone-edge-cloud continuum, shown in Figure 3. Next, we focus on the key deployment aspects regarding drones and edge nodes.

**Area of drone operation.** If the application has a driver, the area of operation is calculated using the navigation information (path or a wider region) for each declared control point. Else, if the application only has passenger components, the area of operation results from the respective points of interest.

**Host candidates for drone components.** If the application includes a driver component with full navigation control, the candidate drones are those found in depot stations at the specified start location. If the driver does not specify the start point or requires partial control, candidate drones are the ones in depots within a (configurable) radius from the driver's first control point. In this case, Fractus provides a *chauffeur* (system driver component) that will fly the drone to the first control point as well as between any other control points. The chauffeur by default follows the shortest path.

If the application only has passengers, Fractus considers as candidates drones already allocated to other applications, provided the points of interest lie within the driver's specified

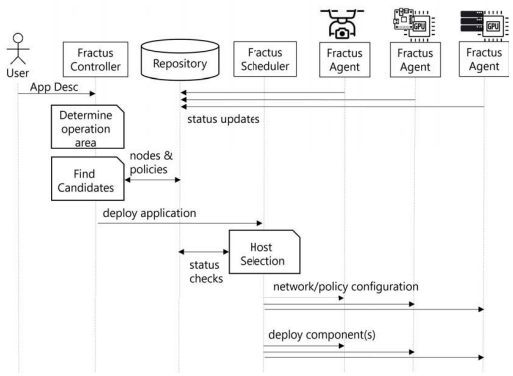


Fig. 3: Sequence of application deployment

path and are visited in the required sequence (if one is specified). In addition, unused drones close to the path's start are also considered as candidates; in this case, similar to above, Fractus provides a chauffeur that will navigate the drone through all points of interest following the shortest path.

The drone candidates are ranked according to their flight readiness level, energy level and estimated (remaining) flight time, in the spirit of [3]. For applications without an own driver, an extra scoring rule is applied to give preference to drones that are already used for other applications, instead of employing new ones as this will lead to additional take-offs/landings and a busier airspace.

**Host candidates for hybrid components.** Apart from the cloud, the candidates for hosting additional instances of a hybrid cloud-edge component are all edge nodes close to the calculated area of drone operation. In this case, the candidates are ranked in descending order of their coverage of the operation area through direct wireless communication.

**Host selection.** Based on the candidates found through the previous location-based filtering process, the Controller produces a deployment plan, which is passed to the Fractus Scheduler. In turn, the Scheduler selects the hosts for placing the different instances of the application components, first by checking the status and resource availability of the candidates in the Repository (discarding those not meeting the respective requirements), and then picking the best remaining candidates.

The best drone candidate is picked to host all the application's drone components. For each hybrid component, multiple edge hosts are selected in order to achieve good coverage of the drone's operation area through direct wireless communication. To this end, a greedy approach is employed, by iteratively picking the next candidate that maximizes the coverage until a (configurable) percentage of the operation area is covered or there are no more candidates.

**Component deployment.** As a last step, the Scheduler deploys the instances of each application component on the selected hosts. Note that the application can be functional even if only the base cloud instance is created for a hybrid cloud-edge component, although this may not achieve the specified deployment requirements in the best possible way.

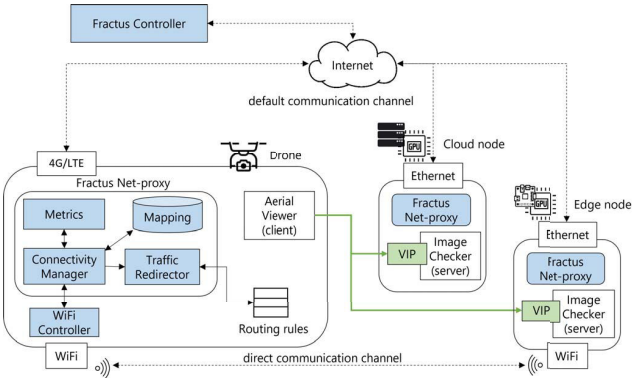


Fig. 4: Requirements-aware networking of interacting components in the traffic monitoring application

### D. Network and application flow management

Drones, edge nodes and cloud nodes have a stable Internet-based communication channel that is used for the system-level interactions with the cloud control plane. For drones, this channel is maintained over cellular 4/5G, whereas edge and cloud nodes employ a wired Ethernet connection. Application-level communication between drone components and the instances of hybrid components placed in the cloud always takes place over this channel. In addition, at runtime, Fractus exploits ad hoc wireless interfaces to form in the background local and possibly ephemeral network links between the drone and edge nodes hosting instances of hybrid components. This is done to examine alternative communication paths for interacting drone-hybrid components and select the target instance of the hybrid component that is optimal according to application requirements. The networking management is carried out, transparently to the application, by the Fractus Controller and the network proxy components that run on each node, as shown in Figure 4. Next, we discuss the main design aspects.

**Addressing.** The routing of application-level traffic is done through virtual IP addresses (VIPs). Fractus assigns a VIP to each service-providing component that is accessed by other client components according to the ingress/egress relationships in the application description. In case the same service component is instantiated multiple times (e.g., *ImageChecker*), a single VIP is assigned to all instances.

