Future Generation Computer Systems 157 (2024) 210–225

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

# Fluidity: Providing flexible deployment and adaptation policy experimentation for serverless and distributed applications spanning cloud–edge–mobile environments

Foivos Pournaropoulos *, Alexandros Patras, Christos D. Antonopoulos, Nikos Bellas, Spyros Lalis

*Department of Electrical and Computer Engineering, University of Thessaly, Sekeri & Heiden Str., 38334, Volos, Greece*

ARTICLE INFO

*Keywords:*
Microservices
Flexible deployment
Runtime adaptation
Edge computing
Serverless computing
Drones

ABSTRACT

We introduce Fluidity, a framework enabling the flexible and adaptive deployment of serverless and modular applications in systems comprising cloud, edge, and mobile nodes. Based on a declarative description of application requirements, a custom placement policy, and a formal system infrastructure description, Fluidity plans and executes an initial deployment of application components in the cloud–edge–mobile continuum. Furthermore, at runtime, Fluidity monitors resource availability and the position of mobile nodes, and adapts the deployment of the application accordingly, without any manual intervention from the application owner or system administrator. These characteristics render Fluidity an enabler for serverless applications, allowing the application developers to focus on the application code itself while abstracting out the infrastructure management. Notably, Fluidity permits developers to provide their own deployment and adaptation policies as well as to switch between different policies while the application is running. We discuss the design and implementation of Fluidity in detail and provide a realistic evaluation using a lab testbed in which the mobile node is represented as a simulated drone. In addition, we evaluate the scalability of the proposed mechanisms. Our results show that the core mechanisms of Fluidity can support flexible application execution at a reasonable overhead and experimentation with different deployment policies with minimal effort.

## 1. Introduction

The proliferation of serverless applications, usually comprised of several microservices, is significantly affecting the operation of modern cloud data centers. This paradigm enables a high degree of flexibility to the application developers who do not have to be concerned about the underlying infrastructure and also offers a more desirable cost model based on the execution time. Moreover, an increasing number of modern serverless workloads are container-based [1], allowing effortless deployment and migration. In addition, a plethora of applications are no longer restricted to the cloud but also involve mobile IoT devices, such as smartphones and vehicles like cars or even drones. In this case, edge computing can improve application Quality of Service (QoS) by moving computations closer to the points where data are produced while leading to better overall system performance and stability, as it reduces data traffic and resource pressure to the cloud.

However, to effectively harness the potential of serverless computing at the network edge, one needs to support flexible and adaptive deployment and orchestration of the application so that edge resources are used taking into account the dynamically changing position of the mobile nodes. Furthermore, it is important to be able to change the policy that drives such deployment decisions. For instance, these decisions can be taken by Machine Learning (ML) models which may need to be updated after retraining or even completely replaced at runtime.

In this work, we introduce Fluidity, a framework that enables flexible application deployment and orchestration across the entire system continuum. We support serverless and modular applications where the developer merely provides the individual components as containers and annotates them with their resource, deployment, and interaction requirements. Based on this information, Fluidity deploys the application components in the cloud, on edge nodes, and mobile IoT nodes, and adapts this deployment at runtime without any intervention from the application owner or system administrator. Notably, Fluidity is responsible for managing a single application on a slice of computational and sensing resources, which may span all layers of the continuum. The interaction between different concurrently executing

* Corresponding author.
*E-mail addresses:* spournar@uth.gr (F. Pournaropoulos), patras@uth.gr (A. Patras), cda@uth.gr (C.D. Antonopoulos), nbellas@uth.gr (N. Bellas), lalis@uth.gr (S. Lalis).

https://doi.org/10.1016/j.future.2024.03.031
Received 11 September 2023; Received in revised form 16 February 2024; Accepted 16 March 2024
Available online 21 March 2024

instances of Fluidity, managing different resource slices, is beyond the scope of our work.

Our framework is built as an extension to Kubernetes [2], which is used as the underlying mechanism for the basic pod/container deployment and health monitoring. Going beyond the standard functionality of Kubernetes, Fluidity supports flexible application deployment for mobile nodes and edge computing with transparent redirection of application traffic over different network interfaces supporting IP-based communication, which may employ different technologies at the physical layer.

The contributions of this paper are the following:

(i) We introduce a framework for the flexible and adaptive deployment and orchestration of serverless applications in the cloud–edge–IoT continuum, with a particular focus on mobile IoT nodes. We describe the internal operation of the framework and indicate how the execution of a deployment/adaptation plan can be parallelized to achieve scalable execution.
(ii) We outline the structured infrastructure and application descriptions used to capture the most significant characteristics of available nodes in the system, as well as the resource requirements and interactions of serverless application components. We cleanly separate, through a structured API, the core deployment mechanism from the policy that takes application placement decisions, making it possible to plug-in and switch between different policies at runtime.
(iii) We evaluate the functionality of the proposed framework using an indicative application running on a lab testbed. We measure the basic policy-independent overheads of the framework's mechanisms. The results confirm that these delays are sufficiently lightweight to enable a wide range of adaptation scenarios for applications that do not have strict real-time requirements. We showcase the ability to run the same application using two different policies and observe their impact on application's performance.
(iv) We evaluate the scalability of the core Fluidity deployment and telemetry mechanisms, showing that it is possible to support the deployment, scaling, and monitoring of a large number of application components with reasonable overhead.

The remainder of the paper is structured as follows. Section 2 discusses related work. Section 3 outlines an application example motivating the importance of flexible and adaptive application deployment in the cloud–edge–mobile continuum. Section 4 presents the design and implementation of the Fluidity framework, while Section 5 explains the scalability of the internal deployment/adaptation mechanism. Section 6 describes the policy API in more detail, and provides a concrete indicative policy example. Section 7 provides an evaluation of Fluidity in terms of the overheads of the core mechanism for an indicative drone mission and the policy-specific impact on the performance of the application. Section 8 evaluates the scalability of the Fluidity mechanism. Section 9 discusses the versatility of the proposed framework. Finally, Section 10 concludes the paper and provides directions for future work.

## 2. Related work

### 2.1. Offloading and flexible application deployment

***Offloading and initial deployment***. The work in [3] introduces the concept of service offloading from mobile devices to a powerful set of computers that are operating at the edge of the network, called cloudlets. The cluster uses VM virtualization to host the respective services. Offloading occurs whenever the mobile node is in the proximity range of the cloudlet to exploit communication locality. [4] focuses on offloading containerized computations of IoT applications to the gateways, as the latter usually remain underutilized. The authors leverage a centralized cloud-based approach for resource orchestration and management. The authors in [5] introduce trusted environments for container operation. However, the authors in [3,5] focus on automating the runtime synthesis of the execution enclaves to implement services, rather than the runtime management of prebuilt application components in the form of containers. With respect to [4,5], apart from supporting the initial application deployment, our main focus is on adapting it at runtime.

***Adaptive deployment***. In [6], the authors support the adaptive, cross-cloud deployment of applications using conventional and serverless components. The deployment is repeated whenever performance drifts from requirements. The framework in [7], supports multi-cloud applications consisting of components with various service levels. Deployment readjustment is supported by complex rearrangement rules, activated by events. The rules are converted to workflows, which can be modified in the course of time to optimize the used plan. Moreover, supervision of the application's precedent-recorded activity is also provisioned for possible reuse or deployment reshaping. [8] focuses on adaptive application deployment across multiple cloud providers. Information about the application's components and the interactions between them is captured using a graph structure. Compared with these works, our focus is on edge-oriented systems where part of the application runs on mobile nodes (such as drones) and the adaptation of deployment is due to mobility.

In [9], the authors focus on service migration for MEC environments with a deep reinforcement learning-based approach to provide a 5G edge infrastructure satisfying users' QoS demands. The main differentiation of our work is that we do not focus on a single policy, but provide a framework for policy experimentation across the continuum with minimal effort from the developers who can implement and plug-in different policies in a flexible way.

[10] introduces a message-oriented middleware for the efficient execution of FaaS workloads on edge-cloud environments. The authors optimize QoS by orchestrating QoS management mechanisms available in the underlying infrastructure. The optimization metrics are fixed (latency, jitter, and enquing time) while there is no support for different/user-provided policies. On the other hand, Fluidity supports flexible QoS-related adaptation, where the optimization targets are typically affected by the respective (custom) policy to meet each application's needs.

***Drone-related solutions***. Ad-hoc computation offloading from a drone to edge servers is studied in [11]. However, it assumed that all servers already have the required service pre-installed. Some works targeting drone orchestration and management offload all application intelligence to the cloud [12,13]. Others enable the native execution of container-based drone applications [14,15]. Our work extends these efforts, aiming at the adaptive end-to-end deployment for next-generation drone applications that can transparently manage resources across the entire edge-cloud continuum, not just the drone.

In [16], the authors present a framework for flexible application deployment in cloud–edge continuum systems with a focus on drones, for which it introduces special path specification directives and employs customized node allocation policies. There is a similar concept of hybrid components (that can run in the cloud and/or at the edge) but in this case, a greedy approach is adopted, by creating live instances on every edge node that has the required resources and is near the (estimated) path of the drone, while also keeping an instance in the cloud as a fall-back option. Also, application deployment is static and does not change at runtime. In contrast, Fluidity enables adaptive deployment during the execution of the application. Moreover, it supports a resource-efficient deployment of hybrid components by creating a single instance per component, which is dynamically relocated on different hosts based on the position of the mobile node. Last but not least, Fluidity separates the core deployment mechanism from the policy that takes deployment decisions, through an API that can be used to develop different policies, which can even be changed at runtime.

### 2.2. Network management

***5G.*** The 5G concept adopts a service-oriented view of the network to satisfy different and possibly contrasting requirements of a variety of applications [17]. Although 5G can benefit a wide range of mobile applications, including ones that employ drones, it necessitates extensive deployments by network operators that will take quite some time to achieve, particularly for 5G-core which supports low-latency applications. In contrast, Fluidity can transparently exploit different networking technologies, without any effort from the application developer.

***Service meshes.*** In Kubernetes, it is common to connect application components through service meshes, like Istio [18], which completely separate the application business logic from the communication logic by creating an abstracted application-aware overlay. These meshes, however, introduce extra overhead due to the injection of sidecar proxy containers in the application pods, which is more noticeable in resource-constrained environments. On the contrary, Fluidity supports traffic redirection over ad-hoc wireless network interfaces. This way applications can transparently benefit from the physical proximity of computing resources, enjoying higher bandwidth and lower latency vs. the default path of mobile connectivity.

***Kubernetes derivatives.*** Recent, lightweight, edge-oriented Kubernetes derivatives [19,20] allow application components to take advantage of various networking technologies. However, these approaches shift the responsibility of connectivity management to the application developer. In contrast, Fluidity can transparently take advantage of different ad-hoc networking capabilities under the hood.

### 2.3. Extensions on top of Kubernetes

It is important to stress that Fluidity is an extension on top of Kubernetes, not an alternative. More specifically, Fluidity relies on Kubernetes for pod deployment, basic inter-pod communication (using Flannel [21] as an overlay network), and resource/pod monitoring. In turn, Fluidity introduces the following extensions on top of Kubernetes: (i) support for multiple node communication interfaces; (ii) application-level traffic redirection between different interfaces; (iii) management of mobile nodes serving as hosts for application components; (iv) flexible deployment, driven by continuum-aware system and application descriptions; (v) support for different deployment policies, separated from the core deployment mechanisms. Below, we discuss some of these aspects in more detail.

