Supporting the Adaptive Deployment of Modular Applications in Cloud-Edge-Mobile Systems

Foivos Pournaropoulos, Christos D. Antonopoulos, Spyros Lalis
Department of Electrical and Computer Engineering
University of Thessaly
{spournar, cda, lalis}@uth.gr

Abstract

We introduce Fluidity, a framework enabling the flexible and adaptive deployment of modular applications in systems comprising cloud, edge, and mobile IoT nodes. Based on a declarative description of application requirements, Fluidity plans and executes an initial deployment of application components in the cloud-edge-mobile continuum. At runtime, Fluidity monitors resource availability and the position of mobile nodes, and adapts the deployment of the application accordingly, without any intervention from the application owner or system administrator. Notably, Fluidity allows applications to provide their own deployment and adaptation policies and to switch between different policies at runtime, while the application is running. We discuss the design and implementation of Fluidity in detail and provide an evaluation using a lab testbed, where the mobile node is a simulated drone. Our results show that the core mechanisms of Fluidity can adapt the application at reasonable overhead.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: *Reliability, availability and serviceability*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms

Design, Experimentation, Measurement, Performance

Keywords

Microservices, Edge Computing, Mobile Computing, Flexible Deployment, Runtime Adaptation, IoT, Drones

1 Introduction

An increasing number of applications are no longer restricted to the cloud but also involve mobile IoT devices, such as smartphones, and vehicles like cars or even autonomous drones. In this case, edge computing can improve the application's quality of service (QoS) by moving computations closer to the points where data is produced, while leading to better overall system performance and stability, as it reduces data traffic and resource pressure to the cloud. However, in order to effectively harness the available edge resources, one needs to support flexible and adaptive deployment and orchestration, taking into account the changing position of the mobile nodes.

In this work, we introduce Fluidity, a framework that enables adaptive application deployment and orchestration across the continuum. Our mechanism is built as an extension of Kubernetes [17], which is used as the underlying mechanism for the basic pod/container deployment and monitoring. Going beyond the standard functionality of Kubernetes, Fluidity supports adaptive node selection and application deployment for mobile and edge computing, with transparent redirection of application traffic over ad-hoc WiFi links, driven by application-specific add-on policies.

The main contributions of this paper are: (i) We introduce a framework for the flexible and adaptive deployment and orchestration of applications in the cloud-edge continuum, with a focus on mobile IoT nodes. (ii) Our design separates the core (re)deployment mechanism from the logic that takes the placement decisions, making it possible for the application to provide different policies using a structured interface. (iii) We quantify the policy-independent overheads of the core framework mechanisms for the actual adaptation of application deployment, application traffic redirection, and policy change, showing that they are sufficiently small to support applications that do not have strict real-time requirements.

The rest of the paper is structured as follows. Section 2 introduces a motivating application example. Section 3 presents the design and implementation of the Fluidity framework, while Section 4 provides the evaluation. Section 5 discusses related work. Finally, Section 6 concludes the paper.

2 Application running-example

We use a simple application example throughout the paper to motivate the problem and illustrate the functionality of the Fluidity framework. Note that Fluidity is application-independent and can support the adaptive deployment of complex applications with different characteristics.

The application consists of three components. The

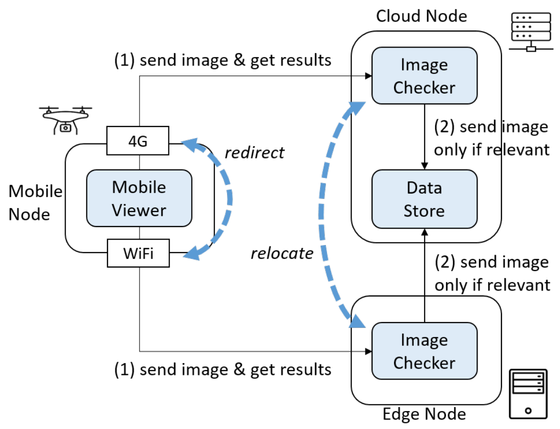
Cloud Node
(1) send image & get results
Image Checker
(2) send image only if relevant
Data Store
4G
redirect
Mobile Node
Mobile Viewer
WiFi
relocate
(2) send image only if relevant
Image Checker
(1) send image & get results
Edge Node

Figure 1. Placement of application components and dynamic relocation of the ImageChecker between cloud and edge with traffic redirection between 4G and WiFi.