**Direct communication links.** At node startup, the Connectivity Manager on the drone and edge nodes creates a separate Interface Controller for each networking interface except the one used for the default communication channel. These Interface Controllers are responsible for performing the technology-specific network creation, discovery and connection operations needed to establish a direct IP connection at the physical layer (besides WiFi, it is possible to include other technologies, e.g., BLE). More specifically, on the edge nodes they prepare the interfaces to receive connections, whereas on the drones they join the local networks of edge nodes.

During the component deployment phase, the Scheduler sends to the Agents of the respective nodes the network configuration information generated by the Fractus Controller based on information stored in the Repository for each drone and edge node. This configuration is stored in the local Mapping directory and essentially consists of entries in the form *[name, VIP, netInfo]*, where *name* is the unique identifier of a component instance, *VIP* is the virtual IP address assigned to a server component, and *netInfo* is the required binding information for the ad hoc networking technology to be used.

During application execution, when there are multiple edge nodes hosting instances of a hybrid component, the Fractus Controller instructs the drone's net-proxy component to create a direct communication link with a specific edge node. The selection of the node is based on the proximity between drone and edge nodes, the specified range of the ad hoc networking technology and other navigation-related parameters. In path-based navigation, current drone position and speed combined with the next waypoint are used to proactively select the next node. In region-based navigation, the drone's position, speed and direction can be used to estimate its trajectory and select a new node accordingly. Upon the reception of such a command, the Connectivity Manager retrieves from the Mapping directory the relative *netInfo* and requests the respective Interface Controller to leave the current local network (if any) and initiate connection establishment to the newly selected one.

**Target instance of hybrid component.** The target instance for hybrid components interacting with drone components (e.g., *ImageChecker* and *AerialViewer*) is selected by the Fractus Controller and can change dynamically during application execution. At application startup, the target is set to the base instance located in the cloud, whereas the instances at edge nodes are in an inactive state, meaning that they neither produce nor receive any application traffic.

When a direct link is created to an edge node hosting an instance of the hybrid component, the Fractus Controller gets informed, and, depending on the corresponding application requirements, it may decide to switch immediately the target instance or start specific performance measurements. More specifically, if the heuristic is to minimize communication costs, it means that direct links are always favored over 4G/Internet communication; thus, the corresponding traffic flows at the involved nodes are redirected without any further delay. Otherwise, the Controller sends a request to the involved nodes to start running and reporting the related metrics. This process is performed by the Metrics component, which may need to measure the bandwidth of ingress and egress traffic, network latency and link quality for the available communication paths. The Fractus Controller monitors the received measurements and switches the target instance whenever deemed beneficial. Note that such a switch may also take place due to moving away from the range of the edge node hosting the target instance, in which case the base instance located in the cloud is selected until a better option arises.

Once a switching decision is made, the Connectivity Managers on the drone and the node hosting the newly selected target instance exchange routing information regarding the interacting components to make the redirection in a coordinated

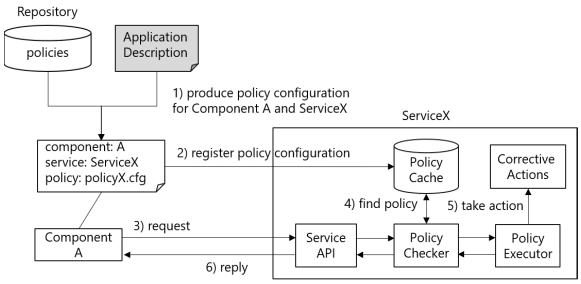


Fig. 5: Policy-based service access

way. More specifically, the Traffic Redirector on the host of the server component changes the local routing rules to enable ingress traffic through the selected communication channel, whereas its counterpart at the host of the client component enables the corresponding egress traffic through this channel. Then, the previous target instance is set in an inactive state by blocking any incoming/outgoing interactions.

### *E. Access of sensor and mobility services*

Sensing and mobility capabilities of drones are exposed in the form of local system services, accessed through well-defined APIs. Our prototype supports (i) the mobility service for controlling the drone's movement and (ii) the camera service for recording and accessing images. As mentioned in Section III-A, access to these services is regulated through policies stored in the Repository. When an application is submitted for deployment, the Controller retrieves the policies relevant to each drone component based on the declared service usage in the application description and produces a specific policy configuration for each such component. These configurations are included in the deployment plan given to the Scheduler, which forwards this information to the corresponding drone Agent during component deployment.

On the drone, the system services provided to the application have a specific internal structure, used to handle the service invocations performed by the application at runtime, as illustrated in Figure 5. In a nutshell, the policies for each application component are extracted from its configuration (before the component starts running) and are registered in a cache. Service requests are intercepted by the Policy Checker which queries the cache for policies matching the current location context, the service method and the component that performs the invocation. Depending on the outcome, the Policy Checker may handle the request as usual, reject it, or apply a corrective action. In the latter case, the request is forwarded to the Policy Executor, which provides implementations for specific corrective actions, and the produced reply is returned to the application. Besides the service call-specific content, replies also carry status information describing the corrective action taken (if any) so that the application is aware of this in order to proceed accordingly.