Regarding the aspect of application-level communication, Fluidity can handle multiple network interfaces, in addition to the one that is used by default for control plane interactions, redirecting application traffic accordingly. This allows the application to exploit direct, low-latency, and high-throughput links to nearby edge nodes, instead of going through the public Internet. To be able to exploit such alternative communication paths, Fluidity considers the mobility of nodes that host application components and monitors their position at runtime.

The system infrastructure spanning the cloud–edge–mobile continuum as well as the application and its components are described via suitable Kubernetes Custom Resource Definitions (CRDs). This way, Fluidity can introduce all the structural elements necessary to support a declarative, intent-driven, deployment of the application in the continuum with the desired flexibility and runtime adaptation.

Last but not least, Fluidity allows developers to provide and experiment with diverse deployment policies (e.g., based on rules, heuristics, or ML). For maximum flexibility, policies are provided as plug-in code, which interacts with the core Fluidity mechanisms via a structured API. Also, the policy that will be used to drive application deployment is specified as part of the application description and can be changed at runtime by the system administrator or even the application itself.

Compared to the already existing automated scripts running on top of Kubernetes, the main aspects that differentiate our work are the following: (i) Fluidity contains its own internal state (internal representation of applications and nodes), where each continuum layer can be separately managed if desired, (ii) we provide a flexible and easy way for policy experimentation to the developers, abstracting out the overhead of the infrastructure management, and (iii) one can effortlessly exploit additional functionality under the hood on top of Kubernetes.

### 2.4. Extensions over our prior work

In our previous work [22], we presented an initial version of Fluidity. The new contributions reported in this version are the following: (i) We introduce the infrastructure and application description templates as well as the Policy API which can be used to develop custom plug-in application deployment policies for the Fluidity framework. (ii) We describe the implementation details of the proposed deployment mechanism for efficient execution of the supported adaptation patterns. (iii) We introduce the telemetry architecture, enabling application components to share application-level performance metrics with the plug-in policies so that the latter can be considered when making deployment decisions. (iv) We perform an experimental evaluation using two different deployment adaptation policies, validating the use of the Policy API and the ability of the framework to support data-driven policies while showing the impact on the application's performance. We also evaluate the scalability of both the deployment and the telemetry mechanisms.

## 3. Application example

To motivate the functionality we wish to support, we use a simple but indicative application and system infrastructure. We also use this as a running example throughout the paper, when presenting the design and implementation of the Fluidity framework and in the evaluation.

More specifically, we consider an application designed in a modular way, consisting of three distinct components that interact with each other as follows. The MobileViewer component takes pictures using the camera of a mobile node and sends them to the ImageChecker component, which processes them in order to detect objects of interest. If such objects are indeed detected in a picture, the ImageChecker sends it to the DataStore component for long-term storage and additional processing (not further discussed here). From a service-oriented perspective, the ImageChecker component provides a microservice that is invoked by the MobileViewer, and the DataStore component provides a microservice invoked by the ImageChecker.

We assume that the application should run on top of a distributed system spanning the cloud–edge–mobile continuum. For the sake of the example, we consider a system infrastructure that includes (i) a mobile node (in our example, a drone) with a camera and 4G and WiFi interfaces for Internet-based and direct wireless communication respectively, (ii) a powerful edge node connected to the Internet and featuring a WiFi interface for direct wireless communication, and (iii) a virtual node (VM) in the cloud with ample processing and storage capacity.

The MobileViewer component should run on the mobile node to access the camera, and the DataStore should run in the cloud to have ample storage space. However, the ImageChecker component can run either at the edge or in the cloud. The choice may be taken purely based on the location of the mobile node with respect to the edge node, or using more complex criteria such as the end-to-end invocation delay between the MobileViewer and the ImageChecker when the latter is hosted at the edge vs. the cloud as well as the cost for relocating the component from the cloud to the edge and vice versa. Fig. 1 illustrates the application deployment with the different placement options for the ImageChecker component. In the spirit of edge computing, running the ImageChecker on the edge node, close to the mobile node hosting

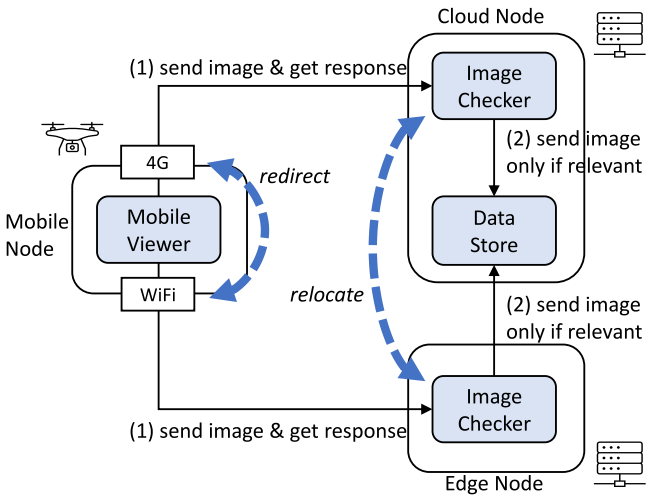



**Fig. 1.** Placement of serverless application components and dynamic relocation of the ImageChecker between cloud and edge with traffic redirection.

```
1  kind: Application
2  name: ObjectDetectionApp
3  components:
4  - name: MobileViewer
5    podSpec: PodMobileViewer.yaml
6    placement: mobile
7    systemServices:
8      camera:
9        methods: [CaptureImage, RetrieveImage]
10       sensorType: RGB
11   egress: [ImageChecker]
12 - name: ImageChecker
13   podSpec: PodImageChecker.yaml
14   placement: hybrid
15   ingress: [MobileViewer]
16   egress: [DataStore]
17 - name: DataStore
18   podSpec: PodDataStore.yaml
19   placement: cloud
20   ingress: [ImageChecker]
21 Policy:
22 - name: simple_policy.py
```

**Fig. 2.** Indicative application description (simplified).

the MobileViewer, can reduce the pressure on mobile communication and cloud resources. Furthermore, it can also improve the application-level interaction between these components, by redirecting application traffic over a direct link (for example WiFi) between the mobile node and the edge node, instead of going over 4G and the Internet.

It is important to stress that the above application and system infrastructure are intentionally kept minimal, for the sake of having a simple running example. Fluidity can support more complex and potentially serverless applications with a larger number of interacting components, as well as more complex system infrastructures with several mobile, edge and cloud nodes that can be used to host the components of such applications.

## 4. Design and implementation

Fluidity is designed to support the adaptive deployment of serverless applications in the cloud–edge–mobile continuum. Fluidity focuses on the seamless and transparent management of stateless components. The assumption is that the system administrator assigns specific nodes (or resource slices on more powerful nodes that can support sharing) to the application. The role of the Fluidity framework is to deploy the application while exploiting the available system resources and to dynamically adapt application deployment and orchestration as needed at runtime. Note that adaptation is a key requirement for applications spanning the continuum. This is because node/resource status changes in combination with node mobility can render any static application deployment highly non-optimal or even completely non-functional.

Importantly, Fluidity also allows to change the deployment policy at runtime. We currently support centralized policies. Namely, telemetry and status information is collected at a single controller that invokes the policy to analyze the data and take adaptation decisions, which are then executed by the Fluidity deployment and orchestration mechanisms. Once a deployment decision is taken, Fluidity can implement the respective plan in a scalable way by executing all independent actions in parallel, as we discuss in Section 5. In Section 8, we show that this leads to acceptable performance even for slices that include a relatively large number of hosts for application components. The following subsections describe the design and implementation of Fluidity in more detail.

### 4.1. Application model and application description

We consider modular applications consisting of distinct components that can be deployed separately on different nodes of the system. Each component is provided by the application developer in the form of a container. Notably, Fluidity's application description template is friendly to serverless applications, as the only information needed to be provided for a given component is the continuum layer selection. Thus, the application developer is unaware of the underlying infrastructure. Application deployment is driven through a structured description in YAML format. The description lists the application components, their desired placement in the continuum, their resource requirements, and the interactions between them. In turn, Fluidity plans, executes, and adapts the application's deployment, based on the nodes that are available in the system.

Fig. 2 shows the description of the example application that was introduced in Section 3. Each application component (MobileViewer, ImageChecker and DataStore) is associated with a pod manifest including the respective container image as well as the resource requirements in terms of (at least) CPU and memory. For example, the manifest for the ImageChecker component is PodImageChecker.yaml (line 13). Apart from the generic resource requirements declared in the manifest, a component may also need to use certain system-level services. For instance, the MobileViewer needs to access the `camera` service (lines 8–10) in order to capture the images that will be subsequently fed to the ImageChecker. In addition, every component is annotated with its desired placement in the continuum (lines 6, 14 and 19). Note that the ImageChecker component is annotated as `hybrid`, indicating that it may be placed in the cloud or at the edge. The interactions between the application components are captured in lines 11, 15–16 and 20. We refer to a component as service-providing, if it has an ingress relation with at least another component, while the latter is mentioned as a client component. This information is used to guide the redirection of application traffic in case application components are placed on nodes that can support direct communication. Finally, the deployment policy to be used is specified in the `Policy` field (lines 21–22). The application description is implemented as a Kubernetes CRD, registered with Fluidity (via Kubernetes). When Fluidity receives such a description, it starts the deployment of the application, using the specified policy. Note that the policy can be changed during the application's lifetime, by resubmitting the application description with a changed `Policy` field.

### 4.2. System infrastructure/node descriptions

Fluidity uses another Kubernetes CRD to capture the salient characteristics of nodes that can host application components. In particular, every node specifies its name, node type and continuum layer, availability status, CPU information, and available memory. Also, every

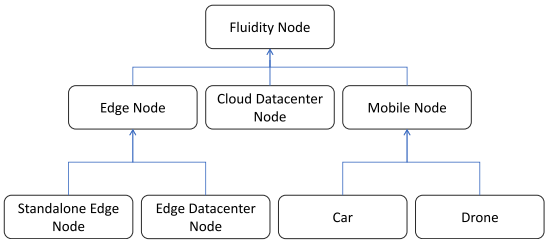



**Fig. 3.** Node ontology.

node declares its network interfaces and the respective connectivity provided through them. We adopt a subclassing approach, whereby specific node types and type-specific information can be added in a flexible way, capturing the special properties and capabilities of nodes in the different layers of the continuum. The class hierarchy for the different node types managed by Fluidity is shown in Fig. 3.

Fig. 4 shows the descriptions for the three nodes (drone, edge node and cloud node) of the example system infrastructure that was discussed in Section 3. For presentation purposes all three descriptions are given in a single figure; in reality, each description is a separate CRD. In the following, we briefly discuss each node in more detail.

As can be seen from its description, the drone node has an ARM Cortex-A53 CPU and features a 4G interface providing Internet connectivity (lines 8–9) and a WiFi interface that can be used to establish a direct link with edge nodes within a certain range (lines 10–12). Note that the 4G/Internet interface is used by default for both the Fluidity control plane and the application data plane. The drone also provides a camera service (lines 14–18), which can be used by application components running on the node. The description of the drone node has two extra fields to capture its current location and speed (lines 20 and 21 respectively). Given that these properties are dynamic, they need to be updated periodically.

The edge node description has similar but slightly different elements to capture the respective capabilities. More specifically, the node has an Intel Core i5 CPU and features an Ethernet interface providing Internet connectivity (lines 30–31) and a WiFi interface that can be used for direct communication with the drone (lines 32–38). For the latter, additional pairing information is provided so that other nodes can connect to the local wireless network. The edge node description also has a field to capture the physical location of the node (line 40). Unlike the drone node, this property is static – edge nodes are assumed to be fixed. Notably, the location information of the drone and edge node combined with the supported range of the network (in this case WiFi) can be exploited by the deployment policy to take smart placement decisions for `hybrid` application components (e.g., the ImageChecker) that can be placed more freely in the continuum, and to create ephemeral direct connections that can be used to redirect application traffic (e.g., the invocations performed by the MobileViewer to the ImageChecker).

