OpenEdgeComputeFramework. A Framework for Seamless Edge-Cloud Computing

Andrei-Robert CAZACU
Bucharest University of Economic Studies, Romania
andrei.cazacu@csie.ase.ro

With the advent of Internet of Things and the success of social media, the amount of data that is being sent to the cloud is ever-increasing. As such, finding possible solutions to alleviate the strain that is currently placed on our infrastructure is a hot topic both in the researcher's community as well as the enterprise solution vendors in pursuit of the next big breakthrough in computing. One such solution is edge-computing, first making its appearance in the 1990s when Akamai launched its content delivery network [1], *moving the content source closer to the user to minimize latency and transmission costs. Today, this concept evolved into a computing paradigm* [2], *which supports deploying applications on nearby devices with near zero downtime. Among the emerging frameworks, be them open source such as EdgeX framework, research originating such as FogBus2, or enterprise solution such AWS GreenGrass, several common architectural patterns were identified such as the heavy use of Containerization, decoupling of components by using message passing interfaces and network segregation among edge and cloud devices. The proposed solution aims to build on top of existing knowledge, by using lightweight peer-to-peer orchestration aimed at running centrally stored Java artifacts.*
Keywords: *Edge computing, Cloud computing, Distributed computing*
DOI: 10.24818/issn14531305/28.4.2024.06

1 Introduction

According to Cisco's VNI Complete Forecast Highlights [3], the average traffic per capita per month experienced an almost 300% increase from 2016 to 2021, growing from 12.9GB to 35.5GB. This sustained growth along with the shift from desktop to mobile devices prompted the need for a more flexible computing model, one that would bring the resources closer to the user.

In the past, a resource would simply represent a static asset: a HTML page or an image. For this use-case, Akamai revolutionized the internet with the introduction of the content delivery network (CDNs). Fast forward to today, and a resource has a vastly different meaning, ranging from a stream that needs to be decoded just-in-time, to user produced media that needs to be processed by cloud services. As such, the strain placed on the network is ever increasing, with efforts to increase the available bandwidth already underway [4] [5]. Since these are finite resources, we are bound to hit a breaking point where adding more resources to support the growing scale is not going to make economic sense. As such, researchers [6] are alluding to an evolutionary step, an extension of our current cloud model meant to split the workload among multiple devices closer to the user.

Also, the current cloud computing paradigm is plagued by several shortcomings such as:

- Potential of increased latency due to distance between request originator and cloud;
- High environmental impact due to non-optimal resource utilization;
- High risk of data leak due to transmission of sensitive data.

To overcome these issues, a significant amount of resources are poured into researching and developing edge computing solutions, with proven applications in artificial intelligence [7], e-commerce [8], and many others.

To accommodate the largest range of devices, a middle ground between edge computing and cloud computing exists called fog computing [9], in which minimal the data is minimally processed at the edge layer with aggregations and complex business processes happening in an intermediate layer, the fog layer, before eventually shipping it to the cloud layer for

terminal processing and long term storage.
Several solutions are emerging in the edge-fog computing space, among those noting EdgeX Foundry of the Linux Foundation [10], AWS Greengrass [11], or FogBus2 [12], enabling on-device compute capabilities by leveraging containerization, thus achieving platform agnosticism and a high level of flexibility in the amount of applications that can be ran.
On a high level, these solutions have a great degree of commonality, devices being softly segmented into 3 layers:

- End-node
  - Device responsible with producing or consuming data
- Edge layer
  - Primary computing layer
  - Able to offload any amount of the computation to the cloud
  - Can also be an end-node
- Cloud (only FogBus2)
  - Resource rich computing layer

