Elastic Services for Edge Computing

Jonathan Fürst*†, Mauricio Fadel Argerich*§ and Bin Cheng*
*NEC Labs Europe, §Sapienza Università di Roma, †IT University of Copenhagen
{jonathan.fuerst, mauricio.fadel, bin.cheng}@neclab.eu

Apostolos Papageorgiou
i2CAT Foundation
apostolos.papageorgiou@i2cat.net

***Abstract*—Edge computing enables new, low-latency services close to data producers and consumers. However, edge service management is challenged by high hardware heterogeneity and missing elasticity capabilities. To address these challenges, this paper introduces the concept of elastic services. Elastic services are situation aware and can adapt themselves to the current execution environment dynamically to adhere to their Service Level Objectives (SLOs). This adaptation is achieved through Diversifiable Programming (DivProg), a new programming model which uses function annotations as interface between the service logic, its SLOs, and the execution framework. DivProg enables developers to characterize their services in a way that allows a third-party execution framework to run them with the flow and the parametrization that conforms to changing SLOs. We develop a prototype and perform an experimental evaluation which shows that elastic services can seamlessly adapt to heterogeneous platforms and scale with a wide range of input sizes, while adhering to their SLOs with little programming effort.**

***Index Terms*—IoT, services, adaptive, programming model**

I. Introduction

The proliferation of tightly coupled data producers and consumers at the endpoints of the Internet challenges traditional cloud-centered data processing and service placement. The reasons are tight responsiveness requirements (e.g., for applications involving user interaction like augmented reality (AR) [1] or smart appliances [2]), strict data privacy requirements (e.g., for camera surveillance data) and the depletion of existing network uplink resources for such massive reverse data flow [3].

A solution to these problems is to move services closer to producers and consumers as proposed in Edge and Fog computing [4], [5]. Previous work [6], [7] shows that this can greatly improve responsiveness, bandwidth use and power consumption of existing services. However, deploying services at the edge creates different challenges as opposed to deploying them in a cloud-scale data center:

Hardware Heterogeneity. Edge IoT services might run on low power, Raspberry Pi form factor devices, traditional x86–64 machines, high-performance edge servers, equipped with powerful GPUs, or highly optimized application-specific integrated circuits (e.g., Google's tensor flow processing unit [8]). The result is that a service has widely different performance characteristics depending on where it is deployed.

Elasticity. The cloud provides elasticity mechanisms, which the edge cannot provide [9]. Services in the cloud scale horizontally over multiple machines or vertically by increasing their RAM and CPU share. The edge might consist of a single machine with limited hardware capabilities for vertical scaling. Edge load is highly dynamic (e.g., determined by the number of close-by AR users).

Different approaches address these challenges. Saurez et al. [10] present a container based distributed execution framework for edge-cloud applications, that uses Quality of Service (QoS) driven task placements. Likewise, Villari et al. [11] envision the concept of osmotic computing, where micro services are deployed opportunistically in both cloud and edge based on QoS requirements and current execution context. The assumption in these works is that the cloud is an always available resource to which tasks can migrate dynamically.

This is not true for many IoT scenarios, where task migration might be (temporarily) restricted due to service responsiveness requirements (e.g., AR response time should be $< 100\,\mathrm{ms}$ [12]), bandwidth limitations and fluctuations (e.g., in wireless settings), expensive, due to power consumption (radio communication can govern power consumption in wireless networks [13]), or ruled out due to privacy requirements (e.g., sensitive camera data). This paper deals with managing such IoT service deployments, where service migration to the cloud is not an option for some, or several of these reasons.

Our key insight is that we need to *bring elasticity properties to the edge* itself. Due to limited hardware capabilities, it is not possible to apply existing cloud elasticity mechanisms, like horizontal and vertical scaling to edge hosted services. Thus, our approach is to make the service itself elastic by introducing the concept of *elastic services*. Elastic services are situation aware and can adapt themselves to the current execution environment dynamically, jointly with the execution platform to stay inside their Service Level Objectives (SLOs). We enable these elastic services through two main contributions:

1) Diversifiable Programming (DivProg), a new programming model, which uses function annotations as interface between the service logic, its SLOs, and the execution framework. *DivProg* enables developers to characterize their services in a way that allows a third-party execution framework to execute them with the flow and parametrization that conforms to changing SLOs.
2) Finally, we develop a prototype and perform an experimental evaluation which shows that elastic services can execute on heterogeneous platforms while adhering to their SLOs without much programming effort. Compared to a non-elastic service implementation, we improve latency three-fold ($0.89\,\mathrm{s}$ vs. $3.3\,\mathrm{s}$), adapting to the hardware platform, and increase input elasticity by a factor of 16.

