Colony: Parallel Functions as a Service on the Cloud-Edge Continuum

Francesc Lordan[1][0000−0002−9845−8890], Daniele Lezzi[1][0000−0001−5081−7244], and Rosa M. Badia[1][0000−0003−2941−5499]

Department of Computer Sciences, Barcelona Supercomputing Center (BSC), Barcelona, Spain {francesc.lordan,daniele.lezzi,rosa.m.badia}@bsc.es

Abstract. Although smart devices markets are increasing their sales figures; their computing capabilities are not sufficient to provide good-enough-quality services. This paper proposes a solution to organize the devices within the Cloud-Edge Continuum in such a way that each one, as an autonomous individual –*Agent*–, processes events/data on its embedded compute resources while offering its computing capacity to the rest of the infrastructure in a Function-as-a-Service manner. Unlike other FaaS solutions, the described approach proposes to transparently convert the logic of such functions into task-based workflows backing on task-based programming models; thus, agents hosting the execution of the method generate the corresponding workflow and offloading part of the workload onto other agents to improve the overall service performance. On our prototype, the function-to-workflow transformation is performed by COMPSs; thus, developers can efficiently code applications of any of the three envisaged computing scenarios – sense-process-actuate, streaming and batch processing – throughout the whole Cloud-Edge Continuum without struggling with different frameworks specifically designed for each of them.

Keywords: Edge · Fog · Cloud · Compute Continuum · Distributed Systems · Programming model · Runtime system · Serverless · Function-as-a-Service · Stream-processing · Task-based Workflow.

1 Introduction

Embedding computing and networking capabilities into everyday objects is a growing trend; smart devices markets – e.g., phones, cameras, cars or speakers – are rapidly increasing their sales. Since the computing power available on them is often not sufficient to process the information they collect and provide a good-enough-quality service, such devices must cooperate in distributed infrastructures to share the hosting and processing of data.

In recent years, the Fog has raised as a complement to the established Cloud model by bringing down the compute and storage capabilities to the Edge. This mitigates the economic expenses and the network-related issues – high latency and low bandwidth – associated with the Cloud and enables new service opportunities relying only on on-premise, commodity devices. Mobile devices are a non-negligible source of resources; involving them in such platforms leads to an increase of the available computing power. However, mobility introduces dynamism to the infrastructure but also unreliability requiring to change the computational model to stateless and serverless.

Major Cloud providers relegate the Edge to be a serf of the Cloud neglecting all the intermediate devices within the network infrastructure. We envisage infrastructures not exclusively building on the Fog or the Cloud but exploiting the whole Cloud-Edge continuum by combining both paradigms. This work elevates the Fog as an autonomous peer of the Cloud functioning even when disconnected from it. This paper describes Colony, a framework for organizing the devices within the Cloud-Edge continuum resembling organic colonies: communities of several individuals closely associated to achieve a higher purpose. Each member of the colony – *Agent* – is an autonomous individual capable of processing information independently that can establish relations with other agents to create new colonies or to participate into already-existing ones. Agents offer the colony their embedded resources to execute functions in a serverless, stateless manner; in exchange, they receive a platform where to offload their computing workload achieving lower response times and power consumption.

In such environments, we identify three possible scenarios that require computation. On the first one, known as *sense-process-actuate*, the infrastructure provides a proper response to an event detected on one of its sensors. A second scenario is *stream processing*: sensors continuously produce data to be processed in real-time. Besides computations triggered by data or infrastructure changes, users can also request synchronous executions directly to the platform or submit jobs to a workload manager; this third scenario, named *batch processing*, is generally used for launching data analytics applications to generate new knowledge out of the information collected by or stored in the infrastructure. As discussed in detail in Section 6, developers must use different state-of-the-art frameworks to tackle each of these scenarios. To contribute to the current state-of-the art, we propose a single solution that tackles the three scenarios by taking a task-based approach. Converting the complex logic of the computation into an hybrid workflow – supporting both atomic and continuous-processing tasks – allows to parallelize and distribute the workload across the whole platform. For testing purposes and without loss of generality, our prototype leverages on COMPSs [27, 30] to make this conversion, and we modified the COMPSs runtime to delegate the execution of the nested tasks onto Colony. Nevertheless, other programming models following a task-based approach, such as Swift [34] or Kepler [15], could also be integrated in the framework with the appropriate glue software.

In summary, the contribution of this work is to bring together parallel programming, Function-as-a-Service (FaaS) and the Compute Continuum. For doing so, this paper describes Colony: a framework to create compute infrastructures following the recommendations of the OpenFog Consortium [19]. In it, each device offers its embedded computing resources to host the execution of functions in a service manner; however, by converting the logic of such functions into hybrid workflows – composed of atomic and persistent tasks –, the device can share the workload with the rest of the platform to achieve an efficient, parallel, distributed execution on any of the three computing scenarios. To validate the viability of the described design, a prototype leveraging on the COMPSs programming model is evaluated on two use cases.