Finally, the cloud node has an Intel Xeon E5 CPU and provides Internet connectivity via an Ethernet interface. In this case, however, there is no support for direct wireless communication with other nodes (over WiFi).

### 4.3. Software architecture

Fig. 5 depicts the software architecture of the Fluidity framework on top of a system infrastructure in the spirit of the example discussed in Section 3 and the node descriptions given in the previous section. The control plane runs on a separate node in the cloud. All nodes are connected through a VPN. The default Internet-based connection of the stationary nodes is via an Ethernet interface, while for the mobile node this is over a 4G link.

```
1  kind: FluidityNode
2  name: mobile-0
3  node-type: MobileNode::DroneNode
4  status: Available
5  CPU: ARM Cortex-A53 @1.20GHz
6  RAM: 1GB
7  network:
8  - interface: 4G
9    connectivity: Internet
10 - interface: WiFi
11   connectivity: direct
12   range: 50 # In meters
13 systemServices:
14     camera:
15       model: RPi Camera v2
16       methods: [CaptureImage, RetrieveImage]
17       resolution: 3280x2464
18       sensorType: RGB
19 DroneNodeSpecifics:
20   currentLocation: [lat, lon, alt]
21   currentSpeed: 4.0 # In m/s
22 ---
23 kind: FluidityNode
24 name: edge-0
25 node-type: EdgeNode::StandaloneEdgeNode
26 status: Available
27 CPU: Intel Core i5-4670 @3.40GHz
28 RAM: 5GB
29 network:
30 - interface: Ethernet
31   connectivity: Internet
32 - interface: WiFi
33   connectivity: direct
34   range: 50 # In meters
35   pairingInfo: {
36     mode: adhoc,
37     SSID: edge0-adhoc-ssid
38   }
39 StandaloneEdgeNodeSpecifics:
40   staticLocation: [lat, lon, alt]
41 ---
42 kind: FluidityNode
43 name: cloud-0
44 node-type: CloudDatacenterNode
45 status: Available
46 CPU: Intel Xeon E5-2620 v2 @2.10GHz
47 RAM: 16GB
48 network:
49 - interface: Ethernet
50   connectivity: Internet
```

**Fig. 4.** Indicative node descriptions (simplified) spanning the continuum.

The Fluidity Controller resides in the cloud, and is responsible for processing application deployment requests, finding a plan with a suitable component-to-node mapping and executing the deployment plan. The Controller monitors the state of the system and, if needed, adapts the current deployment. The application components are deployed on the selected hosts via Kubernetes as pods with the corresponding containers. We use K3s [23] which can run even on low-end devices, while all monitoring information is received via the standard Kubernetes API.

Each node that can act as a host for application components runs the Fluidity Agent. This registers the node with Kubernetes by sending a corresponding resource description, keeps track of the node's state and resource availability, and sends updates to the Kubernetes registry. In a mobile node, the Agent also periodically sends to Kubernetes its current position. Such updates are captured by the Controller, which,

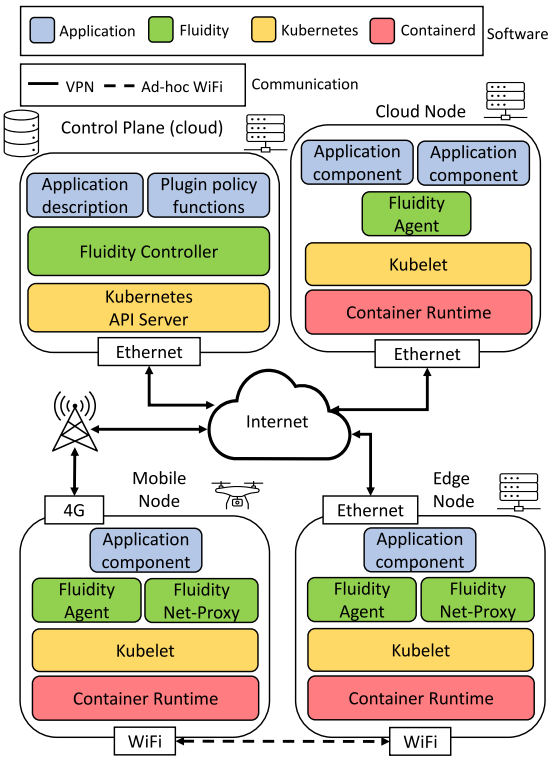



**Fig. 5.** Fluidity architecture and cluster configuration.

in turn, decides if it needs to adapt the deployment of the application. The Agent runs on the side of the Kubelet responsible for the local deployment and management of the pods. The runtime used to execute the application containers is containerd [24].

In addition to the Agent, mobile and edge nodes with a wireless (WiFi) interface run the Net-Proxy, which implements the redirection of application traffic from the default data path (used for all interactions with the control plane) over a direct wireless link (see Section 4.7).

### 4.4. Operation of the controller

The internal structure of the Controller is shown in Fig. 6. The Controller has two main threads, the Scheduler and the Monitor, which interact through a notification queue and in-memory data structures capturing the application deployment and current status. The Scheduler receives from Kubernetes application registration, modification, or removal events, initializes/updates its internal deployment/status data structures, and deploys or removes application components via Kubernetes as needed. The Monitor periodically queries Kubernetes to get the current status of the deployed pods and to update the availability/status of system resources. If significant changes occur, it notifies the Scheduler by posting events in the notification queue, to decide whether/how to adapt application deployment. Note that the Monitor can access and update the internal data structures for the current application deployment and status. Also, the events generated by the Monitor can point to specific parts of these data structures to set the Scheduler's focus on the deployment aspects relevant to the respective notifications.

The policy that initiates, decides and adapts the application deployment is abstracted via *initial_plan()*, *re_plan()* and *analyze_status()* functions. These functions take as input and produce as output standard Kubernetes and application-specific resources in the form of CRDs as well as Fluidity-specific data structures. The *analyze_status()* function is invoked from within the Monitor to examine the status of pods/resources and decide whether to generate a notification for the

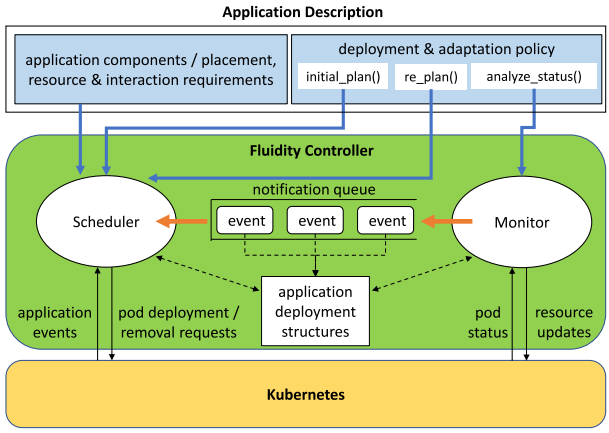



**Fig. 6.** Internal structure of the Fluidity Controller.

Scheduler. The *initial_plan()* and *re_plan()* functions are invoked from within the Scheduler to process application registration and modification requests. The latter also receives notifications generated by the Monitor, in order to produce the potentially updated application deployment plan (component-to-node mapping).

Notably, the above policy functions are not hardwired in the Fluidity framework, but are provided as part of the application description. These functions can be changed at runtime, by the user or the application itself, by updating the application description via Kubernetes. Thanks to this flexibility, each application can come with its own completely different deployment and adaptation policies, and switch between policies while it is running without having to stop/restart or suspend/resume. The policy API is described in more detail in Section 6.

### 4.5. Initial deployment

The procedure for initiating application deployment is briefly as follows. The user submits an application registration request to Kubernetes, which notifies the Scheduler thread of the Controller. The Scheduler retrieves/parses the application description and initializes the internal application data structures as well as the policy-specific functions. Then, it retrieves from Kubernetes the current state of the cluster and the resources available on each node, updates the application data structures, and invokes the *initial_plan()* function to produce the initial deployment plan. Finally, the Scheduler prepares the pod files for the application components, deploys the pods on the selected hosts via Kubernetes and starts the Monitor.

When the initial deployment is decided, the Controller ships the container image of each application component to the respective host. In the case of an edge or hybrid component, the respective container is sent to every edge node that can potentially act as a host for it. This is done proactively, as part of the initial deployment procedure, to accelerate placement adaptations that may be required in the future (the transmission of container images can take a very long time). However, in terms of actual deployment, only one instance of the edge/hybrid component is created in the system, on the host selected by the policy.

### 4.6. Adaptation of current deployment

Fluidity can support flexible adaptation scenarios, in particular, to handle (i) node mobility, (ii) the addition of a new edge node in the cluster, (iii) the removal or disconnection of an edge node from the cluster, and (iv) the failure of a pod hosting an application component. To this end, the Monitor retrieves the current status of the system's resources and passes this information to the *analyze_status()* function.

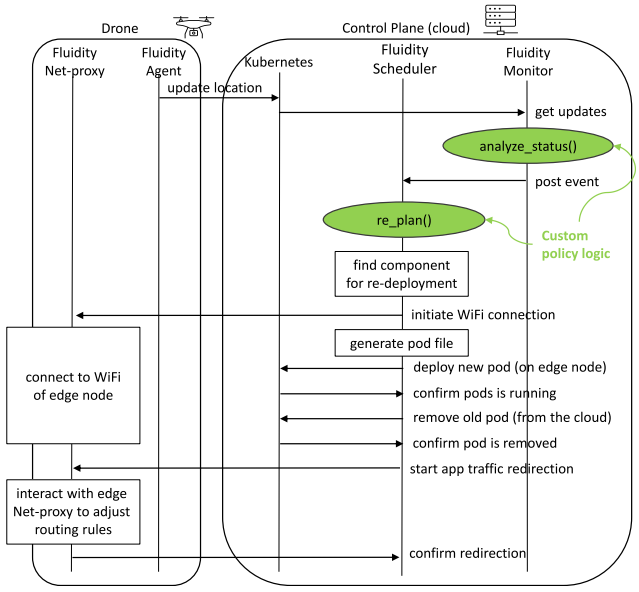



**Fig. 7.** Deployment adaptation process for a hybrid application component that is relocated from the cloud to the edge.

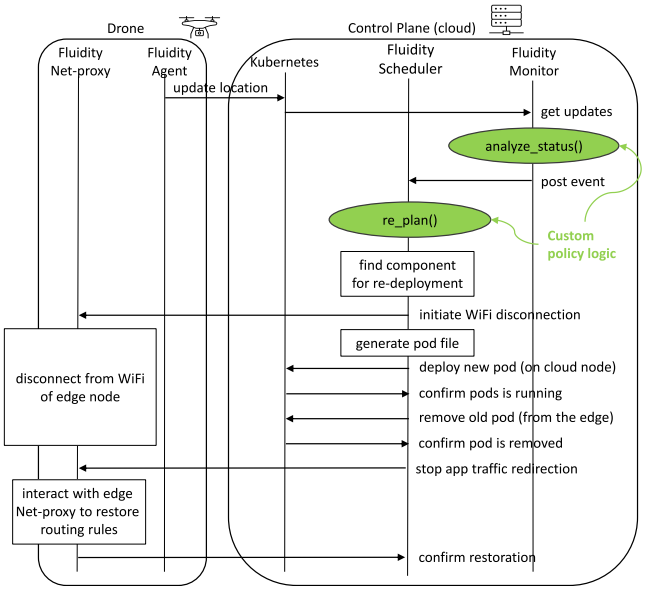



**Fig. 8.** Deployment adaptation process for a hybrid application component that is relocated from the edge to the cloud.