978-3-903176-14-0

II. Gap Analysis of Current Service Design

To motivate elastic services for edge computing, we introduce a camera based IoT service as application scenario. Modern cities contain a wide range of connected cameras. Their camera data can be the input for a variety of smart city applications. For example, they can be used to count people and understand crowd mobility [14], but can also be used to locate specific persons like a lost or abducted child [15]. We consider this "lost child use case" as application scenario for our paper: (1) a camera captures images, (2) a face detection classifier detects faces in the image and (3) a trained face recognition classifier matches found faces against the missing child. When we find a match, we send a notification with the child's location to nearby law enforcement.

When a child goes missing, the service is deployed using existing connected cameras and edge servers in the city. A small end-to-end latency is crucial to ensure a high frame sampling rate, so that we are unlikely to miss the child even if it only appears briefly on the video feed, e.g., when the child is moving: the Service Level Objective (SLO) is 1 s end-to-end latency, while maximizing the achievable accuracy. However, existing cameras are connected differently to upper layers (wireless, Ethernet) and have heterogeneous hardware capabilities [16], while some camera owners might enforce policies that disallow that the service is executed in the cloud due to privacy requirements. This introduces problems for the service to adhere to SLOs as we show now.

A. Limitations of Non-Elastic Services

We implement and deploy the lost-child service on several platforms, representing a range of potential camera and edge hardware: (1) a Raspberry Pi 1B (ARM Cortex-A7), (2) Raspberry Pi 3 (ARM Cortex-A53), (3) an x86-64 based server (Intel i7-4790) and (4) a typical public cloud VM on Google Cloud Platform (GCP) with 8 cores.

We then measure the dominant computation time for a single image frame using two established face detection and face recognition classifiers (Haar feature-based cascade classifiers [17] and Local Binary Patterns [18] respectively). Figure 1 depicts the cumulative distribution function (CDF) of the per frame execution time for our platforms. Both, the dedicated server (0.04 s) and the virtual machine (0.06 s) are well below the 1 s requirement. However, the execution times of the Raspberry PIs are larger by order of magnitude (RPi3: 0.6 s, or even exceed itself alone the SLO (RPi1: 3.3 s).

To understand the holistic service compliance to SLOs, we then measure the overhead of network latency (image upload round trip time) for different cloud geo-regions and for a local edge-server (WiFi, single hop) as depicted in Figure 2. Our results show that cloud computing adds notable latency overheads that themselves can already exceed overall service latency goals (e.g., for `us` and `asia` zones) or add substantial overhead when running in the same zone (`europe`: 0.5 s).

The combined results show that neither cloud nor edge can adhere to the required 1 s end-to-end latency in every execution scenario. Cloud computing is able to provide short execution

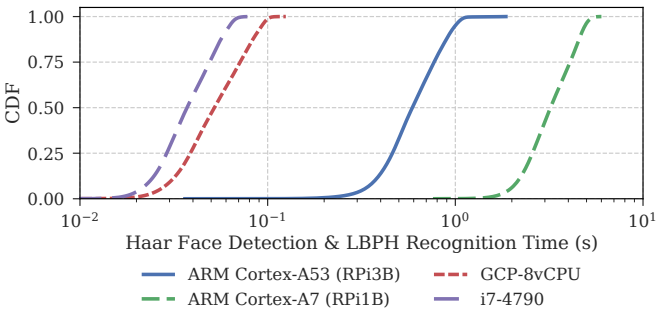

Fig. 1. Overall computation time for 640×460 PNG on different hardware: a Raspberry Pi 1B, a Raspberry Pi 3, a Google Cloud Platform VM and a server CPU (Intel i7-4790) with a GPU (GeForce GTX 950).