MobileViewer takes pictures and sends them to the *ImageChecker*, which processes them to detect objects of interest. If objects are detected, the ImageChecker sends the picture to the *DataStore* for long-term storage and possibly additional processing. In essence, the ImageChecker and DataStore are services that are invoked by the MobileViewer and ImageChecker, respectively.

We wish to deploy and run the application on top of a distributed system spanning the cloud-edge-mobile continuum, which includes a mobile node equipped with a camera (e.g., a drone), an edge node offering computing resources to the application, and a node in the cloud. The MobileViewer needs to run on the mobile node to access the camera, and the DataStore runs in the cloud to have access to ample storage space. However, the ImageChecker can run either on the edge node or in the cloud. For the sake of the example, we assume that it is desirable to dynamically relocate the ImageChecker between the cloud and the edge, depending on the location of the node hosting the MobileViewer component, with the application-level traffic between the interacting components being redirected accordingly, as illustrated in Figure 1. Running the ImageChecker at the edge, close to the mobile node hosting the MobileViewer, not only reduces the pressure on mobile communication and cloud resources but can also improve the interaction between these components.

3 Design and Implementation

The Fluidity framework is designed to support such a flexible and adaptive deployment of applications in the continuum. We start by briefly describing the application model and software architecture of Fluidity. Then, we discuss the internal operation of the framework in more detail.

3.1 Application model

We consider modular applications consisting of distinct components that can be separately deployed on different nodes of the system. Each component is provided by the application owner in the form of a Docker container [4]. Application deployment is driven through a structured description (in YAML format), listing the application components, their desired placement in the continuum, their resource requirements, and the interactions between them. In particular, to guide application placement in the continuum, each component can be targeted for the cloud, edge, or mobile nodes, via explicit labels in the application description. A component can also be labeled as hybrid, indicating that it may be placed in the cloud or at the edge.

Fluidity supports a flexible deployment for both edge and hybrid components: it initially creates an instance on a suitable host and can subsequently relocate it on another host based on the current location of mobile nodes hosting other application components. Moreover, Fluidity supports the dynamic change of the policy used to plan and adapt application deployment. The current implementation supports the relocation of stateless application components/services, by restarting the container on another host without any special state transfer (there is no live container migration).

3.2 Software architecture

Figure 2 depicts the software architecture of the Fluidity framework for an indicative cluster including a mobile IoT node (drone), an edge node, and a cloud node for hosting application components. The control plane runs in a separate node, also in the cloud. All nodes are connected through a VPN. The network connection of the stationary nodes is via an Ethernet interface, while the mobile node uses a 4G link. This system configuration is intentionally simple for the sake of the example. In the general case, the system may include several mobile, edge, and cloud nodes.

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. It also monitors the state of the system and, if needed, adapts the current deployment. The actual deployment of application components on the selected hosts (as pods) is done via K3S [11], while all monitoring information is received via the Kubernetes API. However, the adaptive selection of the target hosts and network connections as a function of the global system status is done by the Fluidity Controller.

Every 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. If the node is mobile, the Agent also periodically sends to Kubernetes its current position. The Agent runs on the side of the Kubelet that takes care of the local deployment of the pods containing the application components. Mobile and edge nodes with a WiFi interface also run the Net-Proxy. This implements the redirection of application traffic from the default data path (via 4G and the Internet) over a WiFi link between these nodes.

3.3 Operation of the Controller

As shown in Figure 3, 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 description and current deployment. The Scheduler receives from Kubernetes application deployment request events, initializes and updates its inter-