Moreover, Fluidity supports the explicit modification of the application description (including a change of the deployment policy) by the user or even by the application itself. These requests are captured and processed directly by the Scheduler, without the involvement of the Monitor.

In the following, to illustrate how the Fluidity framework works for a specific use case, we assume an application like the one described in Section 3 and discuss the case of deployment adaptation due to node mobility. For the sake of the example, we assume a policy that places the hybrid component (ImageChecker) on the edge node that is closest to the current position of the mobile node (drone) hosting the mobile component (MobileViewer). In the reverse scenario, when the mobile node moves away from the edge node, the ImageChecker is relocated back to the cloud. Figs. 7 and 8 show the basic steps of cloud-to-edge and edge-to-cloud relocations, respectively.

The mobile node Agent periodically sends its current location to Kubernetes. This is captured by the Monitor, which updates the application data structures and invokes the *analyze_status()* function to decide whether these updates warrant an adaptation of the current deployment. If so, a corresponding notification event is generated.

When the Scheduler receives such a notification event, it invokes the *re_plan()* function to produce an updated component-to-node mapping, if deemed necessary. If it is decided to change the current deployment, the Scheduler generates a new pod file for the hybrid component that should be relocated (ImageChecker), and implements the adaptation through Kubernetes, by deploying the new pod on the selected host and then removing the unwanted pod from the old host.

### 4.7. Redirection of application traffic

Fluidity actively exploits the ability of the mobile node to interact with an edge node via direct wireless communication (including but not limited to WiFi), rather than via the default communication path (4G and the public Internet). To this end, additional actions are performed depending on whether the hybrid component is relocated (i) from the cloud to an edge node, (ii) from an edge node to the cloud, or (iii) between two edge nodes. We describe the basic functionality using WiFi merely as an example of the secondary communication interface, however, different network technologies can be used instead.

In the first case, shown in Fig. 7, the Scheduler instructs the Net-Proxy of the mobile node to activate its WiFi interface and connect to the wireless network of the edge node (sending the necessary information, such as the SSID and key of the network). This is done before the generation of the pod file and the deployment on the edge node. In this way, the WiFi connection delay overlaps with the component deployment delay. Once the old pod is removed, the Scheduler instructs Net-Proxy to redirect application traffic to the component instance on the edge node over WiFi. In turn, the Net-Proxy of the mobile node interacts with the Net-Proxy of the edge node (not shown in the figure) to jointly set/adjust the routing rules that are associated with the hybrid component.

In case the hybrid component is relocated from the edge back to the cloud, as shown in Fig. 8, the Scheduler instructs the Net-Proxy on the mobile node to disconnect from the edge WiFi network. The WiFi disconnection completely overlaps with the pod-file generation and pod deployment/removal, in a similar manner as the WiFi connection in the previous scenario. Subsequently, the mobile node's Net-Proxy is notified to redirect application traffic, by restoring the routing rules. As above, this is done after removing the component instance from its old host.

A similar process is followed when the hybrid component is relocated between two edge nodes. More specifically, after deploying the pod on the new edge host and removing the pod from the old one, the Scheduler instructs the Net-Proxy to redirect application traffic. In this case, the Net-Proxy configures the WiFi to connect to the network of the new edge host and updates the routing rules for the communication between the application components.

## 5. Scalable execution of deployment plans

To execute the plan produced by the policy in a scalable way, Fluidity exploits opportunities to overlap individual actions. Plans typically consist of one or more of the following patterns: (i) Relocate service-providing and/or client components, optionally with application-level traffic redirection, (ii) Scale-out components by creating instances on selected hosts, and (iii) Scale-down components by removing instances from selected hosts. Individual actions within each of these patterns can be executed in parallel, as discussed in the following sections.

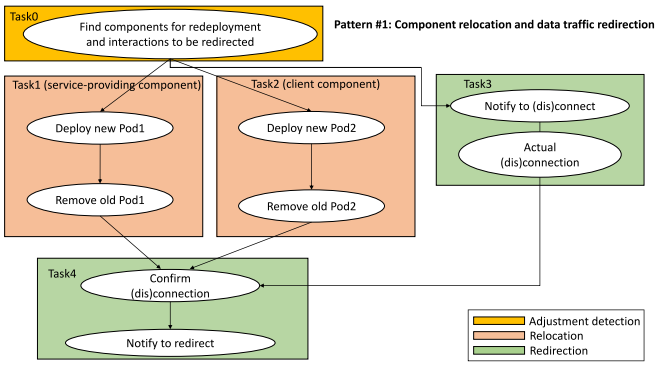



**Fig. 9.** Pattern #1: Relocation of service-providing and client components with application traffic redirection. There are simpler versions that include only a subset of those actions.

### 5.1. Relocation of components & traffic redirection

For any component, its relocation consists of the deployment of the new Pod instance and the subsequent removal of the unneeded (old) one. As mentioned in the previous section, these actions are executed in sequence for each component, to reduce the risk of failed invocations. Depending on the plan, one may need to relocate only the service-providing component, only the client component, or both. Also, if application traffic redirection is needed, this must be initiated after the connection/disconnection to/from the respective ad-hoc wireless network and the relocation of the components. However, the former can be performed in parallel with the component relocation.

The most complex scenario of this pattern is depicted in Fig. 9. We label the respective delays of each action as follows: (i) $D_{plan}$: The overhead of identifying the required plan adjustments. (ii) $D_{relocS}$, $D_{relocC}$: The component relocation overhead of the service-providing and client components, respectively. (iii) $D_{wnet}$: The overhead of connection/disconnection to the wireless ad-hoc network provided by an edge node. (iv) $D_{redir}$: The overhead of traffic redirection. The total delay is:

$$D_{total} = D_{plan} + max(D_{relocS}, D_{relocC}, D_{wnet}) + D_{redir}$$

Note that, as mentioned, the connection/disconnection to the ad-hoc wireless network of the edge node can be performed in parallel with the relocation of the components.

If any of the optional actions are not present in the plan, the total delay can be calculated by omitting the respective delay factors. For instance, if a service-providing component should be relocated without relocating the client component but with traffic redirection, the total delay for that adaptation would be:

$$D_{total} = D_{plan} + max(D_{relocS}, D_{wnet}) + D_{redir}$$

### 5.2. Scale-out/scale-down component

The last two patterns are symmetrical in terms of functionality. When scaling-out a component, the deployment of each new instance (Pod) can be executed in parallel. The same applies to the scaling-down case, for the respective Pod removals. In the general case, where $N$ Pods should be deployed/removed, the total delay is:

$$D_{total} = D_{plan} + max(D_1, D_2, \ldots, D_N)$$

where $D_{plan}$ is the overhead of identifying the components to be adjusted (similar to above) and $D_i$ is the deployment/removal delay of the $i$th Pod ($1 \leq i \leq N$) on/from the respective host.

```python
# Produce initial deployment plan
def initial_plan(app_desc, nodes):
    # Custom logic implementation
    return plan, context

# Analyze updated resources/nodes
def analyze_status(app_desc, nodes, context, system_metrics,
    updated_nodes, curr_deployment):
    # Custom logic implementation
    return notify, context

# Produce a (possibly) new deployment plan
def re_plan(old_app_desc, new_app_desc, context,
    curr_deployment):
    # Custom logic implementation
    return new_deployment, context
```

**Fig. 10.** Functions of the policy API.

## 6. Policy API

This section describes the API through which a custom policy can interface with the rest of the Fluidity framework. Fig. 10 provides the templates/prototypes for the functions that need to be implemented by the policy developer. These are discussed in more detail below.

### 6.1. Context parameter

The *context* parameter can be used by the policy to store arbitrary policy-specific information so that this can be shared between the invocations of the plug-in functions. The developer can define the form and content of the context data structures according to the needs of the policy.

The context of the policy is initialized and returned by the *initial_plan()* function. It is then passed as input to the *analyze_status()* and *re_plan()* functions. Note that both functions may, in turn, modify/update the context, which is why they return it as output.

### 6.2. Initial_plan function

The *initial_plan()* function handles the first deployment of the application. It takes as input: (i) The application description in the form of Fig. 2, which can be parsed to initialize and fill in as needed the internal data structures that are part of the policy context. (ii) The dictionary of available nodes, segregated by node type, as provided by the Fluidity framework. This contains the names of the different node types as keys, while the respective values are individual lists storing node descriptions falling into each node type category. The function produces as output the initialized policy context and the first deployment plan for the application. The latter is the component-to-node mapping to be executed by the Fluidity Controller. It is a dictionary that stores for each application component the node selected to host it.

### 6.3. Analyze_status function

The *analyze_status()* detects potential resource updates that can trigger an adaptation of the application deployment. It takes as input: (i) The application description (as above). (ii) The nodes dictionary (as above). (iii) The policy-specific context that was returned by the last invocation of either the *initial_plan()*, *re_plan()* or *analyze_status()* function. (iv) A read-only structure maintained by Fluidity with system-level metrics about the overhead of the core Fluidity mechanisms, so that this can be considered by the policy to take more sophisticated decisions. More specifically, this is a dictionary having as keys the source and destination nodes for a given component relocation, providing for each such pair the total number of relocations, the average relocation delay, and the individual delay for each relocation that was performed.

(v) A separate node dictionary, storing only the nodes with an updated state/status, which could potentially lead to an adjustment of the deployment plan. (vi) The current deployment plan, which stores the current component-to-node mapping. The function returns a boolean notification flag and the context. If there are no (significant) updates that may trigger an adaptation, the returned notification flag is false. This way, the Fluidity Monitor knows that it does not need to generate an update event for the Scheduler.

### 6.4. Re_plan function

The *re_plan()* function is responsible for adapting the deployment of the application, by producing a new component-to-node mapping. It takes as input: (i) The old (potentially outdated) application description. (ii) The new (updated) application description. This parameter is valid only in case Fluidity receives an updated application description (via Kubernetes). If the adaptation is triggered by the *analyze_status()* function, this parameter is void. (iii) The policy-specific context that was returned by the last invocation of the *analyze_status()* function. (iv) The current deployment plan.

The function returns the context and the new deployment plan to be executed by the Controller. The latter is a dictionary storing for each component the desired actions to be performed on one or more nodes. The possible actions are: (i) *Deploy*, if the pod should be deployed to the node. (ii) *Remove,* if the pod is no longer needed on the node, and should be removed. (iii) *Move,* if the pod should be relocated from its old host to a new one. As discussed in Section 4.6, Fluidity implements relocation as a combination of deployment and removal on the respective hosts. In this case, the policy can also specify if it is desirable to perform application data redirection, as discussed in Section 4.7; Fluidity checks whether this action is applicable based on the capabilities of the nodes that host interacting application components (else the hint is ignored). Note that the *re_plan()* function may decide to keep the same deployment as before. In this case, the returned new deployment plan is void, hinting the Fluidity Scheduler not to perform any component deployment, removal, or relocation.

### 6.5. Policy example

To provide a simple policy example, going back to the application described in Section 3, assume we wish to move the ImageChecker component close to the MobileViewer each time the mobile node enters the range of an edge node that can act as a host of the ImageChecker, and conversely to move the ImageChecker back to a (default) cloud-based host when the mobile node exits the range of the edge node.