## IV. Implementation

The Fractus prototype is based on k3s [4], a lightweight Kubernetes distribution for edge environments, which we extend in various ways. The descriptions of drones, edge nodes, applications and policies are introduced through Custom Resource Definitions. The actual instances are stored in the persistence store of the cluster as custom objects whose lifecycle management takes place through the API Server.

The Fractus Controller is a custom controller following the operator pattern: it registers to the API Server as an event listener for new Fractus objects, on such events transforms the application description to corresponding Kubernetes objects, submits these objects for deployment and monitors their execution. The location-based scoping of candidate nodes is performed by annotating them with application-specific labels. On the other hand, each component instance is transformed to a pod, the smallest deployable unit in Kubernetes, and its main deployment requirements are specified through *nodeSelector* and *nodeAffinity* rules that use these labels. The Fractus Scheduler extends the filtering and scoring phases of the upstream Kubernetes scheduler and is responsible for the binding of the application pods to the selected hosts. We use the default predicates for filtering, and the location-based exclusion of candidate nodes is achieved via the label-matching rules.

The Fractus Agent is a custom daemon, running in parallel with the node's kubelet. At startup, it communicates with the API Server to register a new node object and add Fractus-specific labels corresponding to its features. Then, during application execution, it informs Kubernetes about resource availability and the values of its dynamic properties by updating the respective node object in the persistence store. The camera and mobility services are implemented as gRPC services using SSL/TLS authentication. Thus, for each component that accesses them, the Fractus Scheduler has to add the respective credentials to its pod.

For the default communication channel, we employ the Flannel Container Networking Interface plugin [5] which configures a layer 3 IPv4 overlay network between all cluster nodes. The VIP address of service-providing components is mapped to Kubernetes Service resources. For each such Service, the Kubernetes network proxy (running at each node) uses the Linux kernel *netfilter* framework and installs iptables rules to capture traffic to the service's VIP/port and redirect it through the overlay network. During the selection of the target instance of a hybrid component, the Fractus Controller uses *Network Policies* that control this traffic flow to isolate the other instances through suitable ingress/egress rules.

The Fractus Network proxy essentially expands the capabilities of the Kubernetes network proxy and in our prototype focuses on exploiting WiFi. The WiFi Controller supports discovery using both ad hoc and infrastructure modes. After the successful pairing at the physical layer, if the Fractus Controller selects an edge node as target, the Traffic Redirector (on both hosts) creates a mapping between the VIP, the address in the Kubernetes overlay network of each local server

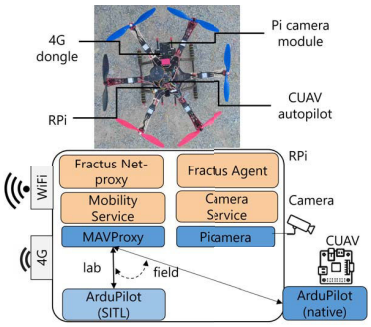


Fig. 6: Drone used in field and lab experiments

application component and a new IP address that is created for it in the direct IP network. Then, corresponding iptables rules are inserted in the *prerouting* and *postrouting* chains of the NAT table, capturing packets sent to the direct network IP addresses and changing their destination to the cluster-wide component addresses. Subsequently, this information is exchanged between the two hosts so that the other side also inserts corresponding iptables rules that capture egress traffic to each such virtual IP and redirect it to the direct network.

## V. Evaluation

We have evaluated various aspects of Fractus in the field, in the lab and through simulations. The drone used in the field and lab experiments, shown in Figure 6, is a custom-made hexacopter with a CUAV V5 nano autopilot board [6] running the ArduPilot autopilot software [7]. The drone has as a companion board a Raspberry Pi 3 Model B [8] with a quad-core ARM Cortex A53 processor (@1.2 GHz, 1GB RAM) running the Raspberry Pi OS Buster. The RPi is connected to the autopilot board over serial and runs the Fractus software. The mobility service employs DroneKit [9], which communicates with the autopilot through MAVProxy [10]. The drone is also equipped with an 8MP Raspberry Pi Camera [11], which is accessed by the camera service using the picamera library [12]. The drone communicates with the Fractus control plane and edge/cloud nodes over Internet via a 4G/LTE USB modem, whereas RPi's WiFi interface is used for direct communication. Next, we describe the system configurations and present indicative evaluation experiments and results.

### A. Resource usage and performance overheads

**Setup.** Our lab measurements focus on capturing key performance overheads of Fractus on the RPi without having to fly the drone. Thus, the Kubernetes cluster consists of the drone, which uses the software-in-the-loop (SITL) ArduPilot configuration [13] (see Figure 6), an edge node and a cloud node, all these interconnected inside a VPN over Internet. The cloud node is placed on the server running the Fractus control plane. As an edge node we employ another RPi that uses its Ethernet interface for the default communication and can also interact with the drone directly over WiFi.

**Resources usage.** In terms of disk space, Fractus, including the Kubernetes components, amounts to slightly more than 1GB, which is less than 4% of the RPi's microSD card. Regarding the memory and CPU usage we use different setups. At first, we consider only the system-level agents ($Base$) and then we add the mobility and camera services when they are idle ($Services_{idle}$) and when they serve periodic requests from two drone components. For the last setup, we measure the requirements at the system level ($Services_{active}$) and including the application ($Services_{active} + App$).