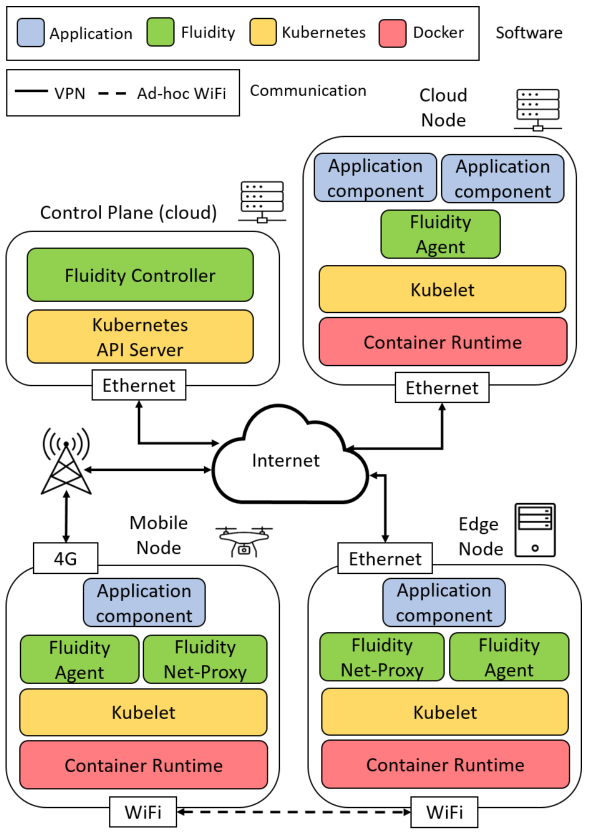

Figure 2. Fluidity architecture and cluster configuration.

nal data structures, selects suitable hosts, and then deploys or removes application components via Kubernetes. The Monitor periodically queries Kubernetes to get the status of the deployed pods and available system resources. If significant changes occur, it alerts the Scheduler by posting events in the notification queue, to decide how to adapt the deployment.

The policy for adapting the deployment is abstracted via the *analyze()* and *plan()* functions. The former is invoked from within the Monitor to analyze the status of pods and system resources and decide whether to generate a notification for the Scheduler. The latter is invoked from within the Scheduler to process application deployment requests as well as the notifications generated by the Monitor, in order to produce an updated component-to-node mapping.

Notably, the above policy functions are not hardwired in the Fluidity framework. They are provided as part of the application description, which can be changed at runtime by the user or the application itself. Thus, each application can come with its own deployment and adaptation policies, and dynamically switch between them during execution without having to stop/restart or suspend/resume.

3.4 Initial deployment

The procedure for initiating application deployment is, briefly, as follows. The user (administrator) submits an application deployment request to Kubernetes, which notifies the Scheduler thread of the Controller. The Scheduler retrieves/parses the application description and initializes some

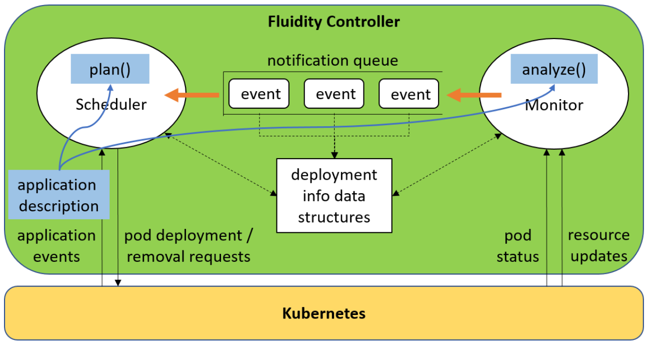

Figure 3. Internal structure of the Fluidity Controller.

internal data structures as well as the policy-specific functions. Then, it retrieves from Kubernetes the current state of the cluster, updates the corresponding data structures, and invokes the *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, and starts the Monitor thread.

For edge or hybrid components, the respective container is proactively sent to every edge node that can potentially act as a host for it, to accelerate placement adaptations that may be decided in the future. However, only one live instance of the edge / hybrid component is actually created in the system, on the host that is selected by the policy.

3.5 Adaptation of deployment

Fluidity can support a wide range of deployment adaptation scenarios to handle node mobility, the addition and removal of nodes in/from the cluster, and the failure of a pod running an application component. Moreover, the user or the application can modify the deployment requirements and policy. To illustrate how Fluidity works, we focus on the application described in Section 2 and discuss the adaptive deployment of the hybrid component (ImageChecker).

Figure 4 shows the basic steps of the adaptation process, for the case where the hybrid component is relocated from the cloud to the edge. The Agent of the mobile node periodically sends its location to Kubernetes. This is captured by the Monitor, which updates the Controller data structures and invokes the *analyze()* function to decide whether to notify the Scheduler. In this case, the Scheduler invokes the *plan()* function to produce an updated component-to-node mapping (if any). If it is indeed decided to change the current deployment, the Scheduler generates a new pod file for the hybrid component and target node, and implements the adaptation, by deploying the new pod on the selected host and then removing the unwanted pod from the old host.