Algorithm 1 gives a high-level description of such a policy in the form of pseudocode. The adaptation logic is in the *analyze_status()* and *re_plan()* functions. The first function analyzes the position updates of the mobile node hosting the MobileViewer and generates adaptation events each time the node enters or exits the range of an edge node. The second function translates these events into corresponding relocation actions for the ImageChecker component and produces the adaptation plan to be executed by Fluidity. Note that the mobile node may pass above two neighboring edge nodes with overlapping ranges, in which case when exiting the range of the one node it will immediately enter the range of the other. Of course, this policy is very simple and may lead to suboptimal performance, e.g., if the mobile node remains in the range of the edge node for a relatively short time vs. the time it takes for Fluidity to execute the requested adaptation. We discuss a more sophisticated policy in Section 7.2.

**Algorithm 1** High-level description of a simple location-based policy for the relocation of the ImageChecker component

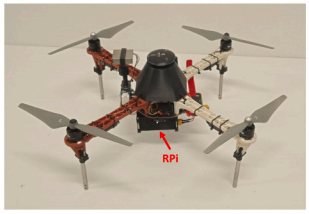
```
1: function INITIAL_PLAN(app_desc, nodes)
2:     context ← CreateContext(app_desc, nodes)
3:     plan ← CreateEmptyPlan()
4:     for each c ∈ app_desc do
5:         n ← FindCandidateHost(c, nodes)
6:         a ← DeployAction(c, n)
7:         AddAction(plan, a)
8:     end for
9:     return(plan, context)
10: end function

11: function ANALYZE_STATUS(app_desc, nodes, context, system_metrics,
                            updated_nodes, curr_deployment)
12:     notify ← False
13:     UpdateContext(context, nodes, curr_deployment)
14:     h ← GetHost(curr_deployment, "MobileViewer")
15:     if h ∈ updated_nodes then
16:         e ← EdgeNodeBoundaryEvent(context, h)
17:         if e ≠ NULL then                          ▷ wireless boundary crossed
18:             AddEvent(context, e)
19:             notify ← True
20:         end if
21:     end if
22:     return(notify, context)
23: end function

24: function RE_PLAN(old_app_desc, new_app_desc, context,
                     curr_deployment)
    ▷ Assuming new_app_desc is void
25:     c ← GetComponent(old_app_desc, "ImageChecker")
26:     h ← GetHost(curr_deployment, "ImageChecker")
27:     e ← GetEvent(context)
28:     new_deployment ← CreateEmptyPlan()
29:     if e.type = ENTER then
30:         a ← RelocateAction(c, h, e.node_entered)
31:     else if e.type = EXIT ∧ e.node_exited = h then
32:         n ← FindCandidateHost(c, context)
33:         a ← RelocateAction(c, e.node_exited, n)
34:     else if e.type = EXIT_ENTER ∧ e.node_exited = h then
35:         a ← RelocateAction(c, e.node_exited, e.node_entered)
36:     end if
37:     AddAction(new_deployment, a)
38:     return(new_deployment, context)
39: end function
```

```
1 # push telemetry information to the system
2 def push_metric(metric_id, value)
```

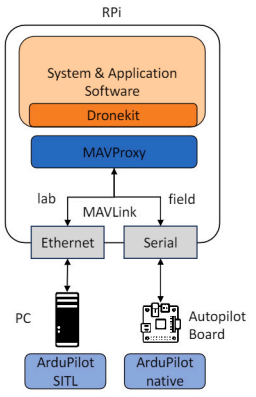
**Fig. 11.** Application telemetry API.

### 6.6. Application-level telemetry

Apart from metrics visible at the system-level that are provided by Fluidity, such as the overhead of specific adaptation actions, policies may be also driven by metrics quantifying aspects of system operation from the application perspective (such as application uptime, application throughput, application latency, etc.). To this end, we provide an API call for application components to communicate such metrics to the policy. Fig. 11 shows the call made available to application components to push application-level telemetry information to the system. The first parameter corresponds to a metric ID, which is merely a tag agreed between the application and the policy for each metric. The second parameter is the actual metric value pushed to the system. This call is implemented in a library (provided by Fluidity) that has to be installed in the container of the application component. In terms of implementation, we follow the OpenTelemetry specification [25] and employ an OpenTelemetry collector [26] to operate in agent mode on each node of the cluster. Each collector receives (via the Fluidity library) custom metrics from the respective application components running locally on the node and pushes them to an OpenTelemetry collector which runs in gateway mode at the control plane. The latter stores the telemetry data in a local Prometheus database [27]. The




(a) Custom drone with RPi.




(b) RPi in the lab testbed vs the drone.

**Fig. 12.** RPi usage as drone companion board vs. the testbed setup.

policy, in turn, can pull metrics from the database by communicating with a global endpoint via Prometheus Query Language [28].

To give an example of telemetry usage in the application given in Section 3, one may wish to monitor (and optimize) the rate at which the MobileViewer invokes the ImageChecker component, which requires measuring the end-to-end invocation delay at the level of the application. This can be done by the MobileViewer component using the *push_metric* API call, for a metric labeled "InvocationDelay" with the recorded invocation delay as the value. The MobileViewer could generate this information after every call or report the average over a time window. This metric, in combination with the recorded adaptation delay, can be used to implement more sophisticated policies, such as the one we discuss in Section 7.2.

## 7. Functional evaluation

In this section, we evaluate the core functionality of Fluidity using the application example described in Section 3. For the purpose of our experiments, the MobileViewer component simply invokes the ImageChecker in an endless loop. Also, the component uses the Telemetry API to report the end-to-end invocation delay. Finally, a timeout is used to abort invocations that are taking too long; we refer to those as failed invocations.

We use two simple policies to validate the flexibility of Fluidity and to illustrate that the core mechanisms and policy API can be used to support different adaptation strategies that have different effects on application performance. Note, however, that our focus is mainly on the overhead of the adaptation mechanisms, not the policies themselves. For practical reasons, we use a lab testbed where the MobileViewer resides on a drone companion computer. We do not use a drone that actually flies, but the flight behavior is realistically simulated using a Software-In-The-Loop (SITL) configuration of the autopilot.

In the following, we describe the cluster configuration of the testbed and the two deployment policies in more detail. Then, we present the test scenario used to validate the Fluidity framework and discuss the results.

### 7.1. Lab testbed setup

The cluster configuration used in our experiments is similar to that of Fig. 5. The edge node functionality is implemented as a VM with 2 cores and 5 GB of memory, running on a laptop with a dual-core Intel Core i5-7200U CPU (@2.5 GHz). The control plane and cloud node are two separate VMs with 4 cores and 16 GB memory each, running on nodes with Intel Xeon E5-2630 0 (@2.30 GHz) and Intel Xeon E5-2620 v2 (@2.10 GHz) CPUs respectively, in the compute cluster of our department.

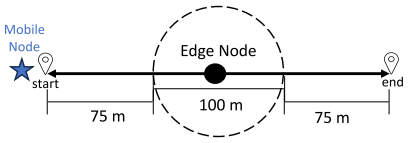



**Fig. 13.** Scenario leading to repeated relocations of the ImageChecker between cloud and edge (with traffic redirection).

**Table 1**
Testing-mission phases.

| Phase number | Round-trips | Velocity (m/s) |
|---|---|---|
| #1 | 1 | 2 |
| #2 | 5 | 10 |
| #3 | 1 | 2 |

The mobile IoT node is a Raspberry Pi 3 Model B (RPi), with a quad-core ARM Cortex-A53 CPU @1.2 GHz and 1 GB of memory. This is used as a companion computer in our custom drone, shown in Fig. 12(a), so that we can run our own system and application software independently of the autopilot subsystem. In our lab experiments, the RPi is connected to the SITL configuration of the Ardupilot autopilot [29] running on a separate machine, which simulates the dynamics/movement of the drone. The software on the RPi talks to the autopilot using the MAVLink protocol [30] via the MAVProxy library [31] and the DroneKit software [32], the only difference is that this communication takes place over LAN and the Ethernet interface, whereas in the real drone setup the RPi is connected to the autopilot board over the serial interface. Fig. 12(b) illustrates the SITL-based configuration used in the lab vs. when using a real drone.

Through several previous experiments [16], we have confirmed that this setup faithfully reproduces the behavior of the real drone. In particular, the difference in the time it takes for the drone to move between waypoints at a given target cruising speed in reality vs. SITL is less than 10% on average. Also, due to uncontrollable variations in the weather/wind conditions, field tests make it hard if not practically impossible to reproduce the same flight behavior, which is needed for a fair comparison between different approaches. For this reason, we perform our evaluation using the SITL configuration.

Connectivity with the control plane is via a VPN, over 4G for the drone (RPi), and over Ethernet for the edge node (laptop). This is also the default connectivity path for application-level traffic. The direct wireless communication between the RPi and the edge node, activated when the ImageChecker component runs on the latter, is over ad-hoc WiFi. The nodes hosting application components run the containerd runtime, Kubelet, and the Fluidity Agent. The RPi and laptop also run the Fluidity Net-Proxy. The Agent on the RPi interacts with the (SITL) autopilot to retrieve the node's current position which is then forwarded to the Fluidity Controller.

### 7.2. Plug-in policies

We run experiments using two different policies that guide the adaptation of application deployment (in this case, the placement of the ImageChecker component). Both policies rely on live information (resource updates) to make deployment decisions. One of them gathers and manages performance data as part of its internal context. This validates the ability of Fluidity to support diverse policies and also allows us to quantify the impact of different adaptation strategies on the performance of the application.

#### 7.2.1. Naive policy

The first policy is trivial. It merely detects when the mobile node enters or exits the wireless communication range of an edge node to trigger adaptation. In the first case, a relocation of the ImageChecker

**Table 2**
Major components of adaptation overhead. Note that the WiFi connection and WiFi disconnection delays overlap with other delay components, therefore they do not affect the total adaptation delay.

| Name | Description | Average delay (ms) |
|---|---|---|
| (1) Resource update | Time needed for the Monitor to update the resource status. | 94 |
| (2) Notification decision | Time needed by the policy-specific *analyze_status()* function to update the context and decide whether to notify the Scheduler. | 1.9 (naive policy)<br>24.4 (data-driven policy) |
| (3) Notification | Time needed for the Scheduler to receive the notification of the Monitor. | 0.7 |
| (4) Planning | Time needed by policy-specific *re_plan()* function to update the context and produce the new component-to-node mapping. | 27.3 (naive policy)<br>30.2 (data-driven policy) |
| (5) WiFi connection | Time needed for the drone to connect to the WiFi network of the edge node (cloud-to-edge relocation). | 156.4 |
| (6) Pod-file creation | Time needed to build the pod-related files for new component instances. | 3.9 |
| (7) Pod deployment | Time needed to deploy the new pod & confirm this is running. | 1659 |
| (8) Pod removal | Time needed to remove the unwanted pod & confirm this has terminated. | 58 |
| (9) Redirection | Time needed to redirect application traffic (from 4G to WiFi or vice versa, depending on relocation direction). | 3022 |
| (10) WiFi disconnection | Time needed for the drone to disconnect from the WiFi network of the edge node (edge-to-cloud relocation). | 536.4 |

**Table 3**
Application metrics.

| Name | Description |
|---|---|
| Invocation delay | The time needed for the MobileViewer to invoke the ImageChecker and get a response. |
| Invocation failure ratio | The number of failed invocations over the total number of invocations attempted while the drone remains in the range of the edge node, including an additional 10-s time window after the edge-to-cloud transition. |

from the cloud to the specific edge node takes place. In the second case, where the mobile node exits the range of the edge node that hosts the ImageChecker, the latter is re-instantiated to the cloud. The high-level logic of the policy is given in Algorithm 1. Note that the only input the policy needs is the current position of the drone node and the location and communication range of the available edge nodes. Also, the policy always activates a redirection of application traffic each time it decides for a relocation of the ImageChecker component.