The article continues by casting a glance over the components involved in our solution, and sections 3 and 4 respectively discuss the internals of an agent and how colonies are organized. Section 5 evaluates two use cases to validate the solution. Finally, Section 6 introduces related work and Section 7 presents the conclusions and identifies potential research lines to complete it.

2 Problem Statement and Solution Overview

The Cloud-Edge continuum is a distributed computing environment composed of a wide variety of devices going from resource-scarce single-board computers – e.g., Raspberry Pi – to the powerful servers that compose the datacenters supporting a cloud. Applications cannot assume any feature from the device where they run. Those devices closer to the application end user – within a PAN or LAN range – are often purpose-specific and dedicated to a single user. However, the further the device gets from the user, the more likely it is to be shared among multiple users and applications. For dealing with software heterogeneity and resource multi-tenancy, Colony proposes **virtualizing** the software **environment** either as a virtual machine or a container. Such environments would ensure that the necessary software is available on the devices running the application, and provide isolation from other applications running on the device.

For the sake of performance, applications can exploit **hardware heterogeneity** by running part of their logic on GPUs, FPGAs or other accelerators embedded on the devices. The underlying infrastructure might be composed of a large number of nodes, and centralizing the hardware-awareness of a dynamically-changing infrastructure on a single device might become a significant burden at computational and networking level. Therefore, each node must be able to work in a standalone manner and run the computation efficiently using all the computing devices embedded in it. For that purpose, each node of the infrastructure hosts a persistent service or agent that provides a Functions as a Service (FaaS) interface to **execute serverless functions** using the computational resources available on the device.

To overcome resource scarcity on edge devices, agent interaction is enabled to **offload task executions onto other agents** either on other edge devices, the Cloud or at any other device from an intermediate tier of the infrastructure. A typical feature of edge devices worth highlighting is mobility; mobile devices are likely to join and leave the infrastructure at any time with or without prior notice. Thus, agents should be aware of that and take appropriate measures to exploit all the computing resources at their best while preventing any failure due to the departure of any device. Both, functions and tasks, are computations totally independent of the state of the agent. All the necessary data bound to the computation is shipped along with the request – or at least a source from where to fetch the value –; thus, tasks can run on any agent of the infrastructure in a **serverless** manner. When new nodes join in the infrastructure, agents previously composing the infrastructure can offload onto them part of their computation, and vice versa. Conversely, when nodes leave the infrastructure, the **fault-tolerance mechanisms** must ensure the ongoing operations completion by resubmitting to still-available resources those tasks offloaded onto the no-longer-available agents.

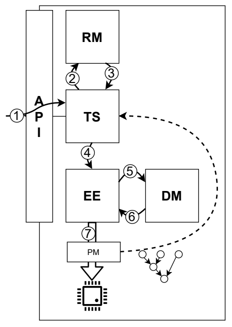
Unlike traditional distributed computing, applications no longer have a single entry point – the main method –; events may arise anywhere in the infrastructure and invoke a handler function to provide the appropriate response. When a node triggers a new execution, it contacts its local agent which orchestrates the effective completion of the operation. By leveraging on task-based parallel programming models, the logic of such functions can be automatically converted into workflows composed of several other isolated compute units (tasks) whose parallel execution can exploit the whole infrastructure when delegated onto other agents through Colony. Thus, if the programming model supports both atomic tasks and persistent tasks with continuous input/output, Colony can handle applications requiring any of the three aforementioned computing patterns: **sense-process-actuate**, **stream processing**, **batch processing**.

3 Agent Architecture

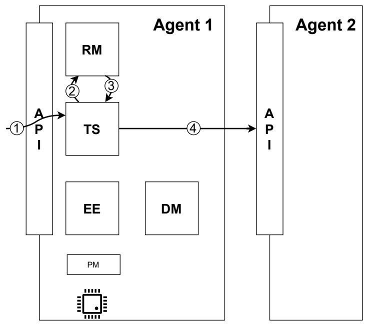
Agents are the cornerstone of the Colony platform. The purpose of an agent is to offer an interface where to request the execution of functions that will run either on the computing resources embedded on the host device or transparently offloaded onto other nodes on the Cloud-Edge Continuum. The entry-point to the agent is its API which offers methods for requesting the execution of a function with a certain parameter values and performing resource pool modifications.