3.6 Redirection of application traffic

Fluidity exploits the ability of mobile nodes to interact directly with edge nodes via wireless communication (such as WiFi), rather than via the default communication path (4G and the Internet). To this end, additional actions are performed when the hybrid component relocates (i) from the cloud to an edge node, (ii) from an edge node to the cloud, or (iii) between two edge nodes.

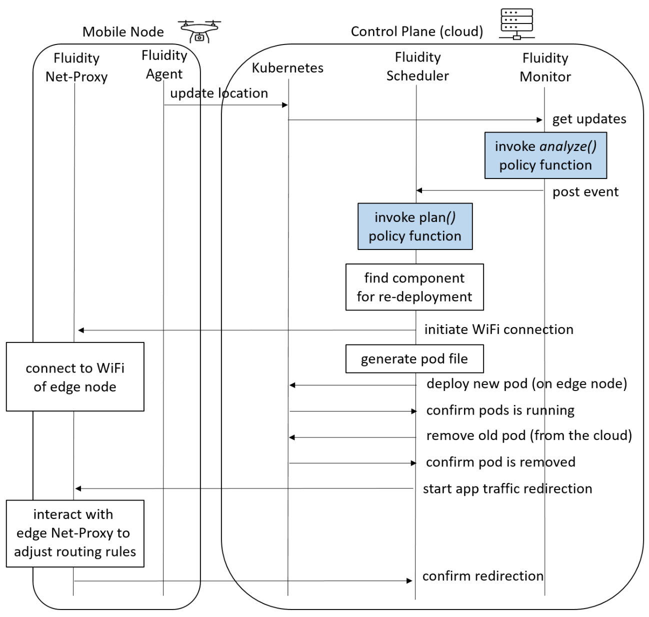
Mobile Node
Control Plane (cloud)
Fluidity Net-Proxy
Fluidity Agent
Kubernetes
Fluidity Scheduler
Fluidity Monitor
update location
get updates
invoke *analyze()* policy function
post event
invoke plan*()* policy function
find component for re-deployment
initiate WiFi connection
connect to WiFi of edge node
generate pod file
deploy new pod (on edge node)
confirm pods is running
remove old pod (from the cloud)
confirm pod is removed
start app traffic redirection
interact with edge Net-Proxy to adjust routing rules
confirm redirection

Figure 4. Deployment adaptation by relocating a hybrid component (ImageChecker) from cloud to edge.

In the first case, shown in Figure 4, 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 network SSID and key). This is done before generating and deploying the pod file on the edge node so that the WiFi connection delay overlaps with the component deployment delay. Also, when the old pod is removed, the Scheduler instructs the Net-Proxy to redirect application traffic to the 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 associated to the hybrid component.

Conversely, when the hybrid component relocates from the edge back to the cloud, the Scheduler instructs the Net-Proxy to restore the routing rules and disconnect from the edge WiFi network. As above, this is done after removing the component instance from the previous host. A similar process is followed when the hybrid component is relocated between two edge nodes. However, in this case, the Net-Proxy needs to disconnect from the WiFi of the old edge host, before it can connect to the WiFi of the new edge host.

4 Evaluation

We evaluate Fluidity using a system/cluster configuration in the lab, which includes an embedded mobile node, an edge node and a cloud node. Our experiments focus on the adaptation of application deployment performed by Fluidity and the respective overhead. They are not designed to quantify how the movement of the mobile node affects the performance of its wireless link to the edge node, as this is not related to the mechanisms of the Fluidity framework.

4.1 Test application and cluster configuration

The test application used in our experiments is the one described in Section 2. We let the MobileViewer take a picture and invoke the ImageChecker to process the image, in an endless loop. Thus, the actual invocation rate during application execution depends only on the invocation delay.