In terms of memory, $Base$ takes 110MB (due to container lifecycle management operations), $Services_{idle}$ accumulates for a little less than 200MB, which is increased by 3MB in $Services_{active}$. Finally, $Services_{active} + App$ results in a total of 250MB, leaving more than 600MB free for other applications. In terms of CPU, $Base$ takes 2.5% of the total CPU capacity, which increases to 9% for $Services_{idle}$ (due to background service threads that poll the autopilot) and to 9.7% for $Services_{active}$ when the services handle application requests. The application components combined utilize less than 1%, leading to a total CPU usage slightly more than 10%, which leaves plenty of room for more demanding components.

**Service access overhead.** To grant an application request access to the target service, the Policy Checker retrieves the current drone position and checks if it lies inside any of the regions of the related policies. The former action takes around 3.7ms, whereas the latter incurs less overhead that depends to the number of policies, e.g., for 10 policies, total granting access overhead is less than 4ms. Further, before invoking the method, one or more policy types may need to be checked to determine whether corrective actions should be applied, which introduces method-specific overhead. For instance, in the mobility service's Goto method, target location and altitude are checked against the geofence and control limit policies. If one of each policy types exists, these checks take about 16ms, resulting in a total method invocation delay of 25ms.

**Network switching overhead.** If it is decided to switch the target instance to the one at the edge, the time needed to start redirecting application traffic over WiFi can be expressed as $T_{redir} = T_{info} + T_{apply}$, where $T_{info}$ is the time to exchange routing information, which takes about 7.5 ms, and $T_{apply}$ is the time required to set the routing table rules at both sides. $T_{apply}$ is equal to $2 \times N \times T_{rule}$, where $N$ is the number of server components (for each one, a rule is set at both sides) and $T_{rule}$ is the time to set a single rule, on average 55 ms. For $N = 1$, $T_{redir}$ is measured around 120 ms, which increases linearly with the number of service-providing components.

### B. Field experiments

**Setup.** Our field experiments focus on testing the drone-related functionality in real-world conditions. Thus, the Kubernetes cluster includes the drone and the Fractus control plane, which is hosted on a Dell Precision Tower 5810 server and is connected to the drone via a VPN. Figure 7 shows the experiment's overview, with the drone located at HOME. We use an application consisting of two drone components. The

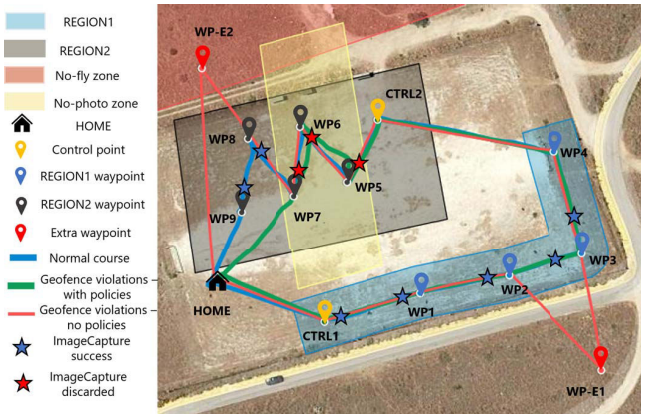


Fig. 7: Overview of the field experiment

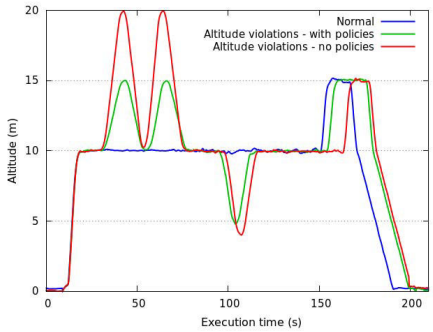


Fig. 8: Drone altitude for normal execution and when the application violates the limits with vs without policies

*Navigator* is a driver with partial control at two points CTRL1 and CTRL2. At CTRL1, it follows path-based navigation visiting in sequence WP1, WP2, WP3 and WP4. At CTRL2, it follows region-based navigation moving inside REGION2. The *AerialViewer* is a passenger component that captures images every 10 seconds, while the drone is in these areas.

**Application deployment and system chauffeur.** At first, Fractus calculates the drone operation areas. For CTRL1 the area is the polygon resulting by applying a safety radius of 7 meters to the line segment specified via the provided waypoints whereas for CTRL2 the area is the user-defined region (light blue and grey shaded areas respectively). Since there is a single drone candidate, this is selected as host for the deployment of the components. Furthermore, as the *Navigator* requests only partial control, and the initial drone location (HOME) does not match the first control point (CTRL1), Fractus introduces a chauffeur to arm the drone, take off and move to CTRL1, where the application driver (*Navigator*) takes over. When the drone reaches WP4 and returns control, the chauffeur drives the drone to CTRL2. Inside this region, the *Navigator* makes a zig-zag movement (CTRL2, WP5, WP6, WP7, WP8, WP9) and returns control. Then, the chauffeur takes the drone up to 15m, drives it to HOME and lands it.

