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, Nikolaos Bellas and Spyros Lalis

Department of Electrical and Computer Engineering, University of Thessaly
Volos, Greece
{spournar, patras, cda, nbellas, lalis}@uth.gr

1. Motivation & Goal

- Microservice-based applications affect the operation of modern data centers
- Serverless paradigm hides the management of the underlying infrastructure
- Edge computing can improve QoS requirements compared to cloud operation

Goals:

1. Support flexible and adaptive deployment and orchestration across the continuum
2. Enable policy-driven management through a structured API
3. Achieve a scalable implementation of the deployment and telemetry mechanisms

2. Fluidity Architecture

Cluster setup

- Nodes connected via a VPN on top of Kubernetes and Open Telemetry

Fluidity Controller

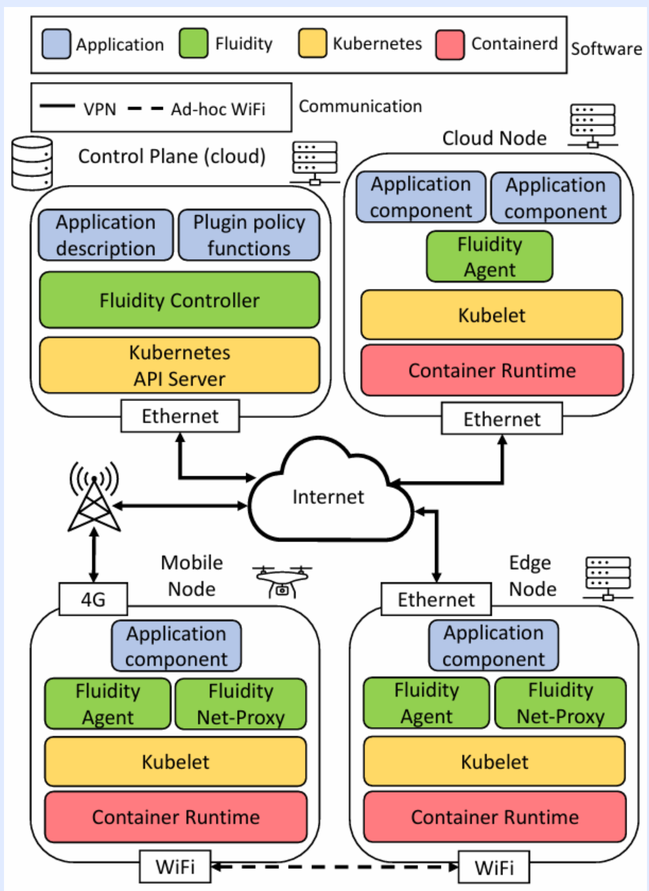
- Resides in the cloud
- Processes deployment requests
- Monitors the state of the system
- Executes custom policy implementations and adapts the current deployment plan
- Supports component deployment, relocation, and removal

Fluidity Agent

- Registers the node to the cluster and sends node state information

Fluidity Net-Proxy

- Performs application traffic redirection



3. Approach & Application-level Abstraction

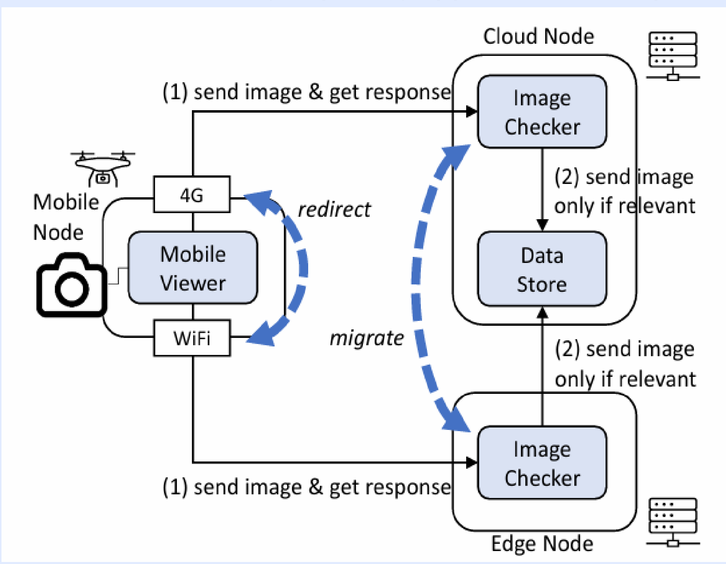
- The user merely provides to the system an application description capturing resource requirements and interactions between components
- Application components can be dynamically placed and executed across edge-cloud-mobile nodes based on app & infra descriptions
- Both app & infra descriptions are represented as Kubernetes Custom Resource Definitions

```yaml
kind: Application
name: ObjectDetectionApp
components:
- name: MobileViewer
  podSpec: PodMobileViewer.yaml
  placement: mobile
  systemServices:
    camera:
      methods: [CaptureImage, RetrieveImage]
      sensorType: RGB
  egress: [ImageChecker]
- name: ImageChecker
  podSpec: PodImageChecker.yaml
  placement: hybrid
  ingress: [MobileViewer]
  egress: [DataStore]
- name: DataStore
  podSpec: PodDataStore.yaml
  placement: cloud
  ingress: [ImageChecker]
Policy:
- name: simple_policy.py
```

