Towards Adaptive Actors for Scalable IoT Applications at the Edge

Jonathan Fürst[1,2], Mauricio Fadel Argerich[1,3], Kaifei Chen[4], Ernö Kovacs[1]
NEC Labs Europe[1], IT University of Copenhagen[2], Sapienza University[3], UC Berkeley[4]

VLIoT @VLDB'18, Rio de Janeiro
August 31, 2018
jonathan.fuerst@neclab.eu

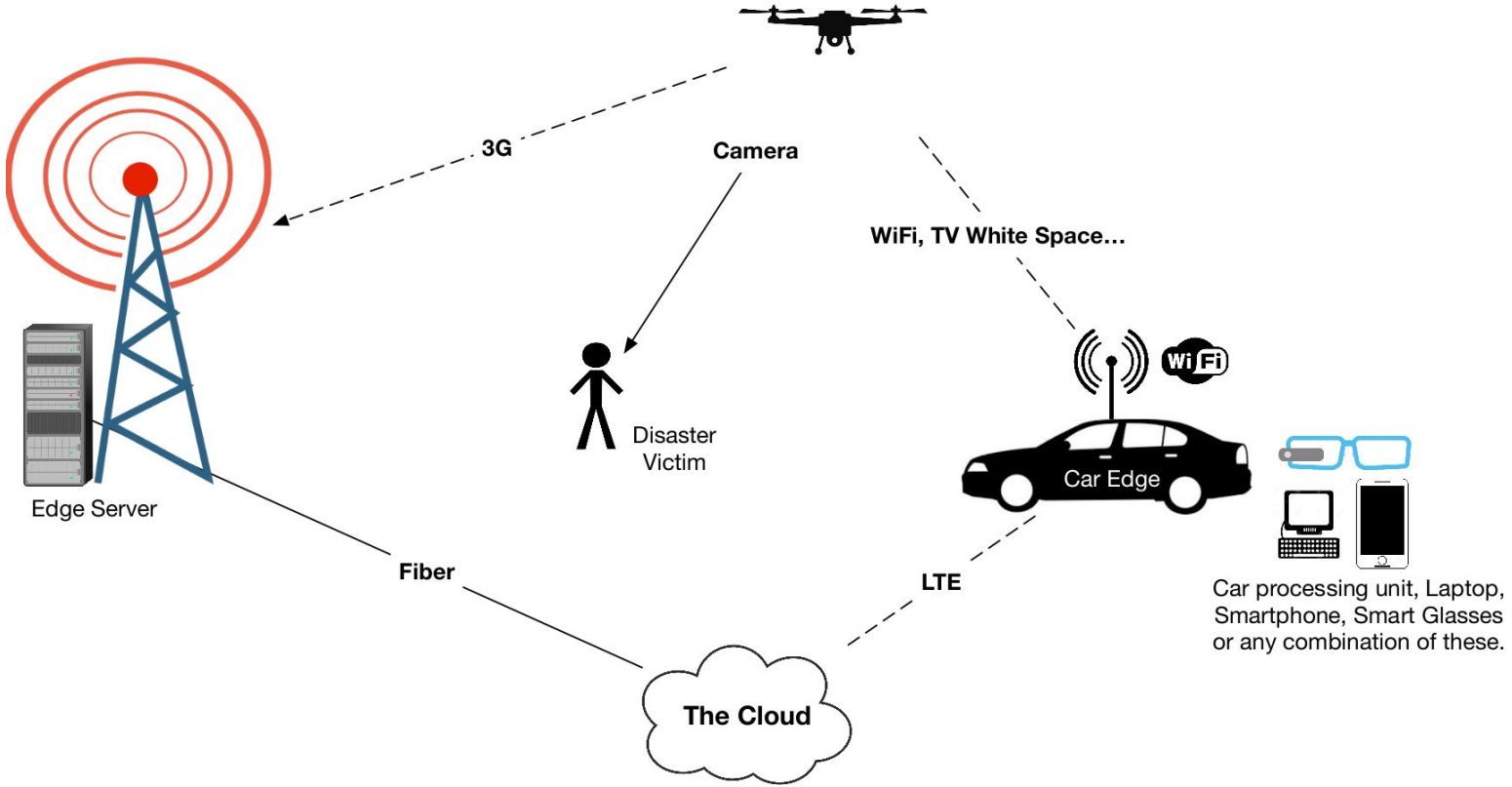
Outline

- Introduction & Motivation
- **Nandu**
  - System Overview
  - Programming Model
  - Adaptation Strategies
  - Experimental Results
- Conclusions & Future Work