**Policies generation.** Based on the submitted application description, the *Navigator* is granted access to the Goto and GetControl/ReturnControl methods of the mobility service while the drone is inside the operation area, through corresponding access control policies. Similarly, the *AerialViewer* is granted access to the CaptureImage and RetrieveImage methods of the camera service when the drone is in the specified area. Goto commands are further restricted by geofence control policies that discard commands with coordinates outside operation area boundaries. In addition, the red shaded region in Figure 7 is considered a sensitive no-fly zone. If the *Navigator* issues a Goto command to this area, it is considered a malicious behavior, and as a result application access to all service calls is revoked and a command for emergency landing to the HOME location is issued. Also, the allowed flight altitude after takeoff is set from 5 to 15 meters above terrain, enforced via a flight limit control policy. Finally, the yellow shaded stripe inside REGION2 is considered a no-photo zone, thus a privacy control policy is added to discard calls to the camera service when the drone is in that region.

**Normal execution.** The bold blue line in Figure 7 depicts the course of the drone during a normal execution of the application, and the blue stars indicate the locations where the *Aerial Viewer* successfully captures and retrieves an image. Also, the blue line in Figure 8 plots the recorded flight altitude (relative to the takeoff location) during the normal execution where the *Navigator* keeps the drone steadily at 10 meters. The rise to 15 meters in the last phase before landing is due to the default altitude set by the system chauffeur.

**Enforcement of geofence limits.** We program the *Navigator* to deviate from the areas declared in the control points. While the drone is in REGION1, after reaching WP2, we introduce a Goto command to WP-E1, and while the drone is in REGION2, after reaching WP7, we introduce a command to WP-E2 in the no-fly zone. The red trace in Figure 7 shows the course followed without any geofence control policies, whereas the green trace shows the course followed with these policies enabled. In the latter case, the drone trace in REGION1 remains unchanged since WP-E1 is simply discarded, whereas in REGION2 the application execution is disrupted after issuing WP-E2, followed by the emergency landing to HOME location under the control of the system.

**Enforcement of flight limits.** We program the *Navigator* to set the altitude of the Goto commands for WP1 and WP3 to 20 meters, and for WP6 to 4 meters, thereby violating the control limit policy for the flight altitude. Figure 8 shows the altitude during such an execution (green) vs an execution without the altitude limit control policy in place (red). As can be seen, every time the application attempts to violate the bounds, Fractus manages to enforce the proper limit.

**Enforcement of privacy restrictions.** We program the *AerialViewer* to take photos also when the drone is above the no-photo zone inside REGION2 (red stars in Figure 7). Such requests are discarded due to the privacy control policy and the respective invocations of the camera service fail, returning an error to the application.

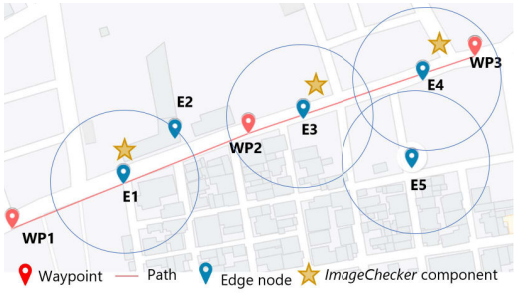


Fig. 9: Waypoints of the *Navigator* and deployment of the *ImageChecker* instances on the edge nodes

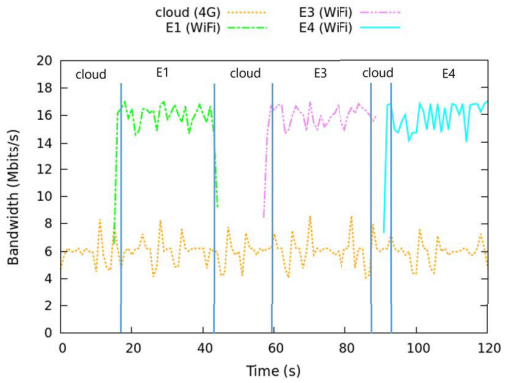


Fig. 10: Measured bandwidth between *AerialViewer* and the *ImageChecker* instances, and selected target instance

### C. Simulation experiments

**Setup.** Our simulation experiments focus on illustrating provided functionality, benefits and deployment overhead for more complex scenarios. We configure a multi-node Kubernetes cluster on a single machine, where each node, including drones, is represented by a Docker container with the corresponding Fractus software. The drone's mobility service accesses ArduPilot SITL, the physics simulation is provided by the Gazebo simulator [14] and the camera service accesses Gazebo's virtual camera stream. Networking between the Fractus control plane and the nodes is provided through an isolated bridge network. For the drone's simulated 4G interface, we set the latency to 60ms and limit the transmission rate to 8Mbps, via the Linux traffic control utility. WiFi networking between drone and edge nodes is through ns-3 [15], set to operate in the ad hoc mode of 802.11g with a data rate of 24Mbps.

**Requirements-aware deployment and networking.** We use an application with three components mimicking the traffic monitoring application. The *Navigator* uses the drone mobility service to take off at WP1, set flying speed to 3m/s, go to WP2 and then to WP3 where it lands. The *AerialViewer* retrieves images every 1 second and sends them to a dummy implementation of the hybrid cloud-edge *ImageChecker* that does not perform any actual processing. Target *ImageChecker* selection heuristic is set to maximize the ingress bandwidth. Figure 9 shows the application waypoints, expected path and the positions of the edge nodes; those with a blue circle have WiFi interfaces and the radius indicates their range. The figure also illustrates the deployment of multiple *ImageChecker* instances on the edge nodes. Note that E2 and E5 are filtered-out as the former does not have a WiFi interface and the latter does not offer WiFi coverage in the drone's path.