The cluster configuration is similar to that of Figure 2. The mobile IoT node is represented by a Raspberry Pi 3 Model B (RPi), with a quad-core ARM Cortex-A53 CPU (@1.2 GHz) with 1GB of memory, which we regularly use as an on-board companion computer on a quadcopter drone. The RPi is connected to the SITL configuration of Ardupilot [1], which simulates the drone dynamics. The edge node is a VM with 2 cores and 5GB of memory, running on a laptop with a dual-core Intel Core i5-7200U CPU (@2.5GHz). Finally, the control plane and cloud node are VMs with 4 cores and 16GB memory, running on physical nodes with Intel Xeon E5-2630 0 (@2.30GHz) and Intel Xeon E5-2620 v2 (@2.10GHz) CPUs in the cluster of our department.

Connectivity with the control plane is via a VPN, over 4G for the drone (RPi) and over Ethernet for the edge node (laptop). This also serves as the default connectivity path for application-level traffic. The wireless link between the RPi and the laptop, activated when the ImageChecker component runs on the latter, is over ad-hoc WiFi.

All nodes hosting application components run the Docker runtime, the Kubelet and the Fluidity Agent. The RPi and laptop also run the Fluidity Net-Proxy. The Agent on the RPi interacts with the autopilot subsystem to retrieve the current position of the drone, via Dronekit [5]. Notably, the RPi runs exactly the same software when mounted on a real drone, the only difference being that the communication with the autopilot system is over serial. In previous tests [7], not related to Fluidity, we have verified that the SITL setup reproduces the behavior of a real quadcopter drone, including the time to take-off, move between waypoints and land.

4.2 Test scenario

Figure 5 illustrates the flight scenario used to evaluate our implementation. More specifically, the (virtual) drone moves between two locations in a straight line at a constant speed, while the edge node is (virtually) placed in the mid-point of this path. The start and end points of the drone's path are set 300 meters apart, while the WiFi range of the drone and edge node is set to a radius of 50 meters.

Each run includes several round-trips, from the start to the end point and back, so in each round-trip the drone enters the range of the edge node twice. Note that this scenario, despite being simple, is equivalent to larger mission scenarios, where the drone operates in a wider region overflying areas close to an edge node followed by areas where the drone is not within the range of any edge node.

The main purpose of the evaluation is to capture the overhead of the core Fluidity mechanism, without focusing on the effectiveness of the adaptation policy. To this end, we employ the simple adaptation policy of placing the instance of the ImageChecker to the edge node whenever the drone hosting the MobileViewer is in its range. More specifically, *analyze()* generates a notification to the Scheduler when the drone enters or exits the area covered by the WiFi of

Table 1. Major components of adaptation overhead.

Name	Description
(1) Resource update	Time needed to update resource status.
(2) Notification decision	Time needed by *notify()* to decide whether to notify the Scheduler.
(3) Notification	Time needed for the Scheduler to receive the notification of the Monitor.
(4) Planning	Time needed by *plan()* to produce the new component-to-node mapping.
(5) Find migration target	Time needed to identify the component(s) to be migrated.
(6) WiFi connection request	Time needed to check if the edge node has WiFi and ask the drone to connect to it (cloud-to-edge migration).
(7) WiFi connection	Time needed for the drone to connect to the WiFi network of the edge node (cloud-to-edge migration).
(8) Pod-file creation	Time needed to build the pod-related files for new component instances.
(9) Pod deployment	Time needed to deploy the new pod & confirm this is running.
(10) Pod removal	Time needed to remove the unwanted pod & confirm this has terminated.
(11) Redirection	Time needed to redirect application traffic (from 4G to WiFi or vice versa, depending on migration direction).
(12) WiFi disconnection	Time needed for the drone to disconnect from the WiFi network of the edge node (edge-to-cloud migration).
(13) Policy switch	Time needed for the Controller to install a new policy.

Table 2. Application metrics.

Name	Description
(1) Invocation delay	The time needed for the MobileViewer to invoke the ImageChecker and get a response.
(2) Invocation failure ratio	The number of failed invocations to the total number of invocations attempted within a 10-second time window at each transition point.

the edge node (specified in the node resource description), while *plan()* produces the adapted deployment for the ImageChecker component.