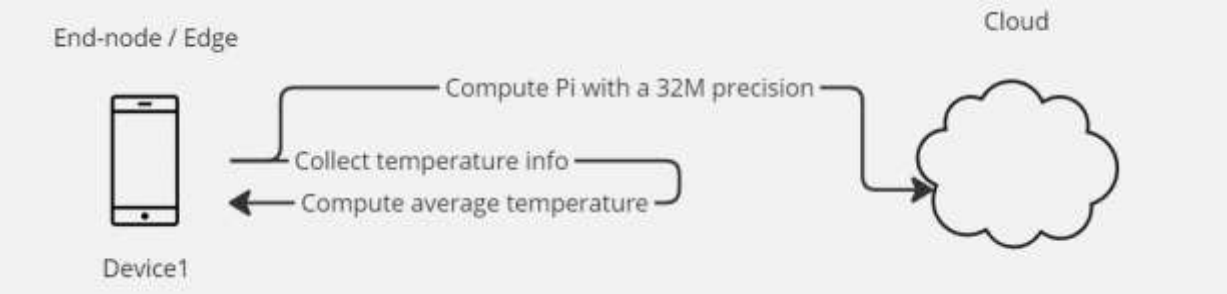

Fig. 1. Edge-Cloud computing architecture

As it can be seen in Figure 1, all these layers work in a symbiotic manner, the edge layer being best suited for privacy or latency sensitive operations, while the cloud is better used for compute-heavy workloads or long-term storage of data.
Although providing a flexible deployment model by using containerization, the above frameworks are limited to being run on fully-fledged machines running General Purpose Operating Systems such as Linux since only those are currently supporting container runtimes. As such, their applicability on lower-powered devices such as IP Cameras, Smart Fridges and IoT nodes is severely impacted.
To overcome the above limitation, this paper proposes a transparent hybrid cloud computing framework which uses peer to peer orchestration to transparently schedule workloads on either edge or cloud. The platform is language agnostic by exposing a simple interface for tasks, but for brevity we'll assume that the task is implemented using Java.

2 OpenEdgeComputeFramework

OpenEdgeComputeFramework aims to bring computing closer to the user by transparently handling any task scheduling in a fair and transparent manner among edge and cloud devices. This approach should result in a lower network bandwidth consumption by performing a significant chunk of computation on-device, leveraging the increase in CPU power of ordinary devices.
To bring some numbers to the equation, an iPhone 15 Pro has a single thread score in the GeekBench 6 benchmark comparable to a desktop 14th Generation Intel core i7 processor [13], which, taking into consideration the fact that a large amount of workloads are purely single-threaded, should result in a large amount of computations able to be performed on-device.
The proposed architecture consists of two layers, edge, and cloud devices, where each node has an identical stack of components and task orchestration is performed in a distributed peer-to-peer manner. The framework consists of several components, each having fundamentally a single responsibility: orchestration, task execution, telemetry collection, and remote logging.