Users or applications request a function execution detailing the logic to execute, the resource requirements to run the task, the dependencies with previously submitted tasks and the sources where to fetch the data involved in the operation. Upon the reception of a request, the API directly invokes the Colony runtime system. The goal of this runtime is to handle the asynchronous execution of tasks on a pool of resources. To achieve its purpose, the runtime has four main components. The first one, the Resource Manager (RM), keeps track of the computing resources – either embedded on the device or on remote nodes – currently available. Upon the detection of a change in the resource pool, the RM notifies all the other components so they react to the change. If dynamic resource provisioning is enabled, this component should also adapt dynamically the reserved resources to the current workload. The second component is the Task Scheduler (TS), which picks the resources and time lapse to host the execution of each tasks while meeting dependencies among them and guaranteeing the exclusivity of the assigned resources. For that purpose, it keeps track of the available resources in each computing device and a statistical analysis of previous executions of each function. The default policy pursues exploiting the data locality; it assigns new ready (with no pending dependencies) tasks to the idle resource having more data values on its local storage; if all the resources are busy, it will pick the ready task with more data values on the node at the moment of the resources release. However, TS policies are designed in a plug-in style to allow the extension of the runtime with application/infrastructure-specific policies; the policy used by the TS is selected at agent boot time. The Data Manager (DM), the third piece of the runtime, establishes a data sharing mechanism across the whole infrastructure to fetch the necessary input data to run the task locally and publish the results. For guaranteeing that the involved data values remain available even on network disruption situations, the DM leverages on distributed persistent storage solutions like dataClay [28]. The last component, the Execution Engine (EE), handles the execution of tasks on the resources. When the TS decides to offload a task to a remote agent, the EE forwards the function execution request to the API of the remote agent. Conversely, if the TS determines that the local computing devices will host the execution of a task, the EE fetches from the DM all the necessary data missing in the node, launches the execution according to its description and the assigned resources, and publishes the task results on the DM. It is during the task execution when the task-based programming models take the scene and convert the logic of the method into a workflow. When the selected PM detects a task, instead of invoking the corresponding runtime, it refers to the TS of the local Agent to create a new task indicating the operation, the involved data and the dependencies that the new task has with previously submitted tasks. The Colony runtime will handle the execution of such task in the same manner as if it were an external request and the TS will guarantee that dependencies with previous tasks are considered. Thus, the expressiveness limitations to convert a function into a workflow depend on the specific programming model selection rather of being a limitation of Colony on its own. Figure 1 depicts the control flow followed by a function throughout the runtime components when the TS decides to host the execution locally (leftmost part of the figure) and to offload the execution onto a remote agent (rightmost part of the figure).

To control the pool of resources, the Agent API offers three methods which notify the desired change to the RM. To increase the pool with more resources, the Agent API offers a method to indicate the name of the new node and its computing capabilities. If the node is already a part of the pool, the agent expands it with the provided resources. To remove resources from the pool, the API offers two methods. The immediate removal method aims to support the disconnection of a remote device, the agent assumes that all the non-completed tasks offloaded onto the node have failed and resubmits them onto other nodes of the infrastructure. On the contrary, the gentle removal method backs the case when the system administrator wants to reduce the usage of a remote node. In this case, the agent does not submit more tasks to the to-be-removed device until there are enough idle resources to satisfy the removal request; then, it releases the resources and continues to use the remaining ones, if any.



(a) Execution on local resources



(b) Execution offloaded onto remote agent

Fig. 1: Function/Task flow throughout the Colony's runtime components

4 Colony Organization

For infrastructures with a small number of devices, a single Colony agent can individually orchestrate the execution of the tasks on any of the other agents composing the infrastructure. However, the bigger the infrastructure is, the higher the complexity of the scheduling problem becomes and its computational burden is more likely to grow bigger than the actual computational load of the application and simply not fit in resource-scarce devices.

To overcome such problem, this article proposes to reduce the amount of scheduling options by gathering resources under the concept of a colony. Colonies are disjoint groups of agents that share their workload; one of the colony members acts as the interaction gateway and orchestrates the workload execution within the colony. Thus, agents no longer consider all the other agents to run the task; it only picks a gateway agent to whom delegate the problem. This gateway agent organizes the agents within the colony in sub-colonies and considers the rest of the infrastructure as a new colony where the first agent acts as the gateway. Thus, a computation triggered by an agent could run on any node of the infrastructure without adding a significant overhead due to the scheduling.

Combining this serverless capability of Colony with the ability to decompose the logic of the computation into several tasks allows any request to use as many resources as needed. If the tablet at home colony of the example illustrated in Figure 2 requires some computation, its local agent could use the embedded processor to host the function execution. After converting it into a workflow, it offloads part of the execution of its inner tasks onto the agent on the fog-enabled – with compute capabilities – Wi-Fi router. In turn, the Wi-Fi router agent decides which part of the computation hosts, which part it offloads onto the laptop, and which part submits to the remote parts of the platform – through the agent on a cluster server. As with the previous agent, the server can host part of the execution, forward it to the other nodes of the cluster or send it to the smartphone or the server in the office. Regardless the device being the source of computation requests, the computing load can be distributed across

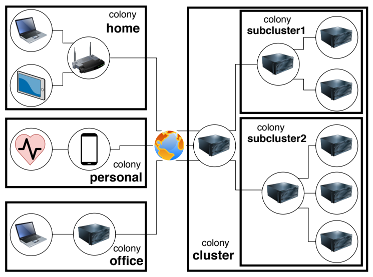


Fig. 2: Colony deployment example connecting devices from four different environments: those at user's home (laptop, tablet, and other devices connected to the Wi-Fi), those at user's office (the laptop and an on-premise server), those moving with the user (smartphone and wearables) and a remote cluster with large computing power. Each of these environments becomes a colony hierarchically organized and the gateway in the cluster acts as nexus among colonies.

the entire infrastructure and thus executed by any node. The decisions taken by each agent will depend on the scheduling policy configured by the device owner.