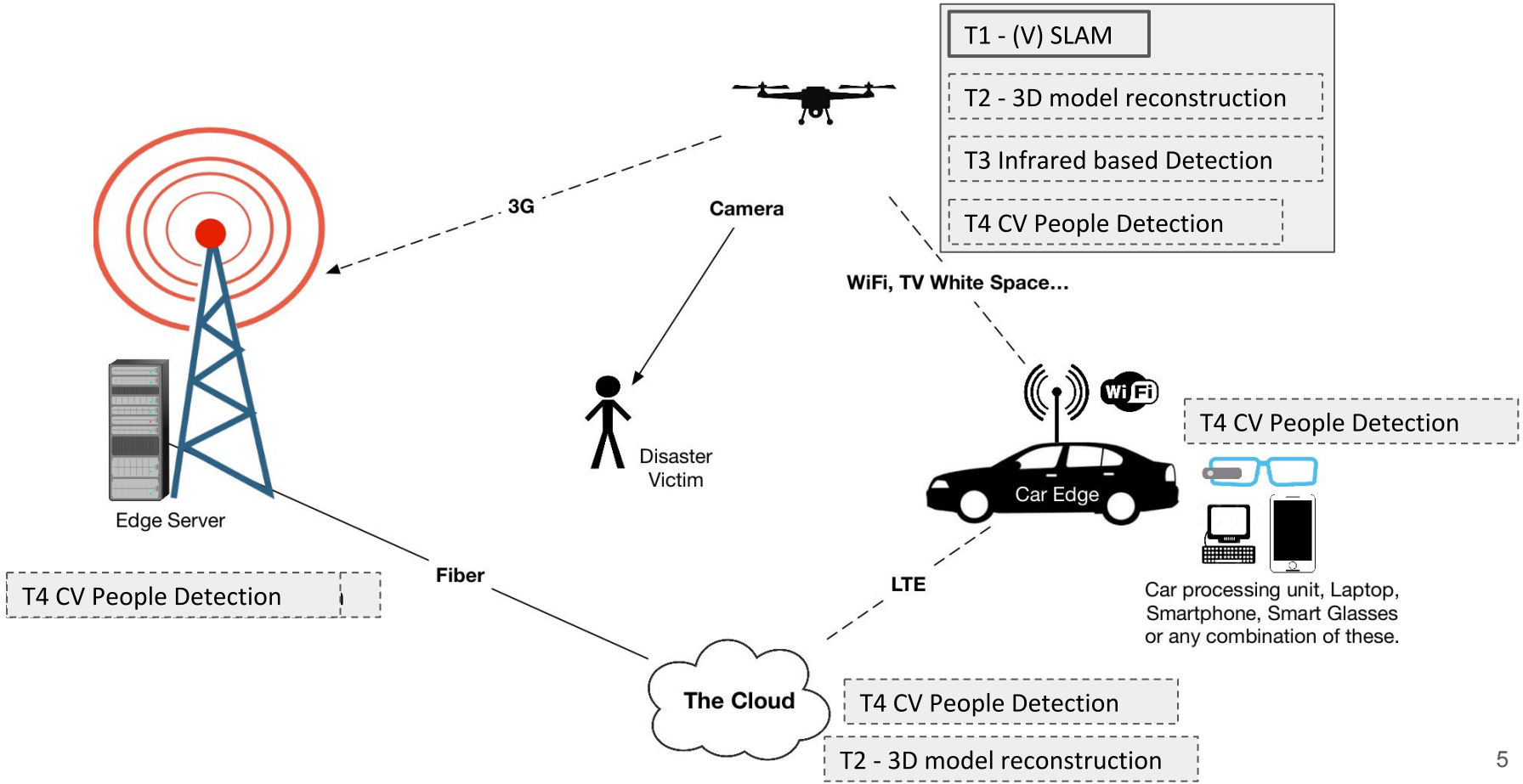
IoT Paradigm Shift

- **Data Flow is being reversed**
  - Traditional: Content distribution from core to edge.
  - IoT: Generates (huge amounts of) data at the network edge and move towards core to process in cloud.
- **Diverse** and **interdependent QoS requirements** of IoT applications
  - Complex tradeoffs between **responsiveness, accuracy, power consumption**, and **cost**.
- **Rich clients** with processing power, that make timely and local decisions
  - E.g., drone, autonomous vehicle, smartphone…