#### 7.2.2. Data-driven policy

A more sophisticated, yet still simple adaptation logic is employed by the second policy. More specifically, the policy considers past monitoring data to capture (i) the application-level average ImageChecker invocation delay when instantiated on the edge- and the cloud-node, as well as (ii) the application reconfiguration overhead (Fluidity-induced adaptation delay), to let the policy take more educated decisions. The former is stored separately for different cluster nodes acting as hosts for the ImageChecker component, while the latter is maintained per source and destination hosts of the respective component relocation.

The policy estimates the time the drone will remain inside the communication range of an edge node and decides if a potential relocation to the edge will benefit the application. To estimate the time that will be spent inside the wireless range of an edge node, (i) it is assumed that the drone will maintain the average speed based on the last two status updates, and (ii) the trajectory the drone will travel across the edge node's coverage area is estimated (again) using the drone's last two consecutive position updates. For the sake of the evaluation, the policy considers a relocation to the edge to be beneficial if it is expected to lead to a higher total number of invocations performed while being in the range of an edge node with direct communication capabilities, compared with leaving the ImageChecker component on the cloud and communicating with it over 4G, without however paying the overhead/application downtime due to adaptation. Like the naive policy, the data-driven policy activates traffic redirection on each relocation.

### 7.3. Test scenario

To validate the flexibility of Fluidity and to quantify the adaptation overhead using its mechanisms, we need to evaluate it using a setup in which the core functionality will be triggered. We opt for a simple scenario (drone flight plan), where the mechanisms are stressed with a high number of adaptation phases/relocations, overheads can be properly attributed, and the results are easily explainable. The simplicity of the scenario does not restrict the usage of Fluidity. The framework can be tested with a plethora of flight scenarios/missions, however, the basic elements that will be quantified are practically equivalent.

Fig. 13 illustrates the scenario used to evaluate our implementation. The (virtual) drone moves between two locations in a straight line, while the edge node is (virtually) placed at the mid-point of this path. The start and end points of the drone's path are set 250 m apart, while the WiFi range of the drone and edge node is set to a radius of 50 m. This way, the drone's path has 75 m where the drone is out of range of the edge node, followed by 100 m in range of the edge node, and then another 75 m where the drone is again out of range.

Each mission includes 7 round-trips, from the start to the end point and back, and during each round-trip, the drone enters the range of the edge node twice. Also, the mission consists of three consecutive phases, for a pre-specified number of round-trips with different but constant drone speeds, as shown in Table 1. We execute and evaluate this mission twice, one for each of the two different policies. Note that the data-driven policy needs to initially collect some data in order to start taking informed decisions. This is accomplished by "forcing" the first four relocations of the ImageChecker (from the cloud to the edge and vice versa) in the spirit of the naive policy. This applies only to the first round-trip. For the rest of the mission, the data-driven policy takes decisions based on previously collected system- and application-level data. In the second phase, the flight speed of the drone is increased and in the third phase it is restored to the initial setting, to evaluate the impact of the decisions taken by the data-driven policy vs. the naive policy.

### 7.4. Adaptation overhead

To capture the overhead of the Fluidity framework, we record the major delay components, summarized in Table 2. All delays are measured at the Controller, except (5) and (10) which are measured on the drone node (RPi). Also note that each of those delay components overlaps with (6)–(8), depending on the direction of relocation. As discussed in Section 4.7, when the ImageChecker is relocated from the cloud to the edge, or vice-versa, the Net-Proxy connects to/disconnects

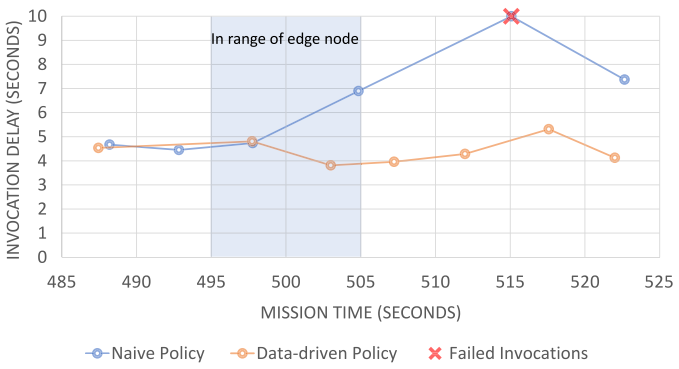



**Fig. 14.** Invocation delays for a pass over the region of the edge node, during the second phase of the mission where the drone maintains a speed of 10 m/s. In this case, the data-driven policy does not relocate the ImageChecker to the edge. As a result, the MobileViewer performs more invocations compared to the naive policy.

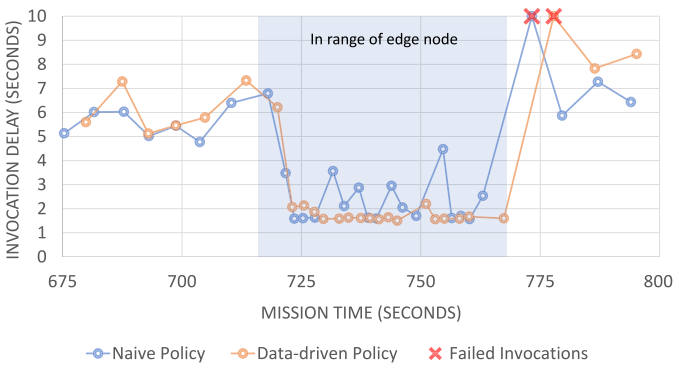



**Fig. 15.** Invocation delays during a pass over the region of the edge node, during the third phase of the mission where the drone maintains a speed of 2 m/s. Both policies take the same decisions in terms of relocating the ImageChecker from the cloud to the edge node and vice versa.

from the WiFi of the edge node after the respective notification is received from the Scheduler. Since this is performed simultaneously with the rest of the actions taken by the Fluidity mechanisms, the WiFi connection/disconnection delays are fully overlapped with the new pod deployment and old pod removal phase, without affecting the actual adaptation delay.

The reported overheads are the averages over the entire execution during the two mission experiments. Note that the "Resource Update" overhead is measured periodically, each time Fluidity Monitor performs a resource check, whereas the "Notification Decision" overhead is captured whenever the Monitor detects a resource modification. All other overheads are measured only when there is an adaptation to the deployment plan. For the naive policy, we report the average for a total of 28 relocations of the ImageChecker component from the cloud to the edge and vice versa, which are performed in the context of the 7 round-trips of the mission. The data-driven policy, in turn, decided to initiate only 8 relocations during the 7 round-trips, thus we report the average delays from those. From all the experiments we have performed, we observed that the overhead of the core Fluidity mechanisms is not affected by the selected plug-in policy (as expected). For this reason, for all policy-independent delay components we report the average of the experiments for both policies (a total of 36 relocations).

The average adaptation delay (excluding the WiFi connection and disconnection, which are performed concurrently with other actions) is roughly 4.9 s. Clearly, the most significant overhead components are application traffic redirection and pod deployment, accounting for 62% and 34% of the total delay, respectively. The overall delay is acceptable for applications that do not have stringent real-time requirements. Notably, the policy-related overhead is merely 0.6% and 1.1% of the total adaptation overhead for the naive and the data-driven policy, respectively. This includes the execution time for both `analyze_status()` and `re_plan()` plugin functions. We have also measured the overhead of switching to a different adaptation policy, once the Fluidity Controller receives such a request via Kubernetes after a respective modification of the application description. The average delay is 25 ms (over a total of 50 policy switches that were performed independently of the above experiments). This overhead is sufficiently low to allow for frequent policy switches, e.g., in order to adopt different strategies at various phases of execution (and locations of the mobile nodes) or to incrementally evolve a strategy through gradually improved ML models.

### 7.5. Application performance

At the application level, we monitor the metrics listed in Table 3, which capture the main characteristics of ImageChecker invocations performed by the MobileViewer component. We measure the end-to-end invocation delay of the ImageChecker, from the time when the MobileViewer performs the call to the time when the call returns. We also measure the invocation failure ratio at the critical points of application execution where the ImageChecker relocates between the cloud and the edge.

Fig. 14 plots the invocation delay experienced by the application for an indicative pass through the coverage region of the edge node (grey area) when using each of the two adaptation policies. This particular snapshot is taken from the second phase of the mission, where the drone has a cruising speed of 10 m/s. For comparison, Fig. 15 depicts a snapshot from the third phase of the mission, where the drone maintains a lower speed equal to 2 m/s. Please note that, since the mobile node speeds are different in the parts of the mission corresponding to each of the two figures, equal times do not correspond to equal distances traveled by the mobile node. Moreover, the horizontal (time) axis scale is different in each figure. The radius of the range of the edge node is the same in both cases. Similarly, one should consider those differences when comparing the invocation rates in the two figures. For example, the invocation rate using the naive policy is essentially the same in both cases.

Note that some invocations actually fail (time-out), denoted with "X" symbols. Such failures occur only at the borders of the region where the relocation of the ImageChecker takes place. They can be caused by the removal of the ImageChecker instance from its old host, which may still be running a previous invocation (we have confirmed that such failures equally occur without application traffic redirection). Another cause for such failures is a WiFi disconnection as the drone moves away from the range of the edge node. Nevertheless, such invocation failures are not fatal for applications in the form of microservices, as the call will be repeated after the timeout.

#### 7.5.1. Invocation delay

Over all experiments including more than 300 invocations, the average invocation delay to the edge-located ImageChecker is 1.89 s vs. 5.29 s to the ImageChecker running in the cloud, an improvement of 64%. Notably, the actual computation takes about 1.03 s on the laptop (edge node) vs. 0.80 in the cloud VM, which is almost 20% faster. However, the lower data transfer times over WiFi vs. 4G (due to lower latency and higher throughput) make the difference in favor of the edge. Obviously, edge computing would be even more attractive with a more powerful edge node. Since in our experimentation we use real wireless channels for both communication paths (4G and WiFi), the invocation delay can suffer from fluctuations due to the time-varying characteristics of the channels. For example, observe time points 505 in Fig. 14 and 755 in Fig. 15 under the naive policy.

**Table 4**
Nodes (VMs) used for the scalability experiments.

| Name | Host machine CPU model | vCPUs | Memory (GB) |
|---|---|---|---|
| VM0; VM1; VM2/VM3 | AMD EPYC 7513 32-Core CPU @2.60 GHz | 64; 32; 16 | 64; 64; 32 |
| VM4;VM5;VM6 | 13th Gen Intel Core i9-13900KF @5.80 GHz | 16; 16; 8 | 32; 16; 16 |
| VM7;VM8;VM9 | Intel Xeon CPU E5-2695 v3 @2.30 GHz | 16; 16; 8 | 32; 16; 16 |
| VM10 | Intel Xeon E5-2620 v2 @2.10 GHz | 4 | 16 |

### 7.5.2. Policy comparison

To compare the two policies, we measure the total number of successful and failed invocations performed while the drone is in the communication range of the edge node, including an additional 10-s time window after the drone exits the coverage region of the edge node in order to capture possible application-level communication timeouts that may occur due to this transition. The main comparison refers to the second phase of each experiment (Fig. 14) where the two policies take different decisions. More specifically, the data-driven policy decides that, based on the drone's speed and the previously experienced adaptation delay, it is not worth relocating the ImageChecker to the edge node. In contrast, the naive policy always relocates to the edge. As a result, the naive policy has an invocation failure ratio equal to 33% (18 successful and 9 failed invocations) in the regions of interest. The data-driven policy results in significantly improved application performance, with 33 successful invocations, a failure invocation ratio of 0% (it does not lead to any failed invocations), and 45% more successful invocations to the ImageChecker compared with the naive policy. During the first as well as the third mission phase, both policies decide to relocate the ImageChecker from the cloud to the edge whenever the drone enters the range of the edge node, and vice versa when the drone exits the coverage region of the edge node. Overall, for all three phases of the mission, the naive policy performs 28 total relocations and has an invocation failure ratio of 14%, whereas the data-driven policy decides to perform 8 relocations and has an invocation failure ratio of just 3.7%.