Mobile devices may join or leave the infrastructure unexpectedly. When a device – e.g., a smartphone –, which might be part of a colony – the *personal* colony of the example –, joins the infrastructure, it attaches to an already-member device – for instance, the Wi-Fi router of the *home* colony. From the smartphone point of view, the whole infrastructure becomes a colony where to offload tasks through the agent on the router; for those devices already part of the infrastructure, the incoming device/colony becomes a new subcolony also available through the router. Likewise, when a device disconnects, the communication link between two agents breaks and they both lose the corresponding colonies. To ensure that ongoing computations finish, the involved agents need to re-schedule the execution of the unfinished tasks already offloaded onto the lost colony and assign them to still-available agents.

Currently, the platform topology is manually set up by the platform administrator and can be changed dynamically using the resource management operations of the Agent API. Good criteria to build the topology are the stability and latency of the network; this ensures that agents will always try to offload tasks onto nearby resources, on the fog, rather than submitting them to the Cloud achieving a higher performance. Besides, the system remains usable even when the cloud is unavailable because of network disruptions.

5 Usage Scenarios and Evaluation

This section presents the results of the tests conducted to evaluate the viability of the described proposal. For that purpose, a prototype leveraging on the COMPSs programming model – whose runtime has been adapted to delegate the task execution onto Colony – has been developed. The implemented tests are based on two use cases aiming two different platforms. The first use case, a service running a classification model, aims to test the whole Cloud-Edge Continuum. The training of the model (batch processing scenario) runs on the Cloud part and allows to evaluate the performance and scalability of the solution; by submitting simultaneous user requests simulating readings from multiple sensors (sense-process-actuate scenario), we evaluate the workload balancing. The second use case, a real-time video analysis, demonstrates the streaming support within Fog environment without support from the Cloud.

5.1 Baseline Technology - COMP Superscalar (COMPSs)

COMP Superscalar (COMPSs) is a framework to ease the development of parallel, distributed applications. The core of the framework is its programming model with which developers write their code in a sequential, infrastructure-unaware manner. At execution time, a runtime system decomposes the application into computing units (tasks) and orchestrates their executions on a distributed platform guaranteeing the sequential consistency of the code.

COMPSs' main characteristic is that developers code applications using plain Java/Python/C++ as if the code was to be run on a single-core computer. For the runtime system to detect the inherent tasks, application developers need to select, using annotations or decorators depending on the language, a set of methods whose invocations become tasks to be executed asynchronously and, potentially, in an out-of-order manner.

To guarantee the sequential consistency of the code, the runtime system monitors the data values accessed by each task – the arguments, callee object and results of the method invocation – to find dependencies among them. For the runtime to better-exploit the application parallelism, developers need to describe how the method operates (reads, generates or updates) on each data value by indicating its directionality (IN, OUT, INOUT, respectively). Such data values can be either files, primitive types, objects or even streams.

Upon the arrival of a new task execution request, the COMPSs runtime analyses which data values are being accessed and which operations the task performs on them. With that information, it detects the dependencies with other tasks accessing the same values and constructs a task dependency graph. Once all the accesses of the task have been registered, the runtime schedules its execution on the available resources guaranteeing the sequential consistency of the original code. Thus, COMPSs is able to convert the logic of any function into an hybrid task-based workflow - data-flow and support the three compute scenarios identified in Section 1.

Conceptually, COMPSs tasks and Colony function execution requests are similar; both refer to serverless, stateless executions; Thus, native support for the COMPSs programming model within Colony is straightforward modifying the COMPSs runtime system to delegate task executions onto the Colony runtime. Upon the detection of a task and the registration of all the data accesses to find dependencies with previous tasks, the COMPSs runtime calls the Colony runtime to handle the execution of the tasks on the underlying platform as if it were a regular invocation of a function. Colony's TaskScheduler runtime will consider the detected dependencies and guarantee the sequential consistency of the tasks and, therefore, of the user code following the COMPSs model.

5.2 Classification Service

This first use case deploys a classification service on the Cloud-Edge continuum. On the one hand, the training of such model allows us to validate the system running compute-heavy applications on a batch-processing scenario; usually, the service administrator or a periodic task triggers the training of the model on the Cloud. On the other hand, having nodes on the infrastructure acting as sensors allows us to validate the behaviour of the system on sense-process-actuate scenarios. These sensors submit their lectures to classify them and react to it.

The algorithm supporting the model, RandomForest, constructs a set of individual decision-trees, also known as estimators, each classifying a given input into classes based on decisions taken in a random order. The final classification of the model is the aggregate of the classification of all the estimators; thus, the accuracy of the model depends on the number of estimators composing it. The training of the estimators are independent from each other, each one consisting of two tasks: a first one that selects a combination of 30,000 random samples from the training set, and a second one that builds the decision tree.