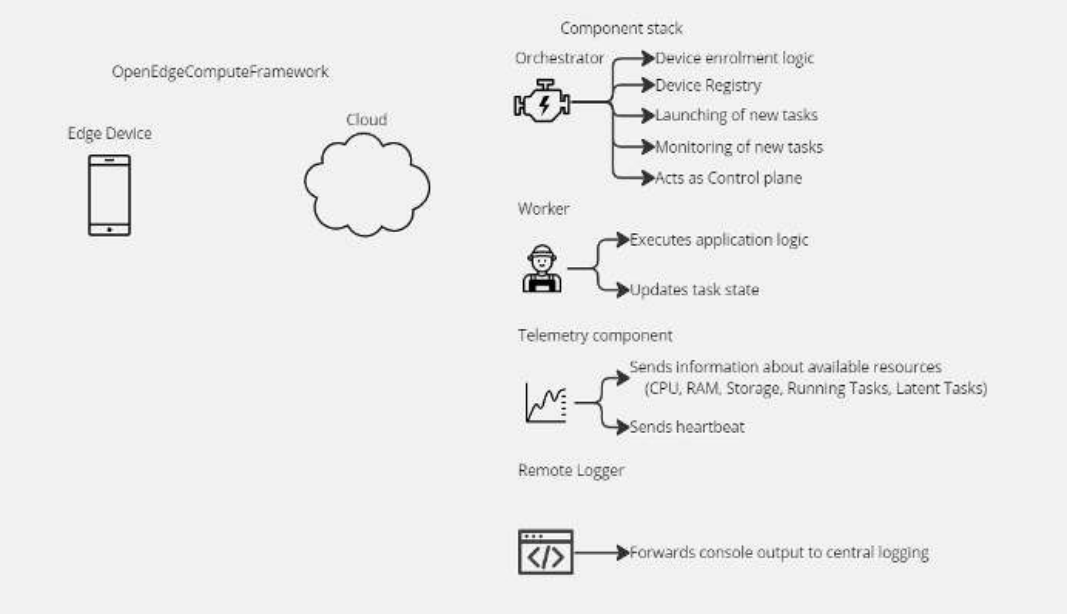

Fig. 2. Proposed architecture

As per Figure 2, the components of the framework are the Orchestrator, the Worker, the Telemetry Component, and the Remote Logger component, going over the functionality of each component one by one.

2.1 Orchestrator

The orchestrator is the most important part of the framework, being responsible for bringing the device to a ready state, storing the device registry, while also scheduling and monitoring tasks.

It acts a control plane, transparently distributing workloads among nodes available in the compute pool based on key metrics such as available CPU, RAM, and flash storage, while also taking into consideration the processing power of the device and the latency.

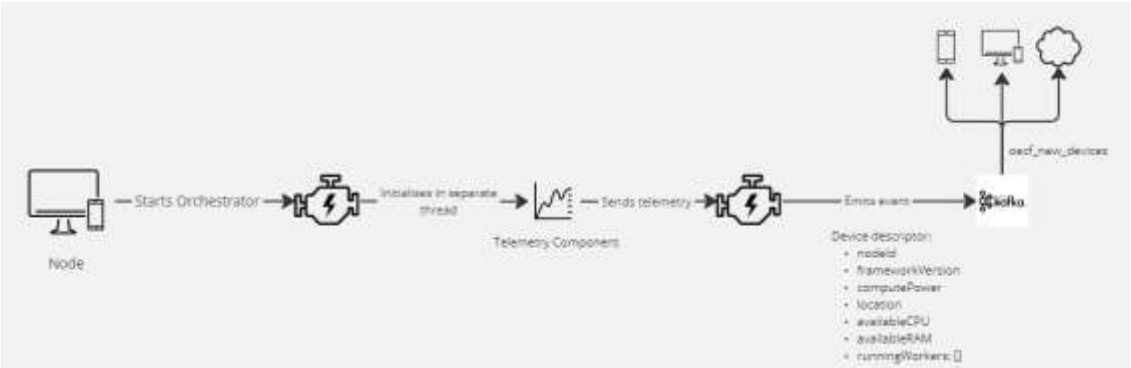

Fig. 3. Device Join Flow

As seen in Figure 3, the role of the orchestrator starts with the device enrolment, which is performed by publishing a JOIN event on a well-known Kafka Topic (*oecf_new_devices*) with the device descriptor and telemetry information. This descriptor contains information relevant for computing the device score, a metric used to determine the best available node for scheduling tasks.

Once the event has been emitted on the *oecf_new_devices* topic, the orchestrator will create and subscribe to the topic *oecf_{nodeId}_enrolled_devices*, a standard framework topic used for one-time population of the device registry with already pooled resources. The payload follows the same format as the new devices' topic, thus maintaining a uniform interface between devices.

From the existing device perspective, the flow is reversed; the node is subscribed to the *oecf_new_devices* topic and used it to enlarge the device registry. Once the event is received, it will publish its own device descriptor in the enrolled devices topic.