4.3 Metrics

To quantify the overhead of the framework, we monitor the major delay components, summarized in Table 1. All delays are measured at the Controller, except (7) and (12) which are measured on the drone node (RPi). As shown in Figure 4, (7) overlaps with (8)-(10) as the WiFi connection is performed by the Net-Proxy in parallel to the deployment of the new ImageChecker pod on the edge node and the removal of the old pod from the cloud. Note that there is no entry for a WiFi disconnection request from the Scheduler to the drone's Net-Proxy. As discussed in Section 3.6, there is no separate interaction for this. Instead, when the ImageChecker migrates from the edge back to the cloud, the Net-Proxy disconnects from the WiFi of the edge node once it confirms the application traffic redirection to the Scheduler. Since this is done asynchronously, the WiFi disconnection delay does not affect the actual adaptation time.

We also record the impact of adaptation to the behavior/performance of the application through the metrics listed

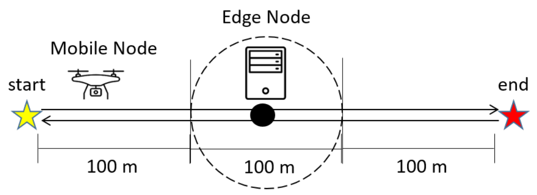

Figure 5. Test scenario, leading to repeated relocations of the ImageChecker between cloud and edge.

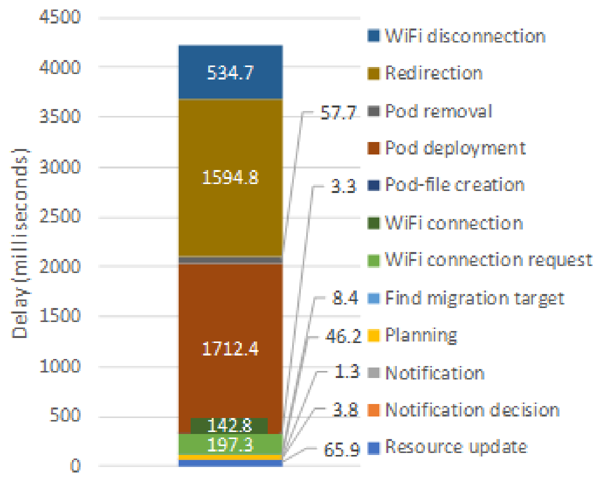

Figure 6. Breakdown of adaptation overhead. Note that the WiFi connection delay fully overlaps with the delay for pod file creation and pod deployment.

in Table 2. Both metrics are recorded at the MobileViewer component. Note that some of the invocations of the MobileViewer to the ImageChecker may fail and this is captured via the second metric. The reason for such failed invocations is that the old instance of the ImageChecker may still be processing a previous invocation when it is removed.

4.4 Results

We measure the adaptation overhead for a total of 60 relocations of the ImageChecker component from the cloud to the edge and back, performed in 15 consecutive round-trips. Figure 6 shows the average recorded delay, broken down to the individual actions that are performed to relocate the hybrid component from the cloud to the edge and vice versa. Note that the WiFi connection delay applies only when the component relocates from the cloud to edge, and fully overlaps with pod file creation and deployment. The WiFi disconnection delay applies for the reverse relocation, from the edge to the cloud, but does not affect the actual adaptation time.

Clearly, the most significant overhead comes from the pod deployment and application traffic redirection, accounting for 46% and 43% of the delay, respectively. The overhead of the (intentionally simple) policy is merely 1.4%. The average adaptation delay, excluding the WiFi disconnection that is performed asynchronously, is roughly 3.7 seconds in total. While quite significant, this is still acceptable for applications that do not have tight real-time requirements.

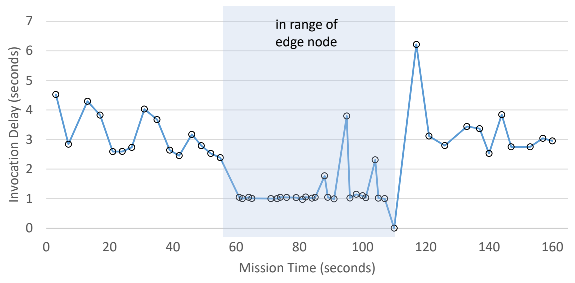

Figure 7. Invocation delay during an indicative pass. The zero value represents a failed invocation.

We have also measured the time needed to change the policy, while the application is running. The average delay is about 25 milliseconds over 50 policy switches. This allows the application to switch policies frequently, e.g., to adopt different strategies at various phases of execution/locations of the mobile node, or to continually improve its strategy through machine learning methods.