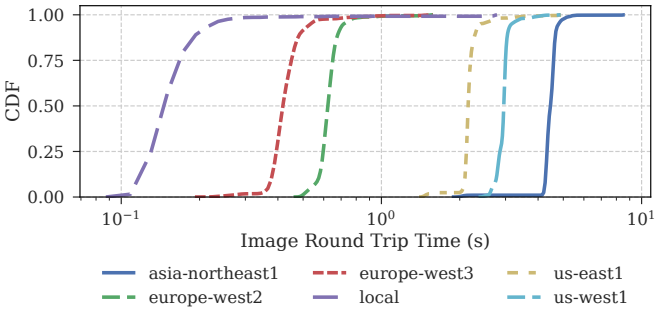

Fig. 2. 640×480 PNG upload round trip time to different geo-zones of Google Cloud Platform (europe-west3: 80km, europe-west2: 665km, us-east1: 7095km, us-west1: 8400km, asia-northeast1: 9405km) and a local lab server

times, but also high variances in latency dependent on geo-location of the cloud resource and network-link characteristics. Edge computing reduces these network latencies, but introduces high variance in execution times due to high hardware heterogeneity. A possible solution to this problem is that the service logic adapts to the execution environment, which introduces additional challenges for current service design.

B. Challenges Towards High Adaptability

IoT execution environments and the required service adaptations are highly complex and hard to capture for either service developers or operators alone. For example, Table I shows four possible runtime states that might occur together with possible adaptations for the lost child service. Even in this simple service, the possible runtime states and their implications on SLOs are complex or unknown to service developers (e.g., the expected number of faces in an image can only be estimated, is dynamic, deployment dependent, and out of control of the service logic). This complexity, in turn, leads to services being implemented with either a low degree of adaptability or with wrong or inefficient logic for selecting among possible modes and parameters (e.g., service logic might not adapt to the number of faces). We now present *DivProg*, a new programming model that deals with such problems.

III. Diversifiable Programming

DivProg is a model that abstracts the selection logic—i.e., the logic that determines how the function selection and their

TABLE I
EXAMPLE RUNTIME STATES AND THEIR POSSIBLE IMPLICATIONS

Runtime State	Possible Problem	Possible Adaptation
Host CPU overload (actual or predicted)	Lost child notification sent too late.	Use faster classifier to process image.
Many people in camera view.	Face-recognition slow, latency outside SLO.	Degrade parameters of classifier.
AI accelerator (GPU or dedicated ASIC) not available on host.	CPU based computation too slow to respect latency requirements.	Use "shallow" learning algorithm instead of deep learning.
Limited wireless link bandwidth to street cameras due to interference.	Video frames arrive delayed and cannot be processed in time.	Reduce image resolution, use sub-sampling to reduce bandwidth usage further.

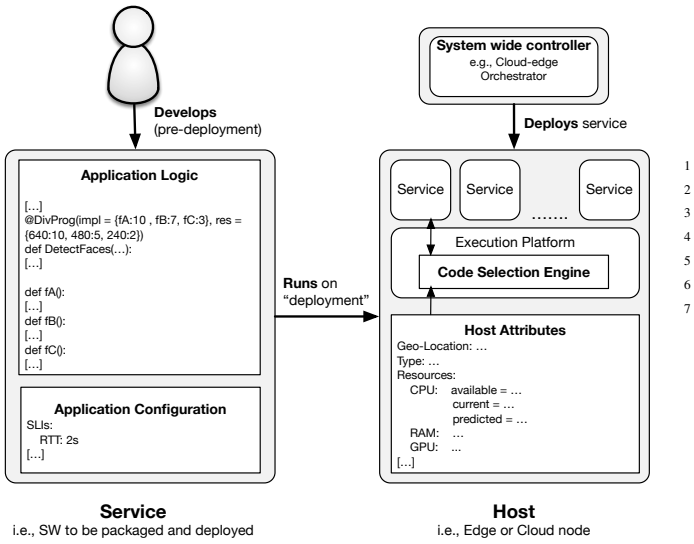

Fig. 3. System and programming model for developing adaptable applications

parameter values depend on different runtime states—to the platform, without requiring developers to know details of this selection logic or any respective APIs of the platform. Figure 3 depicts the overall architecture and works as follows.