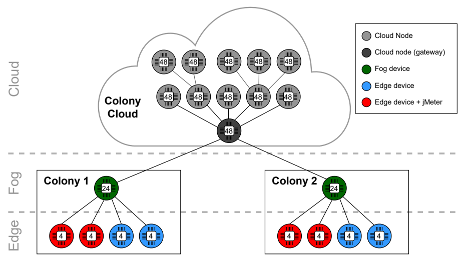
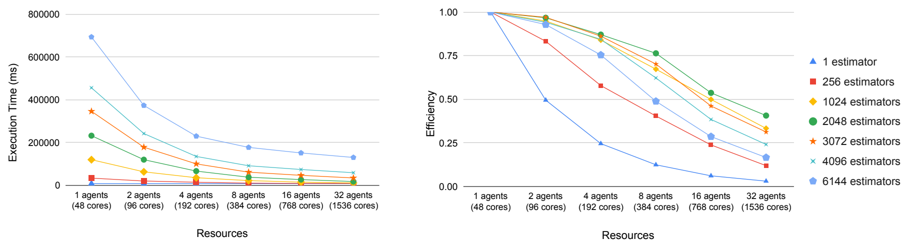


Fig. 3: Testbed for the sense-process-actuate experiment

To conduct the experiment, we deployed a 3-level infrastructure – depicted in Figure 3 – on Marenostrum, a 3,456-node (48 servers of 72 nodes) supercomputer where each node has two 24-core Intel Xeon Platinum 8160 and 98 GB of main memory. Each node hosts the execution of an agent managing its 48 cores. All the agents within the same server join together as a colony and one of them acts as the gateway; in turn, one of these gateway nodes becomes the interconnection point among all the server colonies (darker gray on the figure). Two of these colonies are configured to act as the Fog parts of the deployment: one resource-rich node with 24 CPU cores (green) and four Edge nodes with 4 CPUs each. While four edge devices (depicted in blue) play a passive role and only provide computing power to their respective colony, two edge devices of each colony (depicted in red) act as sensors submitting compute requests to its local agent. To produce the workload, these nodes run an instance of jMeter [5].



(a) Varying the number of agents.

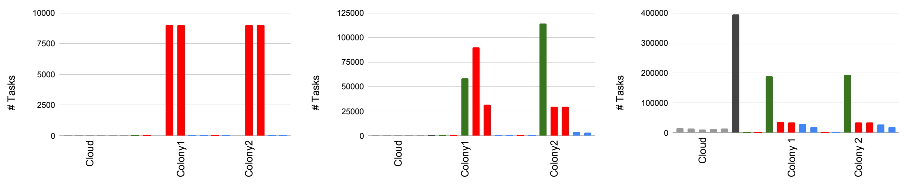
(b) Efficiency (speedup/#agents).

Fig. 4: RandomForest's training time

Charts in Figure 4 depict the results of the scalability tests for the training of the model varying to the number of agents on the Cloud. Figure 4a illustrates the evolution of the training time when changing the number of resources to train a fixed-size model demonstrating the benefits of parallelization: the larger the infrastructure grows, the shorter the execution time becomes. Figure 4b presents the efficiency (ratio between the speedup compared to the 1-agent execution of the same-size problem and the size of the infrastructure) of the execution revealing some scalability problems. Up to 2048 estimators, the performance loss can be explained mainly by the load imbalance. The larger the infrastructure is, the more resources remain idle waiting for others to complete the training. A second cause is the overhead of COMPSs when detecting new tasks. COMPSs sequentially detects tasks as the main code runs; the task creation delays build up. Training one estimator requires about 7 seconds (running both tasks). To keep the whole infrastructure busy on the 1,536 cores scenario, COMPSs must generate a task every 2.25 ms. The granularity of the tasks is too fine given the size of the infrastructure.

Beyond 2048 estimators, the performance loss is explained by the implementation of the Task Scheduler; it has a single thread that handles in a FIFO basis both new task requests and end of task notifications. Hence, several end of task notifications may stack up before a new task request leaving several nodes idle waiting for a new task. Likewise, many new task request might accumulate in front of the task end notification; thus, despite the resources are idle, the scheduler is not aware of that and does not offload more work to the node.

For the second experiment, regarding the processing submitted by the sensors, the cloud part of the infrastructure has 6 nodes on the same server. Figures 5a, 5b and 5c depict the evolution of the number of requests being handled by each device when 10, 100 and 300 users submit requests at an average pace of 1 request per second (with a random deviation of ± 500 ms following Gaussian distribution) during 15 minutes.



(a) 10 users (b) 100 users (c) 300 users

Fig. 5: Number of tasks executed on each node. The colors of the bars match the color from the respective node as depicted in Figure 3.

On the 10-user case, the Edge nodes receiving the request are able to host all the workload; only 3 requests out of 36,000 were offloaded to the Fog nodes on peak-load moments. The average response time was 104 ms. On the 100-user case, the nodes cannot assume the whole workload and offload onto their respective Fog colony which handles most of the requests. On Colony1, one of the edge nodes receives the requests at a periodic pace; thus, it is able to assume the whole workload (90,000 requests) by itself. On the other edge node, the requests arrive in bursts, and the scheduler decides to offload tasks to the Fog node (58,432). On Colony2, both edge nodes also receive the request in bursts, and both offload tasks to the Fog node which actually processes up to 116,041 requests. This Fog node cannot assume the whole workload and decides to offload part of it to the idle edge nodes (3,738 and 3,142 requests, respectively). The average response time is also 104 ms. Finally, on the last case, with 300 users, both colonies cannot assume the workload locally and a large portion of the requests (464,999/1,080,000) are offloaded onto the Cloud. The average response time slightly increases to 109 ms.