Application Scenario



Application Scenario



Challenges

Programming such distributed and heterogeneous edge-cloud systems is hard and requires complex decisions on:

(1) Application **partitioning** into tasks, — **Static**

(2) Task **allocation** and
(3) Task **adaptation**. — **Dynamic**

Current Context:
- Availability of Edge, Cloud… resources.
- Networking
- Battery level

Requirements/Constraints:
- Accuracy
- Cost
- Latency
- Power Consumption

→ **Should be supported by a framework.**

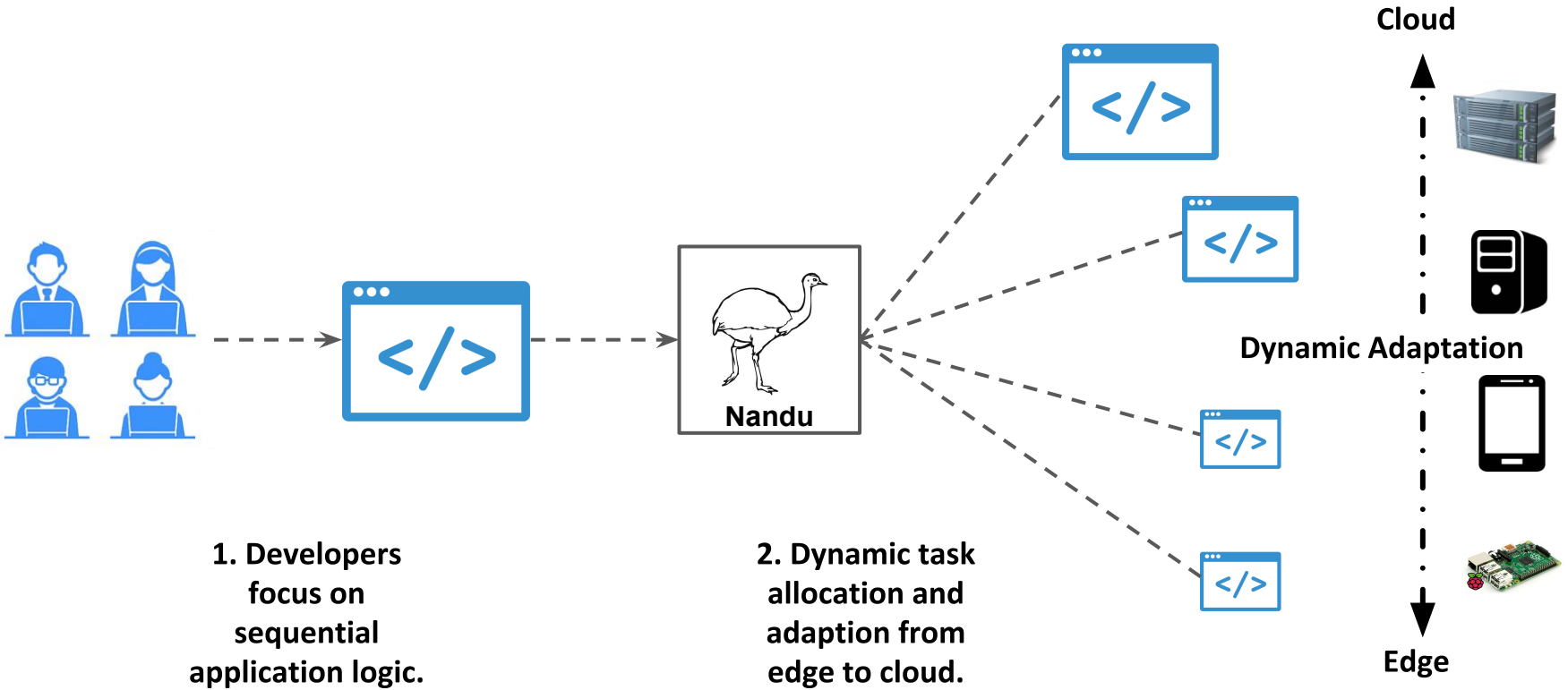
Rich Set of Related Work

- **Distributed Execution Frameworks**
  - MagnetOS (MobiSys05), Sapphire (OSDI14), FogFlow (IEEEIoT17), Ray (HotOS17)
  - **Community/Industry:** Dask, Erlang (to some extent), Serverless Computing (AWS Lambda, Google Functions).
- **Adaptive Programming**
  - Senergy (OOPSLA13), ENT (PWDL17), AWStream (SIGCOMM18)
- **Mobile offloading**
  - MAUI (MobiSys10), CloneCloud (EuroSys11), COMET (OSDI12)
- **But:**
  - Do not deal with **both task adaptation and allocation**.
  - Are **centralized** through a scheduler.
  - Mobile offloading works deal with **mobile-cloud scenario only**.