## 8. Scalability of Fluidity mechanisms

In this section, we evaluate the ability of Fluidity to execute the adaptation patterns identified in Section 5 in a parallel way to minimize the respective aggregate delay. We also evaluate the scalability of the telemetry. The results show that the core Fluidity mechanisms are scalable to applications with a large number of components.

### 8.1. Relocating a pair of interacting components

As an extension of the evaluation presented in the previous section, we report the performance of Fluidity for the most complex adaptation pattern, which includes the relocation of both service-providing and client components with application-traffic redirection. We use the same application as before, but investigate the case where the MobileViewer and ImageChecker components are both relocated to different nodes of the continuum. The testbed setup contains the control plane, the cloud (worker) node and the mobile node (RPi) already described in Section 7.1. As edge nodes we use a desktop with a quad-core Intel Core i5-4670 CPU (@3.40 GHz) and 8 GB of memory, and a laptop with an Intel Core i7-1255U CPU (@1.7 GHz) hosting a VM with 2 cores and 5 GB of memory. The laptop also has a WiFi interface which can be used for direct ad-hoc communication with the mobile node. In this experiment, we switch between the following configurations: (i) The MobileViewer runs on the desktop and the ImageChecker runs on the cloud with application traffic going via Ethernet. (ii) The MobileViewer runs on the RPi and the ImageChecker runs on the laptop with application traffic going over ad-hoc WiFi. The transition from the first to the second configuration and backwards is repeated 20 times and we report the average results.

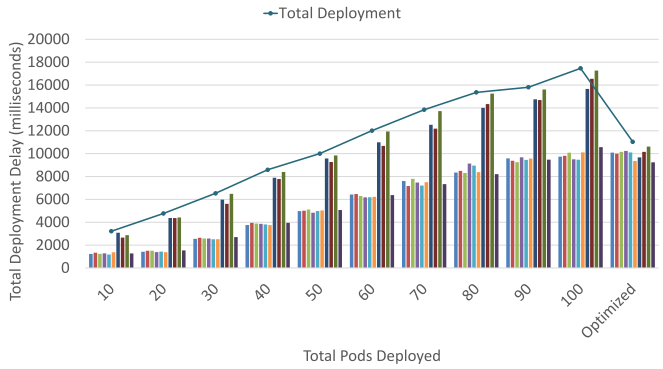



**Fig. 16.** Parallel execution of a scale-out component pattern to 10 nodes/VMs for different values of total pods. For a given total number of pods, each bar corresponds to one of the VMs (1 to 10 from left to right) presented in Table 4.

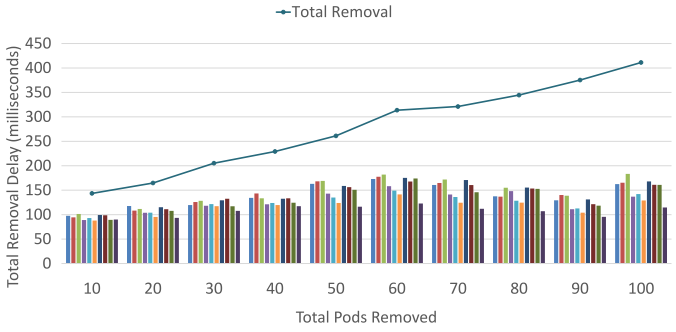



**Fig. 17.** Parallel execution of scale-down component pattern to 10 nodes/VMs for different numbers of total pods. For a given total number of pods, each bar corresponds to one of the VMs (1 to 10 from left to right) presented in Table 4.

The delay of initiating the WiFi connection/disconnection (including the communication delay between the control plane and the mobile node, as well as the actual connection/disconnection delay) is approximately 700 ms. Regarding the individual delays measured on the mobile node, the connection and disconnection delays are equal to 148 and 538 ms respectively. We should note that in the previous experiments (Section 7.4), we reported the actual connection/disconnection delay without the communication overhead. The relocation delay for the service-providing component (ImageChecker) and client component (MobileViewer) is 2.2 and 3.6 s, respectively. Note that the latter delay is higher due to the lower performance of the node involved in the relocation of the ImageChecker component (RPi). The three previous actions are initiated in parallel and fully overlap; therefore, the total overhead equals the maximum delay of these actions (3.6 s). The traffic redirection, which must be performed after the completion of the previous actions, introduces an additional overhead of 3.6 s, thus resulting in a total adaptation overhead of 7.2 s.

### 8.2. Component deployment/removal

In the next set of experiments, we evaluate scale-out and scale-down patterns for a given component. We perform measurements for up to 100 instances as we consider this number reasonable for a

single application. Table 4 summarizes the infrastructure setup for the experiment. The control plane runs in $VM_0$, while $VM_1$–$VM_{10}$ are used as worker nodes that can host application components. The physical machines used for these experiments have sufficient resources to host the aforementioned VMs, and are interconnected via a high-speed 1 Gbps LAN.

Fig. 16 depicts the total overhead for the scale-out experiments. Each column corresponds to the overhead on each node, while the solid line marks the total/aggregate overhead of the respective parallel execution by Fluidity. Pods are equidistributed to the nodes. For example, in the case of 30 total pods, we deploy 3 pods to each node. In addition, the right-most column group shows the per node and total delay for the scale-out to 100 pods provided we perform a more efficient (rather than equi-) pod distribution, based on the speed/performance of each node so as to minimize the total delay. As expected, the total delay is determined by the deployment task/job with the highest completion time. The latter is affected by the node's Kubelet performance for deploying a given number of pods. For example, we can observe that nodes 7 to 9 are consistently slower due to worse disk performance compared to the rest nodes. The control plane, on the other hand, is not the bottleneck; it can distribute and execute overlapping deployment tasks efficiently while having a maximum CPU utilization of 18%.

The results confirm the scalable implementation of the Fluidity deployment mechanism. More specifically, in the case where 100 pods are deployed (10 to each node), the total delay is just 17.4 s. This yields a speedup equal to 6.8x compared to the sequential deployment of 100 pods to the respective nodes which would require at least 118.7 s (based on the individual deployment delays of 10 pods per node). Notably, if we optimize the placement of 100 pods across the 10 nodes, the delay drops to 11 s, corresponding to a speedup of 10.76x compared with the sequential deployment. Similarly, Fig. 17 shows the delay for the scenario where we scale-down a component by removing a given number of instances from each node. For the removal of 100 pods (10 from each node), the total delay is about 410 ms. For a given node, the removal delay for different numbers of pods varies between 100 and 200 ms. The total overhead (blue line) is higher since the actions running in parallel are quite lightweight and do not start perfectly aligned in time on different nodes. The speedup of the parallel scale-down for 100 pods is equal to 3.6x compared to the serialized removal which would have a total delay of approximately 1.5 s.

### 8.3. Telemetry scalability and network requirements

To evaluate whether telemetry can become a bottleneck, we again use the setup reported in Table 4. On each worker node, we run 10 identical application-level telemetry metric producers (application components), where each producer sends telemetry messages, each message carrying a single metric with an average message of 132 bytes, every 1 s to the OpenTelemetry collector running in agent mode on the respective node. All collectors are configured to run with the default settings (including optimizations such as batching) and retry mechanisms, but without having any sending queues. A telemetry-consuming endpoint on the side of the control plane (operating as an HTTP server) receives metrics pushed from the OpenTelemetry gateway instance.

Our results show that we can efficiently support this scenario (transmitting 100 metrics per second in total) without affecting the overall performance of the system. More specifically, the average CPU utilization of the OpenTelemetry collector on all worker nodes was below 1%, while the OpenTelemetry gateway running on the control plane used 3.3% of CPU on average. The maximum recorded memory consumption of the OpenTelemetry collector was equal to 80 MB for the control plane and 62 MB for the worker nodes. The control plane had a reception rate equal to the total production rate of the application components, and the average recorded latency of a single metric transmission was lower than 6 ms for all the nodes without experiencing any loss.

We also performed an additional set of experiments to evaluate the performance of telemetry transmission on an edge device with limited communication throughput over a mobile wireless link, in situations where the link is (i) underutilized and (ii) highly utilized by the application. To this end, we employ the RPi node of our lab testbed described in Section 7.1. The node sends telemetry traffic via its 4G interface and hosts application components with a metrics production rate of 1 Hz as in the above experiments. The effective bandwidth of the 4G link was measured at 2 Mbps. This is very low compared to other results reported in the literature (e.g., up to 50 Mbps in [33]), mainly due to poor 4G connectivity inside our lab. However, this is a good robustness test for our evaluation.

For the first scenario, we isolated and measured the actual traffic that is transferred in the telemetry system, by monitoring the network traffic at the receiving port of the OpenTelemetry collector in gateway mode on the control plane. For a single application component, the average network traffic sent from the mobile node is 5.5 Kbps, and for 10 concurrently executing components is 39 Kbps. Even for our poor 2 Mbps wireless link, the bandwidth requirements for the telemetry system are negligible, less than 0.02% of the available bandwidth thus not inducing a significant overhead to the system. In the second scenario, to stress the wireless link, we let the application produce traffic that is equal to the 75% of the available bandwidth and evaluate the telemetry transmission of 1 application component. Again, we confirm that the telemetry flow can be handled without any issues. The average latency of telemetry is about 130 ms and 250 ms for the underutilized and the stressed link scenario, respectively. Over both experiments, the maximum latency is approximately 1 s, experienced in less than 1% of the message transmissions. Thus, the telemetry mechanism has an acceptable performance even for nodes operating at the edge. Notably, the results would be significantly better for a 5G link (we plan to acquire such a dongle in the near future).

## 9. Versatility of Fluidity

Fluidity is agnostic to the underlying network technology and can support any network implementing IP. Different network implementations represent different points in the latency, throughput, resilience, availability, and cost solutions search space. Thanks to its design, Fluidity can accommodate diverse plugin policies, which may opt for different network implementations under different conditions and optimization targets. It is also possible to extend the system infrastructure descriptions of Fluidity to capture (static) features of the underlying network technologies, which can serve as additional hints for the policies in order to take application traffic redirection decisions.

Fluidity has been meticulously designed and implemented to offer its functionality without requiring any modifications to Kubernetes itself. As a result, beyond being deployed on private nodes, it can also be deployed on public clouds offering either Infrastructure-as-a-Service (IaaS) or – the relatively new popular offering – Container-as-a-Service (CaaS) solutions. The three major public cloud providers have introduced such CaaS offerings in their portfolios: Amazon Elastic Kubernetes Service (EKS), Azure Kubernetes Service (AKS), and the Google Kubernetes Engine (GKE).