Once device information is exchange among parties, the process continues by computing the device score:

$$score_{device} = computePower * (100 - cpu_{used}) * balancingFactor * proximityBonus * taskRunningBonus$$

Legend:

computePower – *computed at orchestrator startup by running performance benchmark*

balancingFactor – *mean compute power; assures fair distribution of work by balancing outliers*

proximityBonus – *value boosting the score of devices in close-proximity; subject to online machine learning for balancing the factor based on network load*

taskRunningBonus – *booster for preferring nodes which already have a worker node of that type spun up*

This device score is constantly recomputed by the orchestrator once any equation factor changes and is the metric used in determining the suitable worker node.

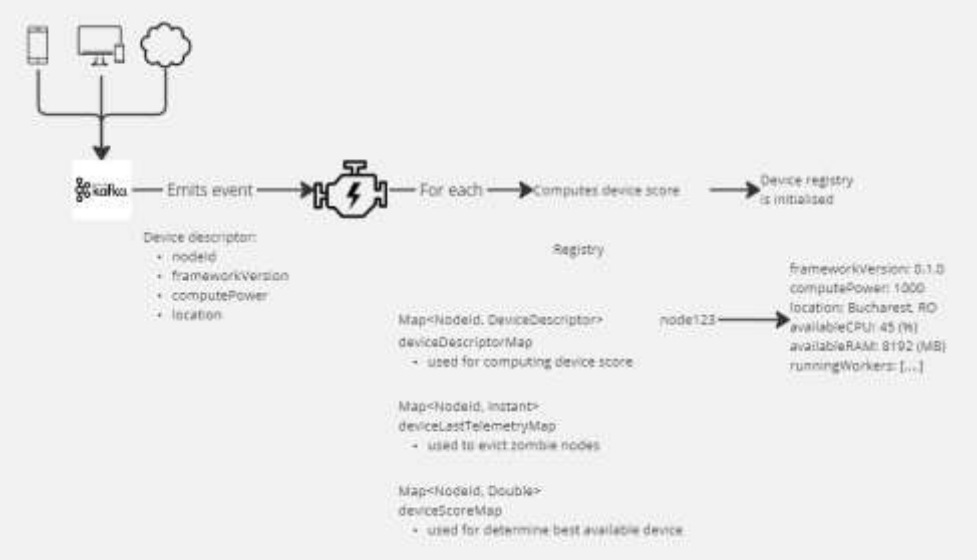

Fig. 4. Initialization of device registry

Having performed the steps outlined in Figure 4, the orchestrator has completed the initialization logic and is ready to compute workloads.

In this state, the device registry is fully initialized, and the orchestrator is subscribed to other parties' telemetry topic used for real-time updates of the device score. Additionally, this sub-component is responsible for purging zombie devices, removing any device from the pool which didn't send telemetry information for more than a configurable threshold value.

The orchestration mechanism is peer to peer, with the orchestrator being responsible for transparently scheduling tasks either on the edge device or remote machines based on the pre-computed device score.

Tasks are created using a COAP POST call exposed by the orchestrator, */createTask*, which expects the task type to be mentioned. Behind the scenes, this API call will start the following orchestration logic:

- By using the pre-computed device scores, determine the best available device
- Generate task name if friendly name not specified
- Based on the running workers telemetry

information:

- If worker is running, continue
- If worker is not running, perform COAP POST */createWorker* specifying the worker type
- Once worker is up, push the arguments via dedicated Kafka topic *oecf_taskType_workerName*
- List all Kafka Topics starting with *oecf_taskType_workerName_* and subscribe to them for receiving tasks updates

2.2 Worker

The worker's responsibility is to carry computations on behalf of the orchestrator. The runtime acts as shell for incoming tasks, each worker being loaded with the logic of a single task type and can carry one or more tasks in parallel. The process is spun-up in a synchronous manner by the local orchestrator either on its behalf or on the request of a remote orchestrator. This process is triggered once the orchestrator receives the COAP POST */createWorker* request.