First, developers provide function annotations in-line for their service implementation in cases where the function choices and their parameter values depend on the current runtime state (e.g., the network link, the host capabilities). As an example, a developer can annotate two face detection classifier functions with their respective utility (e.g., one is more accurate, but slower). Note that both classifiers implement the same functionality, namely detecting faces. Annotations are not limited to function choices, but developers can also provide a selection of different function parameter values, e.g., image sizes. Practically, this means that the actual service flow and parameter selection can be controlled by the platform developer, exploiting the in-line function annotations.

The service can now be packaged without explicit code selection logic and deployed on a host of the PaaS. It has the ability to interact with a (pluggable and potentially different for every host) code selection engine module. The host of the service and the values of the platform configuration parameters and host attributes change at runtime, leading the code selection logic to take different decisions and lead to different program flows each time. We now discuss our implementation of DivProg together with a specific platform code selection logic that enables elastic services for edge-computing.

IV. IMPLEMENTATION AND EVALUATION

We have implemented a DivProg prototype in Python 3.6 that builds on the asyncio library and uses the decorator design pattern for function annotations. Decorators provide a dynamic intersection point to the DivProg execution platform and allow us to execute the platform code selection logic. Listing 1 shows an example of our prototype. This function resizes an image, while abstracting the size selection to the platform. Each size is assigned a utility value between 1 and 10 towards the overall application goal (i.e., greater image sizes will provide better accuracy). This allows the platform to seamlessly reduce the image size to adhere to SLOs, like latency, when necessary.

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

Listing 1. Programming Model. Developers can annotate functions with hints for the platform on what parameters to adapt. In this example, the image size is adapted by the execution platform dynamically according to SLOs.

A. Code Selection Engine

In our prototype, we use a heuristic code selection engine that enables elastic IoT services through dynamic adaptation. Algorithm 1 depicts its working. The `Decorator` function is called whenever the actual service function is called. We then compare Service Level Indicators (SLIs) against SLOs and act accordingly by choosing different parameter values or function implementation. We obtain SLIs by profiling execution time on a function level. Dependent on the chosen velocity, we try to improve service quality in regular intervals by choosing parameter values or implementations with a higher utility (e.g., larger image sizes, more accurate classifiers).

Algorithm 1 Simplified Code Selection Logic

```
oldParams = {...}                          ▷ previous function parameters
windowSize = 20          ▷ tunable velocity parameter to fit application
i = 0
function DECORATOR(oldParams)
    if MMEAN(ObservedCosts, windowSize) > Constraint then
        params = DEGRADE(oldParams); i = 0
    else
        if i >windowSize then
            params = UPGRADE(oldParams); i = 0
        else
            params = oldParams; i++
        end if
    end if
end function
```

B. Evaluation

Using our prototype, we evaluate the analyzed challenges of managing IoT edge services: (1) hardware heterogeneity, and (2) missing elasticity mechanisms. Specifically, we implement the lost-child service and generate random compositions of face images from the published dataset in [19] as input data. Each composition contains a number of N faces.

Hardware Heterogeneity. We deploy the lost-child service on two low power edge platforms: the RPi1B and the newer RPi3B. We use 200 generated 1–6 faces compositions as input. Figure 4 shows the results of our elastic services on both devices together with a baseline implementation that uses a non-elastic service design. On the RPi3B, the non-elastic implementation works well (mean latency 0.3 s). However, the RPi1B with its less powerful hardware violates the 1 s latency requirement (3.3 s). The variance in the depicted results is due to the varied number of faces in each image (1–6). Looking at the elastic service results, both devices perform as desired: the RPi1B now conforms to the latency requirement by adapting; the newer RPi3B shows the same performance as before, because no adaption is required to meet SLOs. Note that the measured overhead of our framework and the implemented code selection logic is 4 ms (based on 1000 repetitions).