→ Not designed for IoT (but mobile computing, data analytics etc.)

Nandu Framework

Usage Scenario



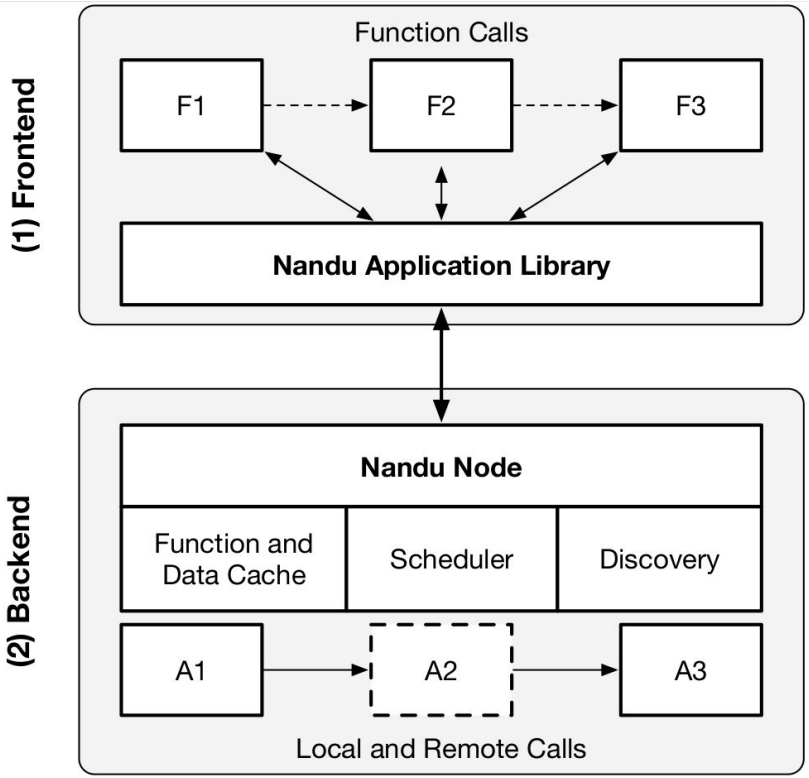
Key Design Goals

1. **Dynamic discovery mechanism.** Mobile IoT devices like drones or smartphones need to discover close-by available computational resources in order to distribute their tasks.

2. **Abstract execution mechanism away from application logic.** Programmers are only required to make annotations (e.g., can this function be offloaded or not?) to their functions to transform the application into a distributed one.

3. **Dynamic adaptation.** Runtime needs to dynamically adapt and migrate application tasks, depending on the current execution context. We need adaptation mechanisms to dynamically adjust function parameters or choose different task implementations to maximize the utility according to application requirements.

System Overview



Two parts:

1. **Application library (frontend)**
2. **Node (backend)**

Each Nandu node can host a variable number of actors. Actors can communicate with other actors on the same node and with actors on remote nodes through RPC calls.

Programming Model

- **Overall application goals**
  - Non-functional requirements that should be achieved by the application or application components. E.g., overall processing pipeline should occur in < 1s.
    → must define a measurable proxy (metric) that captures a requirement well.

- **Individual function hints**
  - Indicate that:
    1. A function is adaptable,
    2. Which of its parameters can be adapted and
    3. The expected application utility of different adaptation values.

Programming Model: Example

RTT < 1s

```
1 from nandu import adapt
2 @adapt(offload = True,
3         size = {640: 10, 480: 9, 320: 6, 240: 3})
4 def resize_image(img, img_id, size = 640):
5     import imutils
6     return(imutils.resize(
7       img, width=min(size, img.shape[1])),
8       img_id)
```