The worker process is language agnostic, the only requirement being the implementation of the common interface:

```
String process(String input)
```

For the sake of simplicity, we'll continue discussing based on a Java reference implementation of the worker. As such, the worker is to be packaged as a jar file and spun up by the local orchestrator on-demand, providing the task type as command line argument.

```
java -jar $OECF_HOME/worker/worker.jar
<TASK_TYPE>
```

The task type argument is needed since the worker runtime is just a shell responsible for supervision of task execution, updating the state of the running task and facilitating incoming task requests, following the lifecycle outlined in Figure 5.

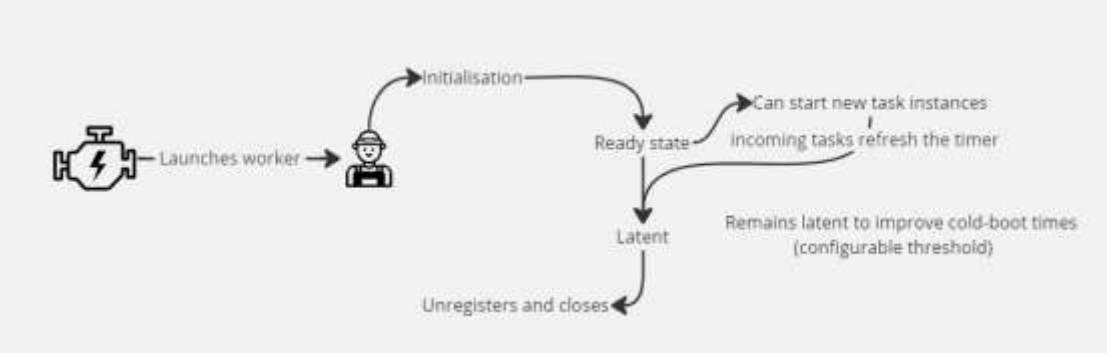

Fig. 5. Worker Lifecycle

In the reference Java implementation, the initialization logic of the worker starts by parsing the task type from the command line argument. Then, the worker needs to determine if the application is already cached or needs to be downloaded. For that step, the worker will list all the files in the *$OECF_HOME/worker/jars* directory and will perform the matching based on the following format: *task_type_version.jar*. If the jar file is not locally present, it will get pulled from the central application repository, in this case an AWS S3 bucket. To prevent malicious code from getting executed on worker nodes, all applications need to be digitally signed by a central authority; as such, once the application file is available locally, the *jar* will have its signature verified with the Public Certificate of the signing authority.

Once the code's integrity and source are verified, a custom class loader will be instantiated that allows the worker process to load application code at run-time. To be able to correctly instantiate the application, all jars are bundled with descriptor files in YAML format.

```
e.g.: mainClass: org.example.TestClass
```

The above example showcases the minimum required application descriptor specifying the entry point for the application. Sanity checks are performed using Reflection API to make sure that this application can be instantiated, verifying that it correctly implements the Worker interface and has a default constructor exposed.