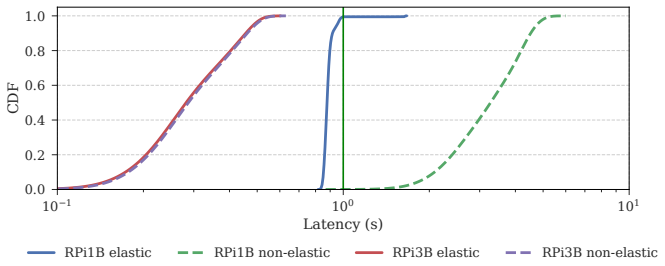

Fig. 4. Elastic Services adapt to hardware platforms according to SLOs.

Elasticity. To explore the elasticity of our service, we then deploy the same service on the RPi3, but increase the load by evaluating an increasing number of people, thereby testing elasticity. Specifically, we generate image compositions with 6, 12, 24, 48, 96 and 192 faces in them (1100 images overall). We then gradually input images from 6 to 192 faces and back to 6 faces into our service to simulate a dynamic crowd (e.g., during a public event). Figure 5 depicts this experiment. The elastic service is able to keep the latency inside SLOs (mean latency 0.53 s), while the non-elastic service violates SLOs with the growing number of people (e.g., 5.28 s for 192 faces). Elastic services use adaptation mechanisms to let the platform fit the service to the execution context. We use different classifiers for face recognition and detection as adaptation methods. Table II shows the accuracy-latency trade-offs for each method (non-elastic) and for the elastic service overall for 6–192 faces. The elastic service achieves better accuracy results than a low quality static implementation and only slightly worse results than the best quality implementation, while conforming to latency SLOs.

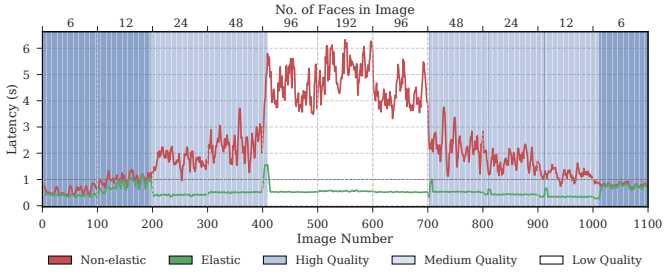

Fig. 5. Elastic Service Adapts to Number of People. We artificially increase the number of people in images (6, 12, 24, 48, 96 and 192). The elastic service adapts by selecting different classifiers to reach latency goals. The static service cannot scale with a growing number of people.

TABLE II
ACCURACY AND LATENCY FOR ELASTIC AND NON-ELASTIC SERVICES

		Face Detection	Child Recognition		95th
		Accuracy	**Precision**	**Recall**	**Percentile Time (s)**
Non-Elastic	High	0.993	0.990	0.541	5.84
	Medium	0.976	0.874	0.412	1.81
	Low	0.932	0.841	0.378	0.71
Elastic		0.963	0.854	0.412	0.91

V. Related Work

In the mobile community, several works facilitate context aware applications through better abstractions: Senergy [20] supports programmers in automating common latency, power, accuracy trade- offs. ENT [21] provides a type-based proactive and adaptive mode-based energy management at the application level, where developers characterize energy behavior of different program fragments with modes. We take some inspiration in these ideas but focus this work on (1) exporting service code selection logic to the execution platform and (2) on the application of this model for edge computing.

Stream processing systems have recently proposed principles of adaptive streaming (e.g., through data transformation techniques and dynamic degradation) [22]–[25]. We have also applied adaptation strategies for our edge computing prototype. However, we provide a more general programming model that allows platform dependent code selection logic.

VI. Conclusion and Future Work

We have presented the concept of elastic services that bring elasticity principles to edge computing through service adaptation. Our implementation focused on an elastic image processing service bound by latency SLOs. Compared to a non-elastic service implementation, we improve latency three-fold (0.89 s vs. 3.3 s), adapting to the hardware platform, and increase scalability by a factor of 16 (192 vs. 12 people in image). Our programming model is general enough to support many different SLOs and code selection logic. We are currently exploring privacy and cost objectives and their implications on code selection logic design and a formal definition of our annotation syntax.

ACKNOWLEDGEMENTS

This work has been partially funded by the European Union's Horizon 2020 research and innovation programme within the CPaaS.io project under Grant Agreement No. 723076.