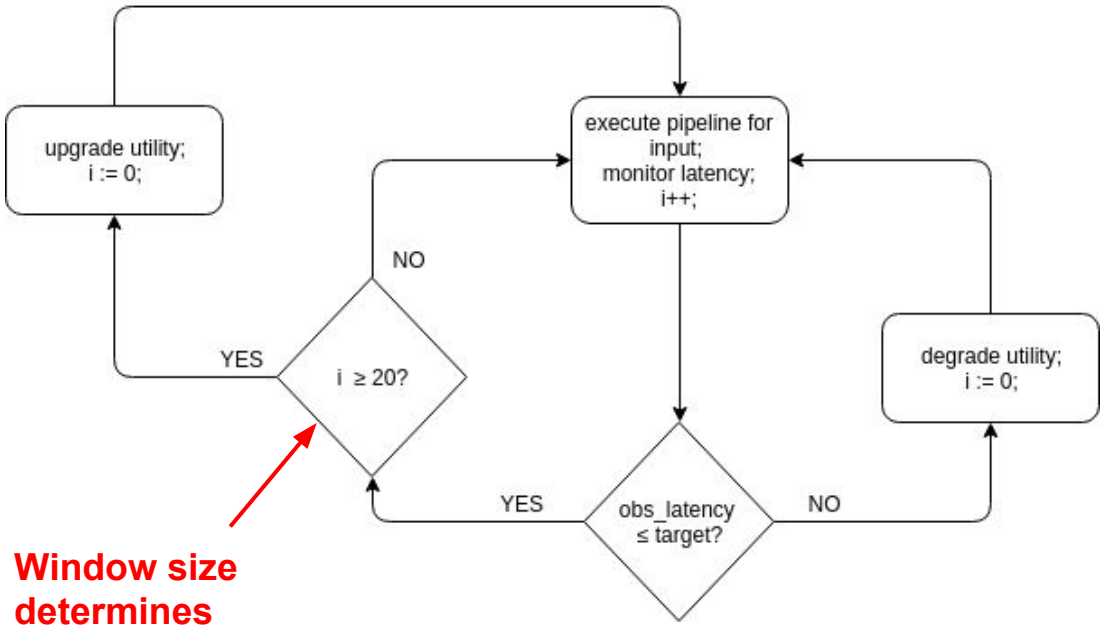
Utility value

Image width in px

Adaptation: Strategies

1. **Actor Migration**
   Migrate actors to more powerful nodes.
2. **Data Degradation**
   Degrade actor input in order to adapt the application to changing network conditions or available resources.
3. **Actor Degradation**
   Select different actor implementation (e.g., a different classifier).

Adaptation: Optimization



Window size determines responsiveness to improvements.

Main Goal:

- **Maximize utility** for current execution context while adhering to overall application goals (e.g., latency requirements).

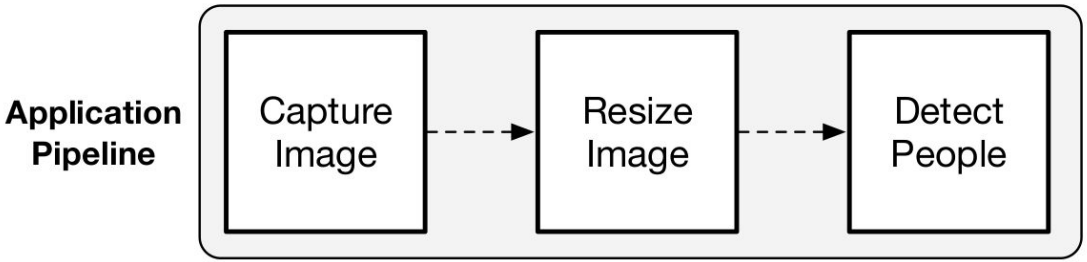
Experimental Results

Implementation

Prototype in Python 3.6:

- Python decorators to allow developers to specify adaptation hints.
- Python's future statement as placeholder for a function output.
- Cloudpickle for object serialization, RPC server and client using asyncio module.

People Detection Pipeline:



- Computer Vision-based people detection
- pre-trained HOG + LinearSVM model

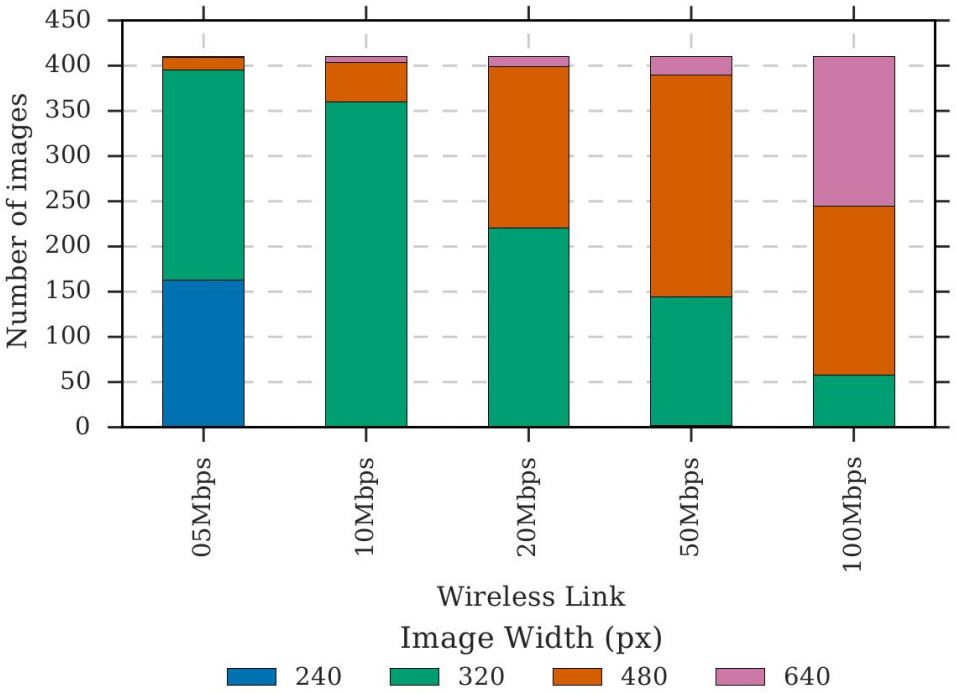
Experimental Setup

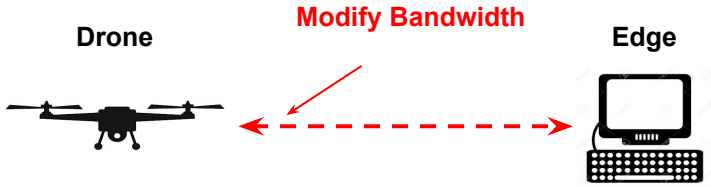
- Mininet (http://mininet.org/) for simulating different network bandwidths.
- INRIA Person Dataset with 410 images.
- Latency target < 1s.

- Three differently implemented classifiers:

Classifier accuracy	Low	Medium	High
Scale	1.30	1.15	1.05
Accuracy (%)	80.7	85.7	88.8
Proportional proc. time (%)	100	145	351

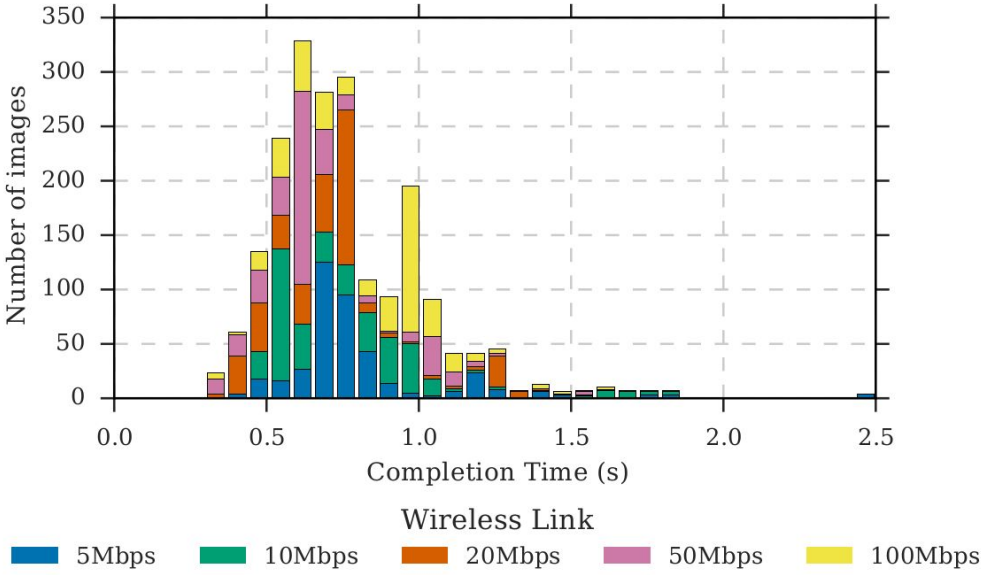
Input Degradation



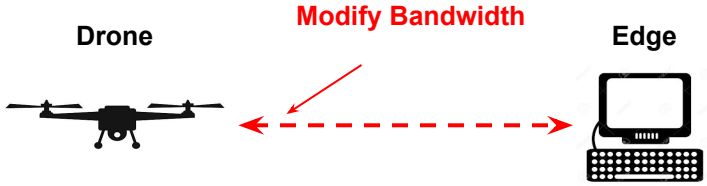


- Nandu uses higher resolution images to keep the highest accuracy possible while not exceeding the target latency.
- The optimizer can not pick an optimal image size value. E.g., Nandu selects both, 480 px and 640 px for 100 Mbps, because perceived latencies of 480 px are < 1 s, while latencies of 640 px are slightly larger than 1s
  → optimal value would be somewhere in between.