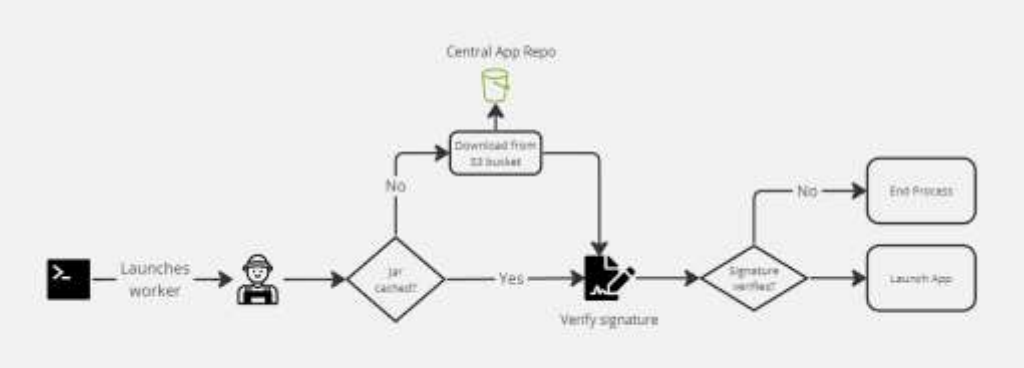

Fig. 6. Application Loading

Having performed the initialization logic as described in Figure 6, the worker will signal to the local orchestrator that it is up and ready to receive commands. This step is necessary since nodes which already have a worker process spun-up are given a device score bonus since the time to execute is significantly decreased by bypassing any initialization logic and going straight to command execution.
The commands will be received via Kafka Topic *oecf_taskType_workerName* from the originating orchestrator, the worker registering an event handler on that topic which wraps the incoming arguments in a *WorkExecutor* that invokes the underlying application logic and handles state updates transparently. All the work is submitted to be ran in a separate thread, orchestrated by an Executor Service. This message passing allows decoupling of worker from orchestrator, the computing being carried in a transparent matter indifferent of the originating source.
Additionally, cold boot times of applications are improved by maintaining workers in a latent state for a configurable amount of time after the last task execution, thus being ready as soon as the any task is being submitted to the orchestrator.

2.3 Telemetry Component

To be able to effectively schedule computing workloads on different nodes, knowing the real-time system load plays a pivotal role, as this directly impacts the device's ability to perform in a reliable and time-efficient manner.
As such, booting up the telemetry component is a mandatory step as part of the device enrolment and initialization process. This module is responsible with tracking the CPU load, available RAM, flash storage and running workers. This information is periodically collected and submitted on the telemetry topic to which all participating orchestrators are subscribed to.

e.g.:
```
{
  availableCPU: 45 (%),
  availableRAM: 8192 (MB),
  availableFlash: 500 (MB),
  runningWorkers: [Type1_Woker1, ...]
}
```

2 4 Remote Logger

Observability [14] plays a pivotal role in modern distributed systems, helping deliver faster, automated problem identification and resolution. As such, this framework was built with rich monitoring support and goes beyond that by incorporating observability staples such as health checks, log aggregation, exception tracking and collection of key metrics.
Key metrics are captured by telemetry component, while health checks are assured by using anu received data as heartbeat signal; this ensures that all parties in the pool are in a ready-

to-compute state, using the last heartbeat as an effective eviction policy.
The remote logger component is responsible for capturing console output of the running workers, thus allowing effective monitoring of the state of the system. These logs can then be aggregated and filtered by using a standard Elastic Stack, thus achieving a good level of observability.

3 Real-World Applications and Future Directions

The above framework presents an opinionated method of developing and deploying edge computing applications on a scale. Besides the traditional IoT applications such as smart homes and industrial automations, this framework presents the user with the ability to perform large scale calculations in a distributed manner on many heterogeneous devices.
This ability can be leveraged in multiple real-world scenarios such as video processing, ML inference, network analysis, and many others. To dive deep into the video processing example, the current computing model for such applications (e.g. YouTube) implies the transfer of videos over the Internet to be encoded and resized to a plethora of resolutions. This kind of application is easily parallelizable, being able to easily breakdown videos into chunks and distribute it among multiple machines; this would greatly speed up the encoding speed but would not reduce the strain placed on the network.
To summarize, this application has two major cost components: the computing power, while albeit seemingly infinite has a hefty price tag, and the network component, where any infrastructure project is in the billions range [15]. As such, the cost of supporting the ever-increasing amount of user generated content is bound to outpace the monetization potential of those video.
This framework attempts to be an enabled to continuous growth by bringing the computation closer to the user base. A tiered approach can be implemented where the computation is first performed on local devices (e.g. transmitting data over LAN), with excess falling over to devices in proximity (e.g. devices in the same region as the user), ultimately offloading it to the cloud if no other devices are available. This model, depicted in Figure 7, shall greatly reduce the network impact, with each chunk being in transit for a lower amount of time.