REFERENCES

[1] K. Chen, J. Fürst, J. Kolb, H.-S. Kim, X. Jin, D. E. Culler, and R. H. Katz, "Snaplink: Fast and accurate vision-based appliance control in large commercial buildings," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 1, no. 4, p. 129, 2018.

[2] J. Fürst, K. Chen, M. Aljarrah, and P. Bonnet, "Leveraging physical locality to integrate smart appliances in non-residential buildings with ultrasound and bluetooth low energy," in *Internet-of-Things Design and Implementation (IoTDI), 2016 IEEE First International Conference on*. IEEE, 2016, pp. 199–210.

[3] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *Computer*, vol. 50, no. 10, pp. 58–67, 2017.

[4] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE pervasive Computing*, vol. 8, no. 4, 2009.

[5] F. Bonomi, R. Milito, J. Zhu, and S. A. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 2012, pp. 13–16.

[6] B.-G. Chun and P. Maniatis, "Augmented smartphone applications through clone cloud execution." in *HotOS*, vol. 9, 2009, pp. 8–11.

[7] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 2010, pp. 49–62.

[8] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 1–12.

[9] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere, "Edge-centric computing: Vision and challenges," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 5, pp. 37–42, 2015.

[10] E. Saurez, K. Hong, D. Lillethun, U. Ramachandran, and B. Ottenwälder, "Incremental deployment and migration of geo-distributed situation awareness applications in the fog," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. ACM, 2016, pp. 258–269.

[11] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan, "Osmotic computing: A new paradigm for edge/cloud integration," *IEEE Cloud Computing*, vol. 3, no. 6, pp. 76–83, 2016.

[12] J. Nielsen, *Usability engineering*. Elsevier, 1994.

[13] W. Ye, J. Heidemann, and D. Estrin, "An energy-efficient mac protocol for wireless sensor networks," in *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3. IEEE, 2002, pp. 1567–1576.

[14] F.-J. Wu and G. Solmaz, "Crowdestimator: Approximating crowd sizes with multi-modal data for internet-of-things services," in *MobiSys'18*, 2018.

[15] B. Cheng, G. Solmaz, F. Cirillo, E. Kovacs, K. Terasawa, and A. Kitazawa, "Fogflow: Easy programming of iot services over cloud and edges for smart cities," *IEEE Internet of Things Journal*, 2017.

[16] T. Zhang, A. Chowdhery, P. V. Bahl, K. Jamieson, and S. Banerjee, "The design and implementation of a wireless video surveillance system," in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*. ACM, 2015, pp. 426–438.

[17] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, vol. 1. IEEE, 2001, pp. I–I.

[18] T. Ahonen, A. Hadid, and M. Pietikäinen, "Face recognition with local binary patterns," in *European conference on computer vision*. Springer, 2004, pp. 469–481.

[19] L. Spacek, "Collection of facial images: Faces94," *Computer Vision Science and Research Projects, University of Essex, United Kingdom, http://cswww. essex. ac. uk/mv/allfaces/faces94. html*, 2007.

[20] A. Kansal, S. Saponas, A. Brush, K. S. McKinley, T. Mytkowicz, and R. Ziola, "The latency, accuracy, and battery (lab) abstraction: programmer productivity and energy efficiency for continuous mobile context sensing," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 661–676, 2013.

[21] A. Canino and Y. D. Liu, "Proactive and adaptive energy-aware programming with mixed typechecking," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2017, pp. 217–232.

[22] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman, "Aggregation and degradation in jetstream: Streaming analytics in the wide area." in *NSDI*, vol. 14, 2014, pp. 275–288.

[23] S. A. Noghabi, J. Kolb, P. Bodik, and E. Cuervo, "Steel: Simplified development and deployment of edge-cloud applications," in *10th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. USENIX Association, 2018 (to appear).

[24] A. Jonathan, A. Chandra, and J. Weissman, "Rethinking adaptability in wide-area stream processing systems," in *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. Boston, MA: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/hotcloud18/presentation/jonathan

[25] B. Zhang, X. Jin, S. Ratnasamy, J. Wawrzynek, and E. A. Lee, "Awstream: Adaptive wide-area streaming analytics," in *SIGCOMM'18*, 2018 (to appear).