Input Degradation

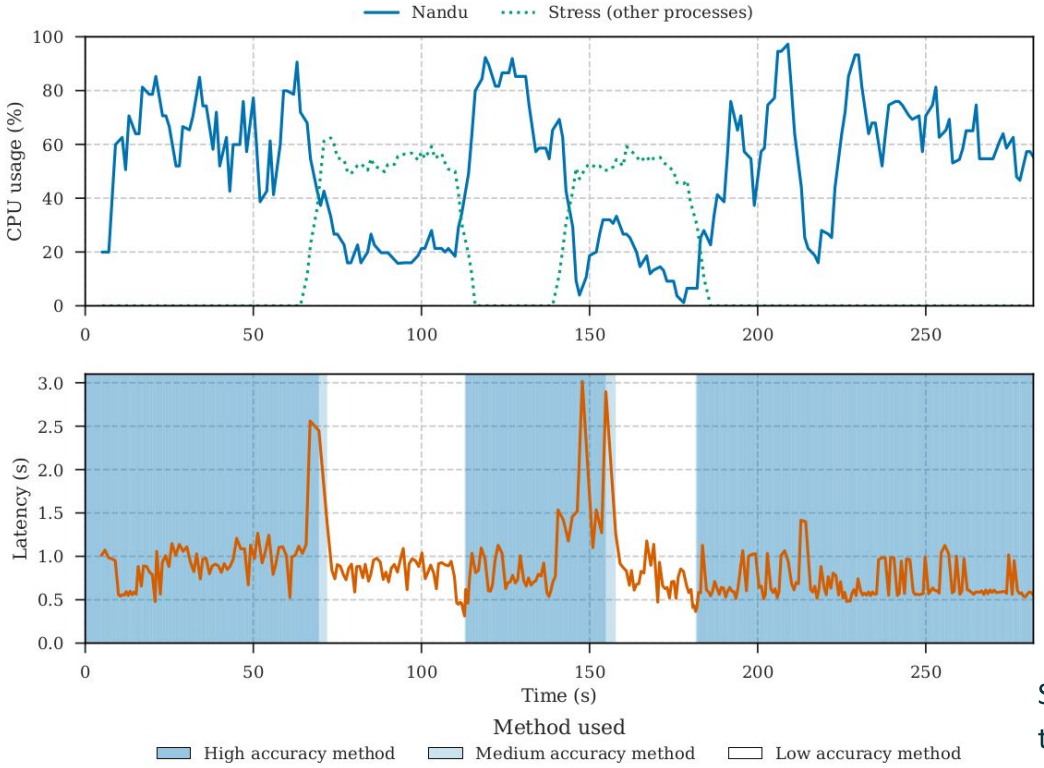


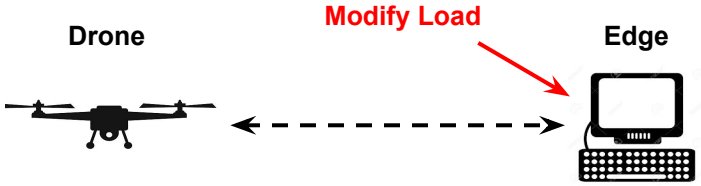
Wireless Link (Mbps)	5	10	20	50	100
Mean Time (s)	0.82	0.76	0.72	0.69	0.85
Accuracy (%)	87.3	88.8	86.8	88.8	90.0



- Even with sub-optimal, discrete parameter selection, majority of images are processed in < 1s.
- Mean accuracy for all bandwidth speeds is 88.34 %, which is higher than what we might achieve with a static parameter selection (83.6 %).