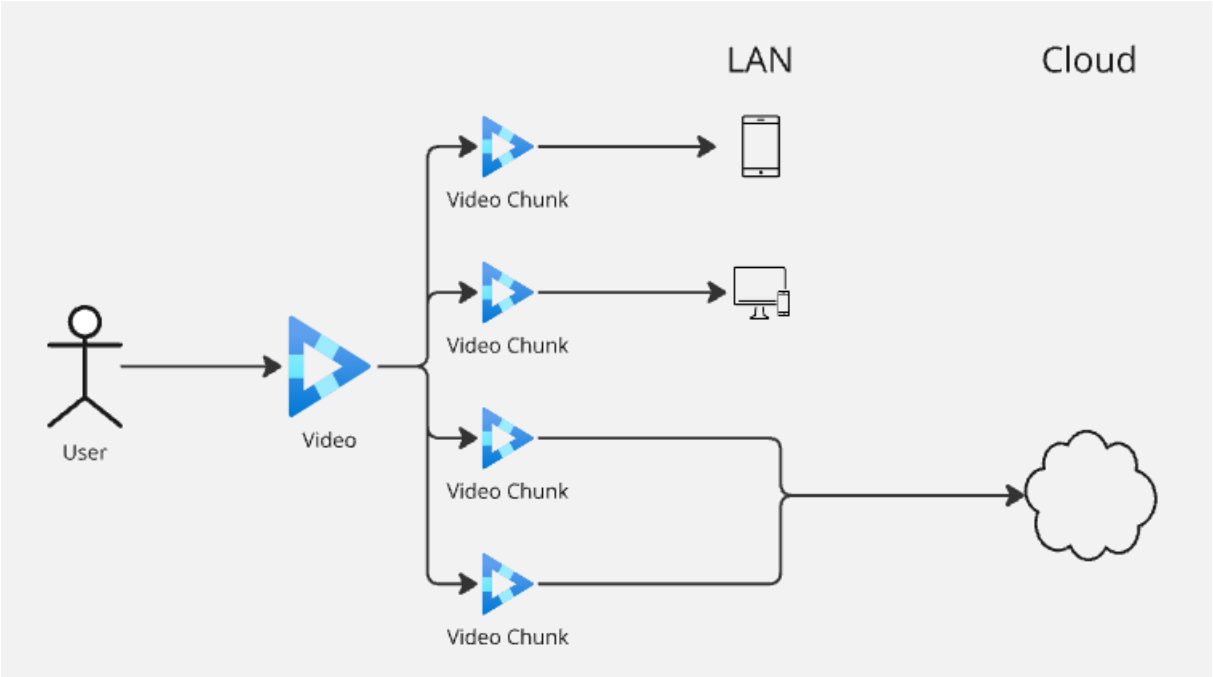

Fig. 7. Video Processing application developed in OpenEdgeComputeFramework

Moving forward, a point of future development of this framework would consist

in evaluating the performance against existing solutions together with identifying current bottlenecks. One such example would be replacing the usage Kafka with the lighter weight MQTT protocol [16].

4 Conclusion

To summarize, this paper proposes a shift from the current direction in edge computing which leverages container runtimes to offer platform agnosticism and offer runtime uniformity guarantees. OpenEdgeComputeFramework proposes a leaner approach, offering a slim Java Worker which can pull application logic from a central repository, but consuming less resources per worker than other solutions, reusing the Java Runtime of the host.

Framework initialization and worker coordination is performed by the Orchestrator, the central component of the framework. This module allows device discovery, careful workload scheduling based on a continuously updated device score and monitoring on initiated tasks. All of this is done in a peer-to-peer and decoupled manner by utilizing Kafka, which adds fault tolerance to the solution.