Indicative (simplified) application description

- Running example: object detection with 3 components
- Available reconfiguration knobs:

1. Component placement/relocation
2. Data traffic redirection from 4G to WiFi (the default channel for control messages remains the same)



Placement of application components and dynamic relocation with traffic redirection

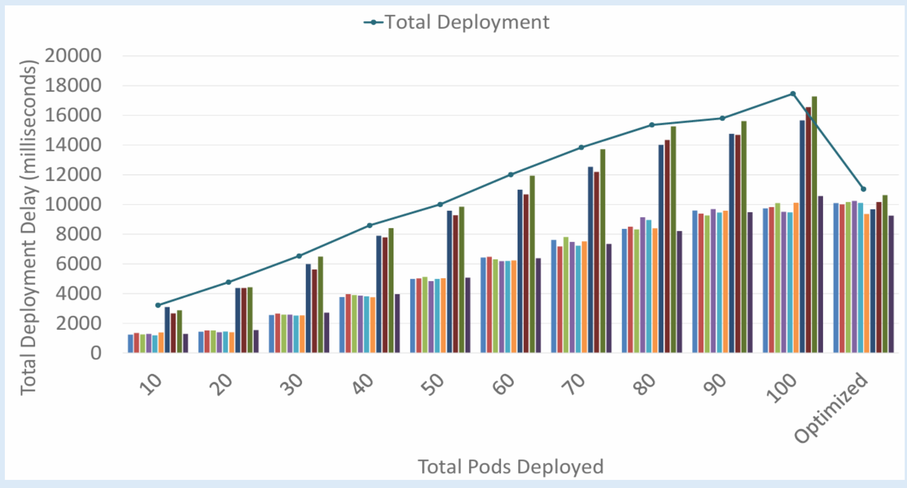
4. Plugin Policy API

Custom policy implementation supporting:

1. Initial application deployment
2. Application & infrastructure monitoring
3. Deployment plan readjustment

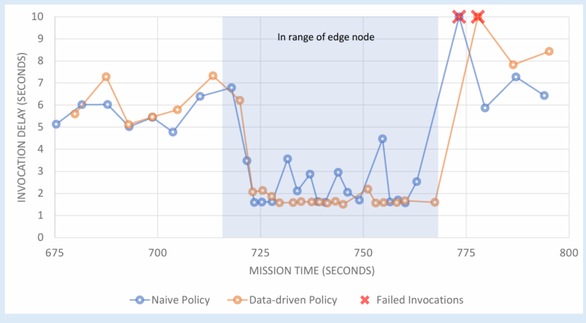
```python
# Produce initial deployment plan
def initial_plan(app_desc, nodes):
    # Custom logic implementation
    return plan, context

# 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
```

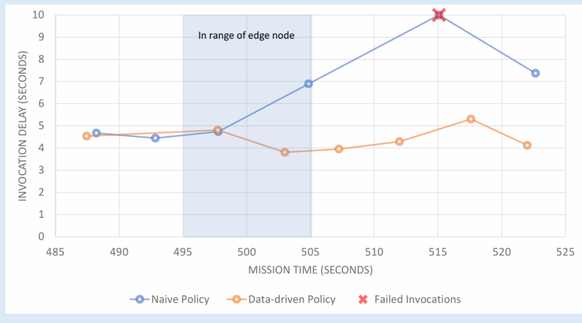
5. Evaluation



Experiment 1 – Parallel execution of a scale-out component pattern to 10 nodes/VMs for different values of total pods.

- 100 Pods deployed (10 to each node), the total delay is 17.4 seconds
- **6.8x speedup vs sequential deployment of 100 Pods**
- **Optimized placement based on recorded delays yields a 10.76x speedup (11 seconds)**
- **18% maximum CPU utilization for the control plane**





Experiment 2 – Evaluation of the end-to-end overhead for the execution of different deployment plans

- Average end-to-end adaptation delay is 4.9 seconds, mainly due to the Pod deployment (1.6 seconds) and application traffic redirection (3 seconds)
- **Policy switching** at runtime is less than 25 milliseconds
- **Application performance evaluation based on 2 adaptation policies**
- Naive policy: Relocates the service-providing component whenever the drone is in range of an edge node: 14% invocation failure ratio (28 relocations)
- Data-driven policy: Estimates performance based on historical data and relocates only when the predicted performance is beneficial: 3.7% invocation failure ratio (8 relocations)
- Invocation delay: 64% reduced at the edge vs the cloud

Experiment 3 – Telemetry scalability on a Raspberry Pi with a 4G interface

- For a single component, the average network traffic sent to the control plane is 5.5 Kbps, and for 10 components, the respective traffic is 39 Kbps **(<0.02%** of the available bandwidth of a 2 Mbps wireless link)
- The average latency of telemetry is between 130 and 250 milliseconds for an underutilized (without external traffic) and stressed link (75% utilized using external traffic), respectively, having acceptable performance even for nodes operating at the edge

Funded by the European Union

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

MLSysOps