Figure 7 plots the invocation delay between the MobileViewer and the ImageChecker during an indicative trip from the start to the end point. This data is collected from the respective Pod logs after the test mission completes. The shaded area marks the time window when the drone is in range of the edge node. Thanks to the adaptation by Fluidity, the ImageChecker relocates to the edge node, leading to a sharp drop of the invocation delay. Fluctuations of invocation delay during each phase of the experiment (ImageChecker in cloud (0-55], edge (55-110], cloud (110-160] respectively) are due to the dynamically varying characteristics of the 4G and ad-hoc WiFi networks. Over all experiments including more than 500 invocations, the average edge invocation delay is 1.89 seconds vs 2.84 seconds for the cloud, an improvement of 33%. Notably, the actual computation takes about 0.54 seconds on the laptop vs 0.30 in the cloud VM (almost $2x$ faster), but the shorter data transfer time over WiFi vs 4G makes the difference in favor of the edge. Of course, edge computing would be even more beneficial with a more powerful edge node.

The zero value in Figure 7 (time point 110) denotes a failed invocation. Such failures may occur at the transition points where the relocation of the ImageChecker takes place. The invocation failure ratio at these points (within a 10-second window around the transition points) is 14.4%. Note, however, that such failures are not fatal. Given that ImageChecker embodies a stateless service, the application can recover from a failed invocation simply by repeating the attempt.

5 Related Work

5.1 Offloading and flexible deployment

The work in [19] introduces the concept of service offloading from mobile devices to powerful computers operating at the edge, using VM virtualization to host the respective services. Offloading occurs whenever the mobile node is in the proximity range of edge nodes. [18] 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. Further, [2] introduces trusted environments for container operation. However, the authors in [19, 2] focus on automating the runtime synthesis of the execution enclaves to implement services, rather than the run-time management of prebuilt application components in the form of containers. With respect to [18, 2], apart from supporting the initial application deployment, our main focus is on adapting it at runtime. Ad-hoc computation offloading from a drone to edge servers is studied in [12]. However, it assumed that all servers already have the required service pre-installed.

Some works, which explicitly target drone orchestration and management, place all application logic in the cloud [20, 13]. Others enable the native execution of container-based drone applications [9, 8]. Our work extends these efforts, aiming at the adaptive end-to-end deployment for applications spanning the entire edge-cloud continuum, rather than dealing only with the part that resides on the drone.

In [14], 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 [15] supports multi-cloud applications consisting of components with various service levels. Deployment readjustment is supported by complex rearrangement rules that are activated by events and are converted to workflows, which can be modified in the course of time to optimize the used plan. [3] 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, we focus on cloud/edge applications (rather than multi-cloud or hybrid cloud/serverless), including mobile nodes such as drones, and we introduce deployment adaptation due to mobility.

In previous work [7], we explored flexible application deployment with special focus on hiring drones on-demand and allowing the application to control the drone's path through suitable directives. While there is a similar concept of hybrid components, application deployment is static and a greedy approach is adopted by creating live instances on every edge node that is near the (estimated) path of the drone, also keeping an instance in the cloud as a fall-back option. Fluidity supports a more adaptive and resource-efficient deployment of hybrid components by creating a single instance that is dynamically relocated on different hosts based on the position of the mobile node. Moreover, Fluidity separates the core deployment mechanisms from the policy that makes deployment and adaptation decisions, which is provided as part of the application description and can be changed at runtime. This allows policy developers to experiment with different policies, providing a significantly larger degree of extensibility than the previous framework.

5.2 Network management

The 5G architecture adopts a service-oriented view of the network to satisfy different and possibly contrasting requirements of a variety of applications [21]. While 5G can benefit mobile and drone-based applications, 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 exploit different wireless networking technologies already supported by edge nodes in a transparent way for the application.

In Kubernetes, it is common to connect application components through service meshes, like Istio [10], which 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 other hand, Fluidity supports traffic redirection over ad-hoc wireless network interfaces, allowing applications to benefit from the physical proximity of edge computing resources while enjoying higher bandwidth and lower latency vs the default path of mobile connectivity.

Recently, edge-oriented Kubernetes derivatives [16, 6] allow the application to take advantage of various networking technologies. These approaches are useful, yet they shift the responsibility of connectivity management to the application developer. In contrast, Fluidity can transparently exploit different ad-hoc networking capabilities under the hood.