Major public cloud providers have also lately focused on Edge-enabling their cloud offerings. Taking Amazon as an example, AWS Wavelength embeds AWS compute and storage devices in the infrastructure of 5G networks. Coupled with the concept of wavelength zones and edge local zones, it provides the infrastructure to develop, deploy and scale ultra-low-latency applications. AWS EKS can be extended to AWS Wavelength compute resources [34]. Finally, Amazon provides AWS Greengrass [35], an open-source edge runtime and a corresponding cloud service that enables standalone edge devices to participate in the AWS ecosystem, allowing them to communicate with application components executing on Edge or Cloud resources. Given that Fluidity relies on unmodified Kubernetes, it can exploit such hybrid

public/private resources at the Cloud, Edge, and Standalone Edge, provided that the management layer allows control on the placement of pods (which is the case in AWS offerings). Moreover, in order to implement transparent traffic redirection to direct wireless links at last-mile, Fluidity needs to have administrator access to nodes accommodating the endpoints of the respective link. This is typically the case for individual edge and far-edge nodes (they are owned by the application owner - that is why cloud providers offer middleware, such as AWS Greengrass to allow such nodes to seamlessly interact with their ecosystem).

## 10. Conclusions & future work

We have introduced Fluidity, a framework for the automated and adaptive deployment of serverless and modular applications in systems with cloud, edge, and mobile nodes. Apart from its architecture and mechanisms, we discussed the aspects pertaining to the description of the system infrastructure and the applications to be registered to Fluidity. We also introduced APIs to facilitate communication of application-level metrics to the system and to enable custom, plug-in adaptation policies, which can be changed at execution time. This flexibility is necessary for next-generation compute systems with complex serverless applications operating on multilayer, heterogeneous infrastructures, with varying characteristics.

Our evaluation shows that the core adaptive deployment and dynamic policy switching mechanisms of Fluidity have an acceptable overhead, enabling support for a wide class of applications that do not have tight real-time requirements. The application- and system-level performance can be significantly improved by developing adaptation policies tailored to the unique, potentially time-varying characteristics of each application and system. Our results further confirm that Fluidity can execute adaptation plans produced by a policy in a scalable way. As future work, we plan to support hierarchical policies where Fluidity agents will accommodate policies based on their respective layers. Focusing on policies, we plan to exploit Fluidity as the platform for designing and evaluating ML-based policies (e.g., to predict trajectories of mobile nodes and proactively reconfigure the system).

## CRediT authorship contribution statement

**Foivos Pournaropoulos:** Conceptualization, Investigation, Methodology, Software, Visualization, Writing – original draft, Writing – review & editing. **Alexandros Patras:** Investigation, Methodology, Software, Visualization, Writing – original draft, Writing – review & editing. **Christos D. Antonopoulos:** Funding acquisition, Methodology, Project administration, Supervision, Writing – review & editing. **Nikos Bellas:** Funding acquisition, Methodology, Project administration, Supervision, Writing – review & editing. **Spyros Lalis:** Funding acquisition, Methodology, Project administration, Supervision, Writing – review & editing.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

No data was used for the research described in the article.

## Acknowledgments

This work has received funding from the Horizon Europe research and innovation programme of the European Union, under grant agreement no 101092912, project MLSysOps.

## References

[1] Datadog, State of Serverless Report. https://www.datadoghq.com/state-of-serverless/, (Accessed 2023-09-08).
[2] Kubernetes, https://kubernetes.io/, (Accessed 2023-09-08).
[3] M. Satyanarayanan, P. Bahl, R. Caceres, N. Davies, The Case for VM-Based Cloudlets in Mobile Computing, IEEE Pervasive Comput. 8 (4) (2009) 14–23.
[4] P. Liu, D. Willis, S. Banerjee, Paradrop: Enabling lightweight multi-tenancy at the network's extreme edge, in: IEEE/ACM Symposium on Edge Computing, 2016, pp. 1–13.
[5] K. Bhardwaj, M.-W. Shih, P. Agarwal, A. Gavrilovska, T. Kim, K. Schwan, Fast, scalable and secure onloading of edge functions using AirBox, in: IEEE/ACM Symposium on Edge Computing, 2016, pp. 14–27.
[6] K. Kritikos, P. Skrzypek, Towards an optimized, cloud-agnostic deployment of hybrid applications, in: International Conference on Business Information Systems, 2019, pp. 435–449.
[7] K. Kritikos, C. Zeginis, E. Politaki, D. Plexousakis, Towards the modelling of adaptation rules and histories for multi-cloud applications, in: International Conference on Cloud Computing and Services Science, 2019, pp. 300–307.
[8] J. Carrasco, J. Cubo, E. Pimentel, Towards a flexible deployment of multi-cloud applications based on TOSCA and CAMP, in: European Conference on Service-Oriented and Cloud Computing, 2014, pp. 278–286.
[9] T. Tsourdinis, N. Makris, S. Fdida, T. Korakis, DRL-based Service Migration for MEC Cloud-Native 5G and beyond Networks, in: 2023 IEEE 9th International Conference on Network Softwarization (NetSoft), 2023, pp. 62–70.
[10] A. Garbugli, A. Sabbioni, A. Corradi, P. Bellavista, TEMPOS: QoS management middleware for edge cloud computing FaaS in the Internet of Things, IEEE Access 10 (2022) 49114–49127.
[11] T. Kasidakis, G. Polychronis, M. Koutsoubelias, S. Lalis, Reducing the mission time of drone applications through location-aware edge computing, in: IEEE International Conference on Fog and Edge Computing, ICFEC, 2021, pp. 45–52.
[12] J. Yapp, R. Seker, R. Babiceanu, UAV as a service: Enabling on-demand access and on-the-fly re-tasking of multi-tenant UAVs using cloud services, in: IEEE/AIAA Digital Avionics Systems Conference, 2016.
[13] A. Koubaa, B. Qureshi, M.-F. Sriti, A. Allouch, Y. Javed, M. Alajlan, O. Cheikhrouhou, M. Khalgui, E. Tovar, Dronemap Planner: A service-oriented cloud-based management system for the Internet-of-Drones, Ad Hoc Netw. 86 (2019) 46–62.
[14] A. Van't Hof, J. Nieh, AnDrone: Virtual drone computing in the cloud, in: Eurosys, 2019, pp. 6:1–6:16.
[15] S. He, F. Bastani, A. Balasingam, K. Gopalakrishnan, Z. Jiang, M. Alizadeh, H. Balakrishnan, M. Cafarella, T. Kraska, S. Madden, Beecluster: drone orchestration via predictive optimization, in: International Conference on Mobile Systems, Applications and Services, 2020, pp. 299–311.
[16] N. Grigoropoulos, S. Lalis, Fractus: Orchestration of Distributed Applications in the Drone-Edge-Cloud Continuum, in: IEEE 46th Annual Computers, Software, and Applications Conference, COMPSAC, 2022, pp. 838–848.
[17] H. Zhang, N. Liu, X. Chu, K. Long, A. Aghvami, V. Leung, Network slicing based 5G and future mobile networks: Mobility, resource management, and challenges, IEEE Commun. Mag. 55 (8) (2017) 138–145.
[18] Istio service mesh, https://istio.io/, (Accessed 2023-09-08).
[19] KubeEdge, https://kubeedge.io/, (Accessed 2023-09-08).
[20] A. Ferreira, E.V. Hensbergen, C. Adeniyi-Jones, E. Grimely-Evans, J. Minor, M. Nutter, L.E. Pena, K. Agarwal, J. Hermes, SMARTER: Experiences with cloud native on the edge, in: USENIX Workshop on Hot Topics in Edge Computing (HotEdge), 2020.
[21] Flannel, https://github.com/flannel-io/flannel, (Accessed 2023-09-08).
[22] F. Pournaropoulos, C.D. Antonopoulos, S. Lalis, Supporting the Adaptive Deployment of Modular Applications in Cloud-Edge-Mobile Systems, in: International Conference on Embedded Wireless Systems and Networks (EWSN), 2023.
[23] K3s, https://k3s.io/, (Accessed 2023-09-08).
[24] Containerd, https://containerd.io/, (Accessed 2023-09-08).
[25] OpenTelemetry, https://opentelemetry.io, (Accessed 2023-09-08).
[26] OpenTelemetry Collector, https://github.com/open-telemetry/opentelemetry-collector, (Accessed 2023-09-08).
[27] Prometheus, https://prometheus.io, (Accessed 2023-09-08).
[28] N. Sabharwal, P. Pandey, Working with Prometheus Query Language (PromQL), A Press, Berkeley, CA, 2020, pp. 141–167.
[29] Ardupilot SITL, http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html, (Accessed 2023-09-08).
[30] MAVLink, https://mavlink.io/en/, (Accessed 2023-09-08).
[31] MAVProxy, https://ardupilot.org/mavproxy/, (Accessed 2023-09-08).
[32] DroneKit, http://dronekit.io/, (Accessed 2023-09-08).
[33] L. Mei, J. Gou, Y. Cai, H. Cao, Y. Liu, Real-time mobile bandwidth and handoff predictions in 4G/5G networks, 2021, ArXiv, abs/2104.12959.

[34] Managing edge-aware Service Mesh with Amazon EKS for AWS Local Zones. https://aws.amazon.com/blogs/containers/managing-edge-aware-service-mesh-with-amazon-eks-for-aws-local-zones/, (Accessed 2023-09-08).
[35] AWS Greengrass, https://docs.aws.amazon.com/greengrass/latest/developerguide/greengrass-v2-developer-guide.pdf, (Accessed 2023-09-08).

**Foivos Pournaropoulos** is a Ph.D. candidate at the ECE Department of the University of Thessaly. He received the diploma in Electrical and Computer Engineering from the University of Thessaly in 2022. His research interests include adaptive deployment, edge computing, high performance computing, heterogeneous systems, system-level programming and virtualization.

**Alexandros Patras** is a Ph.D. candidate at the ECE Department of the University of Thessaly in Volos, Greece. His research interests include machine learning, system optimization, cloud computing and embedded systems. He is also interested in distributed systems and web technologies.

**Christos D. Antonopoulos** is Professor at the ECE Department of the University of Thessaly and a Research Associate at IRETETH-CERTH in Volos, Greece. His research interests span the areas of system and applications software for high performance computing, emphasizing on run-time monitoring and adaptivity with performance and power criteria. He also works on improving the programmability of accelerator-based heterogeneous systems, as well as on techniques for automatic, application-driven redefinition of the hardware/software boundary on accelerator-based systems. He earned his Ph.D.,M.Sc. and Diploma from the Computer Engineering and Informatics Department of the University of Patras, Greece. He also served as a postdoctoral research associate at the Computer Science Department of the College of William and Mary, VA, USA. He has published over 60 papers in international conferences and journals.

**Nikos Bellas** is Professor at the ECE Department of the University of Thessaly in Volos, Greece. He received the diploma in computer engineering and informatics from the University of Patras,Greece, in 1992, the M.Sc. and Ph.D. degrees in electrical and computer engineering from the University of Illinois, Urbana-Champaign, in 1995 and 1998, respectively. His research interests include approximate computing, reconfigurable computing, low-power design, CAD tools for architectural synthesis . From 1999 to 2007, he was a principal member of Technical Staff at Motorola Labs in the US. He was one of the architects of Falcon, an MPEG4 video decoding chip used by the first Motorola camera phone, and has worked extensively on architecting systems on chip for multimedia and imaging applications. He holds 10 issued US patents.

**Spyros Lalis** is Professor at the ECE Department, University of Thessaly, Greece. He received a Diploma in Computer Science and a Ph.D. in Technical Sciences from ETHZ, and has worked at the Computer Science Department, University of Crete, the Foundation for Research and Technology Hellas (FORTH) and the Center for Research and Technology Hellas (CERTH). His research interests are mainly in programming models, operating systems, distributed systems and ubiquitous computing. He has actively contributed in the development of system software and middleware for market-based resource allocation, distributed computing and metacomputing, ad-hoc wearable computing, wireless sensor/actuator networks, mobile/crowd sensing and drone-based systems. He has published over 100 scientific papers in international journals and conferences, and has received significant funding for his work through competitive EU and national research projects.