Actor Degradation

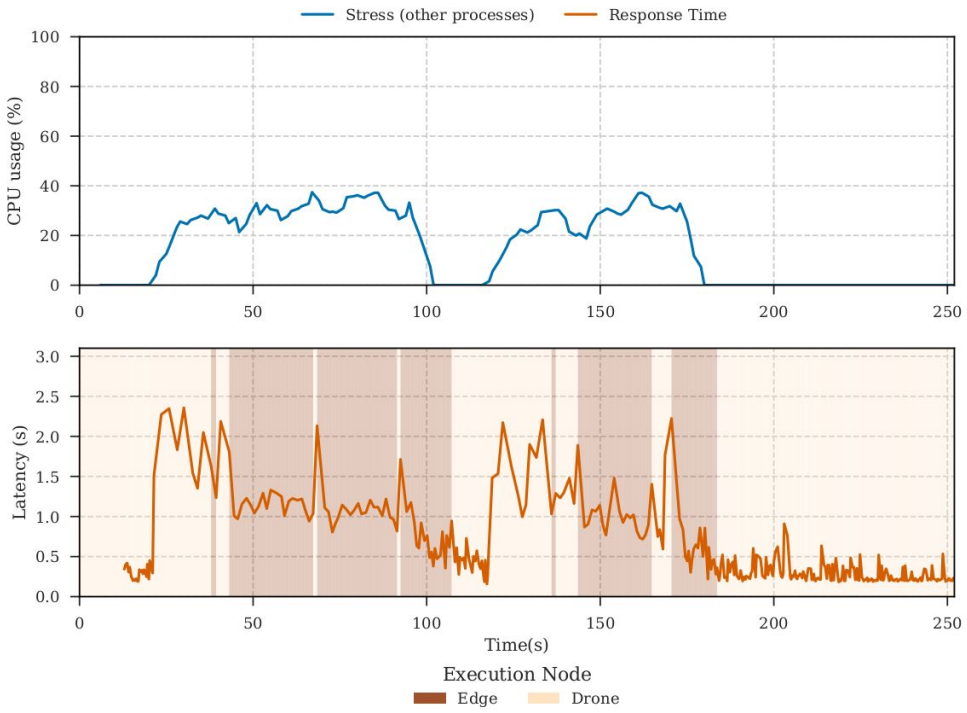


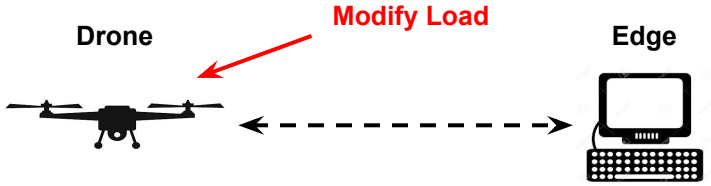


- Increased load on the hosting node leads to increased latency after 60s and to dynamic actor adaptation to stay in QoS latency requirements.
- When CPU resources become available again (e.g., at 180s), Nandu returns to higher accuracy classifier.

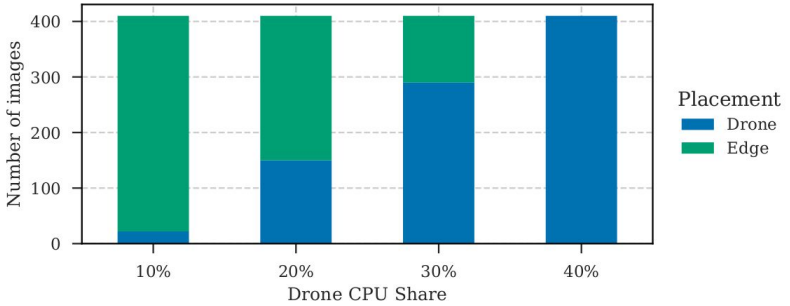
Stress tool (https://people.seas.harvard.edu/~apw/stress/) to simulate additional load on a node.

Actor Migration





- Simulate additional load on the drone node to trigger actor migration to edge node.
- Mean latency of 0.7s.



Wrap-Up

Takeaways, Limitations and Future Work

Takeaways: Framework for adaptive programming at the edge

- Programming model that allows execution framework to perform fine-grained adaptations during runtime in unison with developer hints, in order to make applications conform to non-functional constraints, such as latency.

Current Limitations:

- Possible to define as many constraints as necessary, but only one utility objective.
- Utility value for any combination of parameters is assumed to be known or easily computed.

Future Work:

- Automate assignment of utility values (e.g., through offline profiling).
- Reinforcement Learning to replace current heuristic approach for optimization.
- Continuous parameters instead of discrete set.
- Integration of concepts in NEC FogFlow framework [IEEEIoT17] + [CNSM18].

Thank you!

Any questions?

jonathan.fuerst@neclab.eu

NEC

IT University of Copenhagen

Sapienza Università di Roma

Berkeley University of California