6 Conclusion

We have presented Fluidity, a framework for the adaptive deployment of modular applications in systems that include cloud, edge and mobile nodes. Moreover, Fluidity allows the application to provide its own adaptation policies and switch between them at runtime. The core mechanisms of the Fluidity framework have a non-negligible but still acceptable overhead, allowing a wide range of applications that do not have tight real-time requirements to exploit edge computing and networking resources in a flexible way.

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.

7 References

[1] Ardupilot SITL. `http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html`.

[2] K. Bhardwaj, M.-W. Shih, P. Agarwal, A. Gavrilovska, T. Kim, and K. Schwan. Fast, scalable and secure onloading of edge functions using Airbox. In *IEEE/ACM Symposium on Edge Computing*, pages 14–27, 2016.

[3] J. Carrasco, J. Cubo, and E. Pimentel. Towards a flexible deployment of multi-cloud applications based on TOSCA and CAMP. In *European Conference on Service-Oriented and Cloud Computing*, pages 278–286, 2014.

[4] Docker. `https://www.docker.com/`.

[5] Dronekit. `http://dronekit.io/`.

[6] A. Ferreira, E. V. Hensbergen, C. Adeniyi-Jones, E. Grimely-Evans, J. Minor, M. Nutter, L. E. Peña, K. Agarwal, and J. Hermes. SMARTER: Experiences with cloud native on the edge. In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge)*, 2020.

[7] N. Grigoropoulos and S. Lalis. Fractus: Orchestration of Distributed Applications in the Drone-Edge-Cloud Continuum. In *IEEE 46th Annual Computers Software and Applications Conference (COMPSAC)*, pages 838–848, 2022.

[8] S. He, F. Bastani, A. Balasingam, K. Gopalakrishnan, Z. Jiang, M. Alizadeh, H. Balakrishnan, M. J. Cafarella, T. Kraska, and S. Madden. Beecluster: drone orchestration via predictive optimization. In *International Conference on Mobile Systems, Applications and Services*, pages 299–311, 2020.

[9] A. V. Hof and J. Nieh. AnDrone: Virtual drone computing in the cloud. In *Eurosys*, pages 6:1–6:16, 2019.

[10] Istio service mesh. `https://istio.io/`.

[11] K3S. `https://k3s.io/`.

[12] T. Kasidakis, G. Polychronis, M. Koutsoubelias, and S. Lalis. Reducing the mission time of drone applications through location-aware edge computing. In *IEEE International Conference on Fog and Edge Computing (ICFEC)*, pages 45–52, 2021.

[13] A. Koubâa, B. Qureshi, M.-F. Sriti, A. Allouch, Y. Javed, M. Alajlan, O. Cheikhrouhou, M. Khalgui, and E. Tovar. Dronemap Planner: A service-oriented cloud-based management system for the Internet-of-Drones. *Ad Hoc Networks*, 86:46–62, 2019.

[14] K. Kritikos and P. Skrzypek. Towards an optimized, cloud-agnostic deployment of hybrid applications. In *International Conference on Business Information Systems*, pages 435–449, 2019.

[15] K. Kritikos, C. Zeginis, E. Politaki, and D. Plexousakis. Towards the modelling of adaptation rules and histories for multi-cloud applications. In *International Conference on Cloud Computing and Services Science*, pages 300–307, 2019.

[16] KubeEdge. `https://kubeedge.io/`.

[17] Kubernetes. `https://kubernetes.io/`.

[18] P. Liu, D. Willis, and S. Banerjee. Paradrop: Enabling lightweight multi-tenancy at the network's extreme edge. In *IEEE/ACM Symposium on Edge Computing*, pages 1–13, 2016.

[19] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.

[20] J. Yapp, R. Seker, and R. Babiceanu. UAV as a service: Enabling on-demand access and on-the-fly re-tasking of multi-tenant UAVs using cloud services. In *IEEE/AIAA Digital Avionics Systems Conference*, 2016.

[21] H. Zhang, N. Liu, X. Chu, K. Long, A. H. Aghvami, and V. C. M. Leung. Network slicing based 5G and future mobile networks: Mobility, resource management, and challenges. *IEEE Communications Magazine*, 55(8):138–145, 2017.