5.3 Real-time Video Processing

On the real-time video processing use case, the application obtains a video-stream directly from a camera, processes each of the frames to identify the people in it, and maintains an updated report with stats of their possible identifications.

Figure 6 depicts the workflow of the use case deployed on a 2-device testbed composed of a Raspberry Pi (rPi) equipped with a camera and a laptop. The *watch* task is a persistent task that obtains the input from a webcam and publishes frames onto a stream. By defining constraints for the task, the agent ensures that the task runs on a camera-enabled device (rPi). For each frame, a *detectPeople* task finds the areas in the picture containing a person using a pre-trained Caffe implementation of Google's MobileNet Single-Shot Detector DNN [1]. Each detection triggers the execution of an *IdentifyPerson* task. *IdentifyPerson* detects the face of the person – using a DNN provided by OpenCV [18]

[1] https://github.com/chuanqi305/MobileNet-SSD

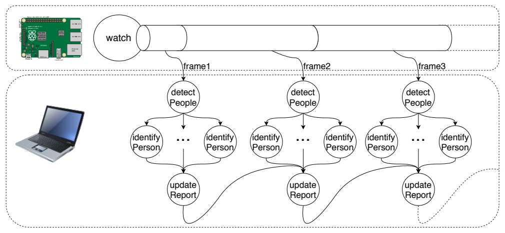


Fig. 6: Testbed and workflow of the video-processing use case

–, aligns the image by recognizing 68 landmarks on the face, and extracts 128 features – using OpenFace's [16] nn4.small2 DNN. Finally, a SVM classification [2] – identifies the person. Finally, the *update_report* task relates the people detected on the current frame with people appearing on the previous ones – tracking – and the individual information is aggregated into a *Report* object that can be queried in parallel.

The average time to process a frame containing one person on the rPi is 41,293 ms – *detectPeople* lasts 18,920 ms and *IdentifyPerson*, 22,339 ms – providing a framerate of 0.024 fps. As baseline for the comparison on distributed systems, we implemented the service as two different applications communicating through a TCP Socket. The application running on the rPi obtains the images from the webcam, serializes the frame and ships it through the socket; thus, the rPi produces a frame every 453 ms. The second application, running on the laptop, reads from the socket and processes the frame (132 ms). Thus, hand-tuned code is able to process 2.14 fps. Finally, we conducted the test running the service taskified with COMPSs obtaining a processing rate of 2.79 fps. The analysis of the time elapsed on each part of the application revealed that the performance difference between both versions lies in the frame serialization, which shrinks from 120 ms to 20 ms. Both tests run with the same JVM and they use the same code to perform the operation; therefore, we attribute this difference to the JVM internal behavior – probably, JNI verifications – or to OS memory management.

6 Related Work

This papers aims to bring together Function-as-a-Service (FaaS) with the Computing Continuum. Regarding FaaS, Zimki was the first framework to offer serverless computing in 2005; in 2014, Amazon introduced Lambda [3], becoming the first large cloud service provider to offer FaaS, followed by Google and Microsoft that respectively launched Cloud Functions [8] and Azure Functions [6]. In 2016, IBM announced an open-source FaaS project, OpenWhisk [13], that allowed FaaS deployments on private clouds; Microsoft adapted its solution to support the execution on on-premise cloud. OpenWhisk broke the vendor lock-in by supporting the execution on container managers and IBM branched the

[2] model trained with 10,000 images of 125 people from CASIA-WebFace facial dataset

project to release Cloud Functions [10]. Two other open-source alternatives to OpenWhisk are OpenFaaS [12], supported by VMWare, and Fn[14], backed by Oracle.

For economic reasons – it is cheaper to provision the resources directly as cloud instances –, all major FaaS solutions limit the execution time for each function inhibiting long-lasting (batch-processing) computations. To develop such computations, cloud vendors usually offer programming solutions perfectly integrated in their platforms; otherwise, developers manually set up the cluster to use a myriad of distributed programming models. For instance, to develop workflows and scientific computing, they can turn to Swift [34], COMPSs [27], Kepler [15], Taverna [23] or Pegasus [21]; on the data analytics field, MapReduce [20] and other solutions building on it – such as Twister [22] or Apache Spark [35]. Likewise, major cloud providers also offer their own stream-processing alternatives such as Amazon Data Pipeline [1]. However, developers frequently use the stream-oriented low-level frameworks, such as Apache Kafka [25], or dataflow models like Apache Storm [32], Apache Spark-Streaming [36] or Heron [26]. Apache Beam [4], COMPSs [30] and Twister2 [24] have gone a step further aiming to merge both workflows and dataflows in one single solution.