The orchestrator is also responsible for creating new workers, always started on the local machine either on its behalf or on the behalf of another orchestrator. The worker process is a slim wrapper around run-time loaded application code which is responsible for transparently handling any framework abstractions such as task updates, latent timers used to keep the worker alive for a configurable amount of time to improve cold-boot times, reporting back to orchestrator, command marshalling and many others. All the loaded application logic is loaded in a centralized repository, in this example an AWS S3 bucket, in the form a signed jar file. The Java Archive is signed by a certified signing authority to prevent execution of malicious code, thus ensuring the origin and integrity of the downloaded application.

Playing a pivotal role in the good working of the framework is the telemetry component, which collects vital information about the state of the host system such as available CPU, RAM and flash storage, while also keeping track of the running workers on the system. This information is periodically emitted as an event on the telemetry topic to be picked up by all the other nodes participating in the pool, which will become inputs in the computation of the device score and will also act as a heartbeat signal marking that the node is up and healthy.

Observability is ensured by remote logger component, which forwards all application log to be later aggregated and visualized using Elasticsearch.

To sum this all up, this paper proposes a complete solution for edge-computing which allows transparent orchestration of work among different nodes, be them edge or cloud, using lightweight worker nodes which encapsulate Java application logic at startup while also providing good observability over the inner workings of the system.

References

[1] Akamai, "What is Edge Computing?," [Online]. Available: https://www.akamai.com/glossary/what-is-edge-computing.

[2] K. Cao, Y. Lafiu, G. Meng and Q. Sun, "An Overview on Edge Computing Research," IEEE Access, vol. 8, 2020.

[3] Cisco, "VNI Complete Forecast Highlights," 2021.

[4] M. Pollet, "EU looks to boost secure submarine internet cables in 2024," Politico, 11 October 2023.

[5] Light Reading, "The future of the Internet is underwater," Light Reading, 8 December 2022.

[6] W. Shi, J. Cao, Q. Zhang, Y. Li and L. Xu, "Edge Computing: Vision and Challenges," IEEE Internet of Things Journal, vol. 3, no. 5, pp. 637-646, 2016.

[7] H. Hua, Y. Li, T. Wang, N. Dong, W. Li and J. Cao, "Edge Computing with Artificial Intelligence: A Machine Learning Perspective," ACM Computing Surveys, vol. 55, no. 9, 2023.

[8] D. Silitonga, S. A. A. Rohmayanti, Z. Aripin, D. Kuswandi, A. B. Sulistyo and Juhari, "Edge Computing in E-commerce

Business: Economic Impacts and Advantages of Scalable Information Systems," EAI Endorsed Transactions on Scalable Information Systems, vol. 11, no. 1, 2024.
[9] S. Yi, C. Li and Q. Li, "A Survey of Fog Computing: Concepts, Applications and Issues," Mobidata, 2015.
[10] EdgeX Foundry, "Why EdgeX?," [Online]. Available: https://www.edgexfoundry.org/why-edgex/.
[11] Amazon Web Services, "What is AWS IoT Greengrass?".
[12] Q. Deng, M. Goudarzi and R. Buyya, "FogBus2: a lightweight and distributed container-based framework for integration of IoT-enabled systems with edge and cloud computing," in International Workshop on Big Data in Emergent Distributed Environments, 2021.
[13] Geekbench, "Geekbench Browser," [Online]. Available: https://browser.geekbench.com/.
[14] IBM, "What is observability and why is it important?," [Online]. Available: https://www.ibm.com/resources/automate/observability-basics.
[15] F. Khan, "The Cost of Latency," [Online]. Available: https://www.digitalrealty.com/resources/articles/the-cost-of-latency.
[16] C. L. D. Ho, C. Lung and Z. Mao, "Comparative Analysis of Real-Time Data Processing Architectures: Kafka versus MQTT Broker in IoT," in 4th International Conference on Electronic Communications, Internet of Things and Big Data, 2024.

Andrei CAZACU has graduated the Cybersecurity Master Program, Bucharest University of Economic Studies in 2022 and is currently pursuing a PhD in the field of Fog-Edge Computing. Currently, he is activating as a software developer in the payments field.