Figure 10 plots the measured bandwidth between *AerialViewer* and the instances of the *ImageChecker* on the cloud and the WiFi-connected edge node as reported by *iperf*. The vertical lines show the intervals where each instance is selected as target. It can be seen that, whenever possible, Fractus favors the nearest instance located at the edge since direct WiFi connectivity offers higher bandwidth. Note that since the drone can have a single direct WiFi link with one of the edge nodes at any point in time, in cases where the WiFi coverage of different edge nodes is overlapping, Fractus switches to the cloud instance as target (via 4G) in order to connect with another edge node that is closer to the drone direction (see 90th second). The intermediate target switch to the cloud instance ensures a continuous data flow at the application, during the time it takes to establish a direct wireless connection to the next edge node and to evaluate the link before deciding to switch the target instance.

**Development benefits.** We have built a version of the above application that accesses the Fractus camera and mobility services and achieves the same behavior without utilizing the network management of Fractus. To this end, we let the *AerialViewer* access directly the drone's WiFi interface to perform the discovery of nearby edge nodes, the connectivity establishment and the traffic redirection. Corresponding code is also placed at the *ImageChecker*. Even for such a simple Python application, indicative implementations of the components are 193 and 125 lines of code larger compared to their original versions (increases of 240% and 440%, respectively).

Note that, since the management of the network addresses has to be performed manually and the heuristic that leads to network switching is hard coded in the application code, the adaptation for a different system setup or different application needs, requires extra manual programming effort. Also, in order for the component code to perform these operations, the respective pods must run in privileged mode, which creates many security risks and gives almost unrestricted access to the resources on the host system. Further, the deployment of the application components would have to be performed in a hardwired way through manual inspection of each node's networking capabilities. From the above it is clear that Fractus greatly reduces the programming and deployment effort.

**Application deployment overhead.** The total application deployment time is $T_{deploy} = T_{area} + T_{loc}^{filter} + T_{res}^{filter} + T_{host} + T_{comp}$, consisting of the time needed to calculate the area of operation, filter the candidate hosts based on their location and resource availability, select the best hosts and deploy the application components, respectively. Here, we focus on the overhead of Fractus, leaving out $T_{res}^{filter}$

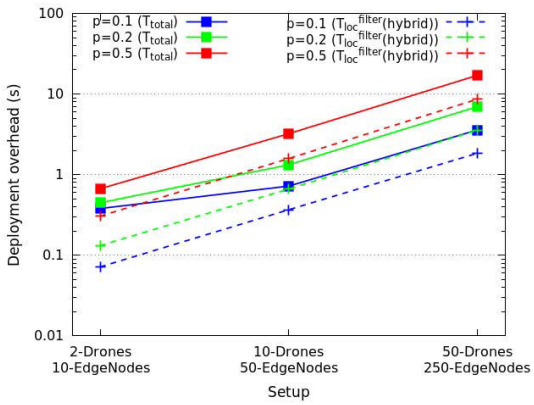


Fig. 11: Deployment overhead of Fractus (y-axis is log-scaled)

and $T_{comp}$ which are introduced by Kubernetes. We use the traffic monitoring application described in Section II, where the path of the *Navigator* includes 20 waypoints. We take measurements for different configurations where we vary the number of drone and edge nodes (at a ratio of 1:5) and set their location so that only $10\%$, $20\%$ and $50\%$ (but at least one) is a candidate for hosting each application component. Figure 11 presents the results. At larger scales, most of the overhead comes from $T_{loc}^{filter}$ and in particular the time to filter candidates for the hybrid cloud-edge component. This is due to the number of checks performed when many edge nodes are located in the area of operation. One way to reduce this overhead is to stop checking for additional candidates once enough have been found, based on a threshold similar to the *percentageOfNodesToScore* setting in Kubernetes [16].

## VI. Related Work

**Edge computing and orchestration.** ParaDrop [17] and AirBox [18] present edge platforms for deploying functionality on behalf of cloud services through a single intermediate layer, whereas CloudPath [19] supports the hierarchical deployment of such services over the geographic span of the network, based on a FaaS architecture. Both ParaDrop and AirBox allow the user to interactively deploy Docker containers on edge nodes via a cloud interface, with the latter leveraging trusted execution environments for enhanced security. Although Fractus has similarities with their application deployment model and mechanisms, in addition it provides a transparent computation-communication continuum across drones, edge nodes and the cloud, while addressing aspects that differentiate drones from mobile user devices.

There are works targeting the orchestration and management of drones, some following the cloud robotics approach that offloads all application intelligence to the cloud [20] [21], whereas others enable the native execution of container-based drone applications [22] [23]. Also, BeeCluster [24] proposes a predictive optimization strategy for minimizing the total execution time of drone-based sensing tasks. Our work extends these efforts, aiming at the end-to-end deployment of next-generation drone applications that can take full advantage of edge and cloud resources in a transparent way, rather than dealing only with the part that resides on the drone.