Towards exploiting the Compute Continuum, all the solutions from the major cloud vendors (Amazon IoT Greengrass [2], Microsoft Azure IOT Edge [7] and Google Cloud IOT [9]) maintain the cloud as a necessary part but allow the manual deployment of function executions on Edge devices. The Osmosis framework [33] also follows this top-down approach. The developer defines microElements (MELs) and describes how these MELs relate to each other. From the cloud, Osmosis orchestrates MEL deployments and migrations taking into account resource availability, each MEL's QoS and the infrastructure topology.

As of today, for all major Cloud providers the Edge is totally reliant on the Cloud; it should become autonomous and function even disconnected from the Cloud, and, in that direction, a lot of research is being done. Selimi et al. [31] attack the problem from a bottom-up approach and propose a framework for placing cloud services in Community Networks. Ramachandran et al. identified the challenges to provide a peer-to-peer standing for the Edge to the Cloud [29]. Departing from a real-world case of study, Beckman et al. [17] diagnose the technical shortcomings to implement a solution; they consider Twister2 for data analytics and indicate the necessary implementations.

The novelty of the work presented in this paper is to bring together programmability, FaaS and Compute Continuum. Unlike other frameworks targetting the Continuum, Colony offers a FaaS approach to submit computations. Compared to FaaS solutions on the Cloud, able to delegate tasks on Edge devices, Colony allows to automatically convert the logic of the function into a workflow; thus, being able to parallelize and distribute its execution to achieve lower response times and better infrastructure exploitation. By leveraging on the COMPSs programming model, the overall solution is able to tackle the three described computing scenarios: batch processing, stream-processing and sense-process-actuate.

7 Conclusion and Future Work

This manuscript introduces Colony: a framework to develop applications running throughout the whole Cloud-Edge Continuum. This framework proposes a hierarchic organization of the computational resources building on the concept of an Agent: an autonomous process running on each device that allows executing software in a Function-as-a-Service manner. By automatically transforming these functions into task-based workflows, an Agent is able not only to parallelize the execution; it can also distribute the workload offloading part of the execution onto other Agents. Thus, the workload is balanced across the whole platform while taking advantage of the low-latency network interconnecting nearby resources. COMPSs is a task-based programming model that supports not only atomic tasks, but also persistent tasks producing/consuming a stream of data. By natively supporting COMPSs, Colony offers a common programming interface to deal with the three computing patterns necessary on Cloud-Edge services.

Section 5 suggests some shortcomings of the proposal. The current prototype only allows the detection of tasks on the main function, enabling the detection during the execution of any task would allow to parallelize the task generation, and thus, improve the application performance. The default scheduling policy problem aims to keep the resources busy unnecessarily offloading tasks as shown on the 100-user test on Section 5.2. Other scheduling policies – maybe based on QoS and SLA with time-constrains – would distribute the workload differently.

Other Cloud-Edge-related issues remain open for further investigation, for instance, the automatic resource discovery and configuration of the resources. Regarding data, privacy/IP-sensitive data should never abandon the device or on-premise resources, extending the model with data-access/movement control policies is also a future research line. As Section 3 explains, Colony delegates the data management; significant infrastructure divisions may entail misbehaviours in the used solution and some data values become unavailable. We aim to enable a fault-tolerance mechanism – e.g., lineage – to recompute missing values.

Automatically triggering executions based on events related to stored data or external webservices (Eventing) is supported by most of the framework named in Section 6. Colony does not include such component; applications need to run a persistent task monitoring a state/value/stream that triggers the computation when certain condition is met or reliy on external services like IFTTT [11].

Finally, as discussed in the related work section, large cloud providers enforce users to follow their own software. To overcome this vendor lock-in while using their cloud platforms, Colony must use their IaaS solution and deploy an agent there. Another line of research to delve into is Interoperability; implementing software to let Colony offload tasks onto these large clouds through their FaaS software would entail a significant improvement in terms of cost efficiency.

Acknowledgements

This work has been supported by the Spanish Government (PID2019-107255GB), by Generalitat de Catalunya (contract 2014-SGR-1051), and by the European

Commission through the Horizon 2020 Research and Innovation program under Grant Agreement No. 101016577 (AI-SPRINT project).

References

1. Amazon Data Pipeline. https://aws.amazon.com/datapipeline/
2. Amazon Greengrass. https://aws.amazon.com/greengrass/
3. Amazon Lambda. https://aws.amazon.com/lambda/
4. Apache Beam. https://beam.apache.org/
5. Apache JMeter. https://jmeter.apache.org/
6. Azure Functions. https://azure.microsoft.com/services/functions/
7. Azure IoT-Edge. https://azure.microsoft.com/en-us/services/iot-edge/
8. Google Cloud Functions. https://cloud.google.com/functions
9. Google IoT Cloud. https://cloud.google.com/solutions/iot/
10. IBM Cloud Functions. https://www.ibm.com/cloud/functions
11. IFTTT. https://ifttt.com/
12. OpenFaas. https://www.openfaas.com/
13. OpenWhisk. https://openwhisk.apache.org/
14. The Fn project. https://fnproject.io/
15. Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., Mock, S.: Kepler: an extensible system for design and execution of scientific workflows. In: Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004. pp. 423–424. IEEE (2004)
16. Amos, B., Ludwiczuk, B., Satyanarayanan, M.: Openface: A general-purpose face recognition library with mobile applications. Tech. rep., CMU-CS-16-118, CMU School of Computer Science (2016)
17. Beckman, P., Dongarra, J., Ferrier, N., Fox, G., Moore, T., Reed, D., Beck, M.: Harnessing the computing continuum for programming our world. Fog Computing: Theory and Practice pp. 215–230 (2020)
18. Bradski, G.: The OpenCV Library. Dr. Dobb's Journal of Software Tools (2000)
19. Consortium, O., et al.: Openfog reference architecture for fog computing. Architecture Working Group pp. 1–162 (2017)
20. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6. p. 10. OSDI'04, USENIX Association, Berkeley, CA, USA (2004), http://dl.acm.org/citation.cfm?id=1251254.1251264
21. Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P.J., Mayani, R., Chen, W., Ferreira Da Silva, R., Livny, M., Wenger, K.: Pegasus, a workflow management system for science automation. Future Generation Computer Systems **46**, 17–35 (2015). https://doi.org/10.1016/j.future.2014.10.008
22. Gunarathne, T., Zhang, B., Wu, T.L., Qiu, J.: Scalable parallel computing on clouds using Twister4Azure iterative MapReduce. Future Generation Computer Systems **29**(4), 1035–1048 (2013). https://doi.org/10.1016/j.future.2012.05.027, http://www.sciencedirect.com/science/article/pii/S0167739X12001379
23. Hull, D., Wolstencroft, K., Stevens, R., Goble, C., Pocock, M.R., Li, P., Oinn, T.: Taverna: A tool for building and running workflows of services. Nucleic Acids Research (2006). https://doi.org/10.1093/nar/gkl320

24. Kamburugamuve, S., Govindarajan, K., Wickramasinghe, P., Abeykoon, V., Fox, G.: Twister2: Design of a big data toolkit. Concurrency and Computation: Practice and Experience **32**(3), e5189 (2020). https://doi.org/10.1002/cpe.5189, https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5189, e5189 cpe.5189
25. Kreps, J., Narkhede, N., Rao, J.: Kafka: a Distributed Messaging System for Log Processing. ACM SIGMOD Workshop on Networking Meets Databases (2011)
26. Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J.M., Ramasamy, K., Taneja, S.: Twitter heron: Stream processing at scale. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. vol. 2015-May, pp. 239–250 (2015). https://doi.org/10.1145/2723372.2723374
27. Lordan, F., Tejedor, E., Ejarque, J., Rafanell, R., Álvarez, J., Marozzo, F., Lezzi, D., Sirvent, R., Talia, D., Badia, R.M.: ServiceSs: An Interoperable Programming Framework for the Cloud. Journal of Grid Computing **12**(1), 67–91 (2014). https://doi.org/10.1007/s10723-013-9272-5, http://dx.doi.org/10.1007/s10723-013-9272-5
28. Martí, J., Queralt, A., Gasull, D., Barceló, A., José Costa, J., Cortes, T.: Dataclay: A distributed data store for effective inter-player data sharing. Journal of Systems and Software (2017). https://doi.org/10.1016/j.jss.2017.05.080
29. Ramachandran, U., Gupta, H., Hall, A., Saurez, E., Xu, Z.: Elevating the edge to be a peer of the cloud. In: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD). pp. 17–24. IEEE (2019)
30. Ramon-Cortes, C., Lordan, F., Ejarque, J., Badia, R.M.: A programming model for Hybrid Workflows: Combining task-based workflows and dataflows all-in-one. Future Generation Computer Systems (2020). https://doi.org/10.1016/j.future.2020.07.007
31. Selimi, M., Cerdà-Alabern, L., Freitag, F., Veiga, L., Sathiaseelan, A., Crowcroft, J.: A Lightweight Service Placement Approach for Community Network Micro-Clouds. Journal of Grid Computing (2019). https://doi.org/10.1007/s10723-018-9437-3
32. Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J.M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S., Ryaboy, D.: Storm @Twitter. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. pp. 147–156 (2014). https://doi.org/10.1145/2588555.2595641
33. Villari, M., Fazio, M., Dustdar, S., Rana, O., Jha, D.N., Ranjan, R.: Osmosis: The osmotic computing platform for microelements in the cloud, edge, and internet of things. Computer **52**(8), 14–26 (2019)
34. Wilde, M., Hategan, M., Wozniak, J.M., Clifford, B., Katz, D.S., Foster, I.: Swift: A Language for Distributed Parallel Scripting. Parallel Comput. **37**(9), 633–652 (2011). https://doi.org/10.1016/j.parco.2011.05.005, http://dx.doi.org/10.1016/j.parco.2011.05.005
35. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark : Cluster Computing with Working Sets. HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing (2010). https://doi.org/10.1007/s00256-009-0861-0
36. Zaharia, M., Das, T., Li, H., Shenker, S., Stoica, I.: Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters (2012)