Thematically closer to our work are platforms creating a unified layer of resources that can be accessed transparently by the applications, like PCloud [25] which forms a personal cloud that seamlessly combines the appropriate resources according to application needs. From a design perspective, our work is more similar with FocusStack [26], which extends the well-known OpenStack platform for managing edge devices as typical datacenters, while we opt for Kubernetes, the most widely adopted container orchestration platform. FocusStack employs a geographical routing layer providing location-based scoping to reduce management communication between edge and cloud and also to enable interactions between edge devices. However, in their system prototype, the georouter server is in the cloud, whereas Fractus supports direct wireless communication between drone and edge nodes based on application requirements. Moreover, the above systems are focused to computing and passive sensing applications. Our work goes beyond that point, by including the aspect of mobility control and taking care of the related requirements.

**Network management.** Software-defined networking (SDN) and network function virtualization (NFV) through management frameworks like OSM [27], follow cloud orchestration techniques for the efficient provisioning of network services. The 5G concept adopts this service-oriented view of the network in order to realize its vision of satisfying different and possibly contrasting requirements of a variety of applications [28]. Although 5G can benefit drone-based applications, it necessitates extensive deployments by network operators that can take quite some time to achieve the coverage and stability required for drone-based applications, which may need to operate not only in urban but also in remote areas. Furthermore, a large number of static edge devices will most likely keep using various wireless networking technologies based on specific application needs [29]. Fractus can incorporate and exploit different networking technologies in a flexible and transparent way, without any effort from the application developer.

In the edge environment, our work shares similarities with systems that support technology-agnostic interactions. Haggle [30] separates networking details from the application allowing seamless connectivity of web applications across infrastructure and infrastructure-less communication environments. ubiSOAP [31] provides network-agnostic connectivity with QoS-aware network link selection and SOAP-based communication over a multi-network overlay. Similarly, Omni [32] is a recent effort towards a multi-networking middleware that enables the opportunistic use of wireless technologies by leveraging various device-to-device communication capabilities of IoT devices. Given that ubiSOAP and Omni target opportunistic IoT networking, a large part of these works is dedicated on the discovery of nearby devices offering compatible services. Differently, Fractus uses a centralized cloud layer that has concrete knowledge of the location and available communi-

cation technologies of drone and edge nodes and exploits this knowledge to produce suitable deployment configurations.

In the context of Kubernetes, it is common to connect application components in a loosely coupled way through the Services abstraction, and service meshes, like Istio [33], completely separate the application's business logic from the communication logic by creating an abstracted application-aware overlay. Although these meshes allow fine-grained centralized management, they introduce extra overhead due to the injection of sidecar proxy containers in the application pods, which is more noticeable in resource constrained environments. Recently, more lightweight, edge-oriented Kubernetes derivatives have been proposed, which allow application components to take advantage of various networking technologies. For instance, KubeEdge [34] introduces an MQTT-based communication model through custom protocol mappers and SMARTER [35] makes use of the device plugin to directly expose the low-level networking interfaces to the application components. Although these approaches are definitely useful, they break the service-based interaction pattern and shift the responsibility of connectivity management to the application developer. In contrast, Fractus can transparently take advantage of different ad hoc networking capabilities under the hood.

## VII. Conclusion

We have presented Fractus, a framework for the orchestration of distributed drone-based applications that supports mission-aware deployment and is able to transparently redirect application traffic through ephemeral direct communication links. This is achieved through structured descriptions that accompany the application code and allow users to specify placement and communication requirements, as well as corresponding, extensible, system-level mechanisms. Further, safety and privacy constraints are enforced through the policy-based access of critical resources. Using a real drone and a simulation setup, we show through indicative scenarios that Fractus achieves the required functionality, while decreasing the development effort and incurring acceptable overhead.

## Acknowledgment

This work was funded by research grants of the University of Thessaly through the Research, Innovation, and Excellence (DEKA) scholarship program. It has also been co–financed by the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH - CREATE - INNOVATE, project PV-Auto-Scout, code T1EDK-02435.

## References

[1] EASA, "Civil drones (Unmanned Aircraft)," https://www.easa.europa.eu/domains/civil-drones-rpas, 2022.
[2] Kubernetes, "Container Orchestration," https://kubernetes.io/, 2022.
[3] R. R. Price, "How to estimate the maximum and recommended flight times of a UAS UAV," LSU AgCenter, Tech. Rep. 3469, Jan 2016.
[4] K3S, "Lightweight Kubernetes," https://k3s.io/, 2022.
[5] flannel, "L3 network fabric," https://github.com/flannel-io/flannel, 2022.
[6] CUAV, "V5 nano," http://doc.cuav.net/flight-controller/v5-autopilot/en/v5-nano.html, 2022.
[7] ArduPilot, "Open source autopilot," http://ardupilot.org, 2022.
[8] Raspberry Pi 3 Model B, "Third-generation single-board computer," https://www.raspberrypi.org/products/raspberry-pi-3-model-b/, 2022.
[9] DroneKit, "Developer tools for drones," http://dronekit.io/, 2022.
[10] MAVProxy, "Ground Station," https://ardupilot.org/mavproxy/, 2022.
[11] Raspberry Pi, "Camera Module," https://www.raspberrypi.org/products/camera-module-v2/, 2022.
[12] picamera, "Pure Python interface to the Raspberry Pi camera module," https://github.com/waveform80/picamera, 2022.
[13] ArduPilot, "SITL Simulator," http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html, 2022.
[14] Gazebo, "Robotics simulator," http://gazebosim.org/, 2022.
[15] ns-3, "Network Simulator," https://www.nsnam.org/, 2022.
[16] Kubernetes, "Scheduler performance tuning," https://kubernetes.io/docs/concepts/scheduling-eviction/scheduler-perf-tuning/, 2022.
[17] P. Liu, D. Willis, and S. Banerjee, "ParaDrop: Enabling lightweight multi-tenancy at the network's extreme edge," in *Proc. IEEE/ACM Symposium on Edge Computing (SEC)*, 2016, pp. 1–13.
[18] K. Bhardwaj, M.-W. Shih, P. Agarwal, A. Gavrilovska, T. Kim, and K. Schwan, "Fast, scalable and secure onloading of edge functions using AirBox," in *Proc. IEEE/ACM Symposium on Edge Computing (SEC)*, 2016, pp. 14–27.
[19] S. H. Mortazavi, M. Salehe, C. S. Gomes, C. Phillips, and E. de Lara, "CloudPath: A multi-tier cloud computing framework," in *Proc. ACM/IEEE Symposium on Edge Computing (SEC)*, 2017, pp. 1–13.
[20] J. Yapp, R. Seker, and R. Babiceanu, "UAV as a service: Enabling on-demand access and on-the-fly re-tasking of multi-tenant UAVs using cloud services," in *Proc. IEEE/AIAA Digital Avionics Systems Conference*, 2016.
[21] A. Koubâa, B. Qureshi, M.-F. Sriti, A. Allouch, Y. Javed, M. Alajlan, O. Cheikhrouhou, M. Khalgui, and E. Tovar, "Dronemap Planner: A service-oriented cloud-based management system for the Internet-of-Drones," *Ad Hoc Networks*, vol. 86, pp. 46–62, 2019.
[22] A. Van't Hof and J. Nieh, "AnDrone: Virtual drone computing in the cloud," in *Proc. EuroSys*, 2019, pp. 6:1–6:16.
[23] N. Grigoropoulos and S. Lalis, "Flexible deployment and enforcement of flight and privacy restrictions for drone applications," in *Proc. IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2020, pp. 110–117.
[24] S. He, F. Bastani, A. Balasingam, K. Gopalakrishnan, Z. Jiang, M. Alizadeh, H. Balakrishnan, M. J. Cafarella, T. Kraska, and S. Madden, "Beecluster: drone orchestration via predictive optimization," in *Proc. International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2020, pp. 299–311.
[25] M. Jang, K. Schwan, K. Bhardwaj, A. Gavrilovska, and A. Avasthi, "Personal clouds: Sharing and integrating networked resources to enhance end user experiences," in *Proc. IEEE Conference on Computer Communications (INFOCOM)*, 2014, pp. 2220–2228.
[26] B. Amento, B. Balasubramanian, R. J. Hall, K. Joshi, G. Jung, and K. H. Purdy, "FocusStack: Orchestrating edge clouds using location-based focus of attention," in *Proc. IEEE/ACM Symposium on Edge Computing (SEC)*, 2016, pp. 179–191.
[27] ETSI, "Open Source MANO (OSM)," https://osm.etsi.org/, 2022.
[28] H. Zhang, N. Liu, X. Chu, K. Long, A. H. Aghvami, and V. C. M. Leung, "Network Slicing Based 5G and Future Mobile Networks: Mobility, Resource Management, and Challenges," *IEEE Communications Magazine*, vol. 55, no. 8, pp. 138–145, 2017.
[29] S. Schick, "Will 5G replace Wi-Fi?" https://enterprise.verizon.com/resources/articles/s/will-5g-replace-wifi/, 2022.
[30] J. Su, J. Scott, P. Hui, J. Crowcroft, E. de Lara, C. Diot, A. Goel, M. H. Lim, and E. Upton, "Haggle: Seamless networking for mobile applications," in *UbiComp 2007*. Springer, 2007, pp. 391–408.
[31] M. Caporuscio, P. Raverdy, and V. Issarny, "ubiSOAP: A Service-Oriented Middleware for Ubiquitous Networking," *IEEE Transactions on Services Computing*, vol. 5, no. 1, pp. 86–98, 2012.
[32] T. Kalbarczyk and C. Julien, "Omni: An application framework for seamless device-to-device interaction in the wild," in *Proc. International Middleware Conference*, 2018, pp. 161–173.
[33] Istio, "Service mesh," https://istio.io/, 2021.
[34] KubeEdge, "Edge Computing Framework," https://kubeedge.io, 2022.
[35] A. Ferreira, E. V. Hensbergen, C. Adeniyi-Jones, E. Grimely-Evans, J. Minor, M. Nutter, L. E. Peña, K. Agarwal, and J. Hermes, "SMARTER: Experiences with cloud native on the edge," in *USENIX Workshop on Hot Topics in Edge Computing (HotEdge)*, 2020.