## *LiveDewSim: A stream processing experimental platform for* ***running*** *deep learning on smartphone clusters at the edge*

# Cristian Mateos, Matías Hirsch, Juan Manuel Toloza, Alejandro Zunino

ISISTAN-UNICEN-CONICET, Tandil, Argentina

{cristian.mateos,matias.hirsch,juanmanuel.tolozajuanmanuel.toloza,alejandro.zunino}@isistan.unicen.edu.ar

# Abstract

# Keywords

# Mobile devices, stream processing, deep learning, Dew computing, Android

# Metadata

|  |  |  |
| --- | --- | --- |
| **Nr** | **Code metadata description** | ***Please fill in this column*** |
| C1 | Current code version | *V1.0* |
| C2 | Permanent link to code/repository used for this code version | [*https://github.com/matieber/livedewstream*](https://github.com/matieber/livedewstream) |
| C3 | Permanent link to reproducible capsule | [*https://github.com/matieber/livedewstream*](https://github.com/matieber/livedewstream) |
| C4 | Legal code license | GNU GPL |
| C5 | Code versioning system used | *git* |
| C6 | Software code languages, tools and services used | *Python, Android, shell scripting* |
| C7 | Compilation requirements, operating environments and dependencies | *The emanager\_server and scnrunner modules run on Linux-based machines using Python 3.7+ (no compilation needed); Normapp runs on Android 6+ (please build it using the provided Android Studio project). A full list of dependencies and installation instructions are available at the GitHub site* |
| C8 | If available, link to developer documentation/manual | [*https://github.com/matieber/livedewstream/doc*](https://github.com/matieber/livedewstream/doc) |
| C9 | Support email for questions | [matias.hirsch@isistan.unicen.edu.ar](mailto:matias.hirsch@isistan.unicen.edu.ar) |

# Motivation and significance

# The Fog computing paradigm [2] was introduced in 2012 with the goal of providing highly-scalable network and computing infrastructures for latency and location-aware (mobile) IoT applications, while augmenting resource-constrained devices with processing/storage resources in their proximity. Several and varied technological alternatives to realize this idea, including cloudlets, mobile edge computing, micro datacenters, nano datacenters and femto clouds, have been introduced [1], which aim at processing data/computations using computing resources located at the edge of the network -accessible through wireless protocols- and optionally using remote resources in the distant Cloud when necessary.

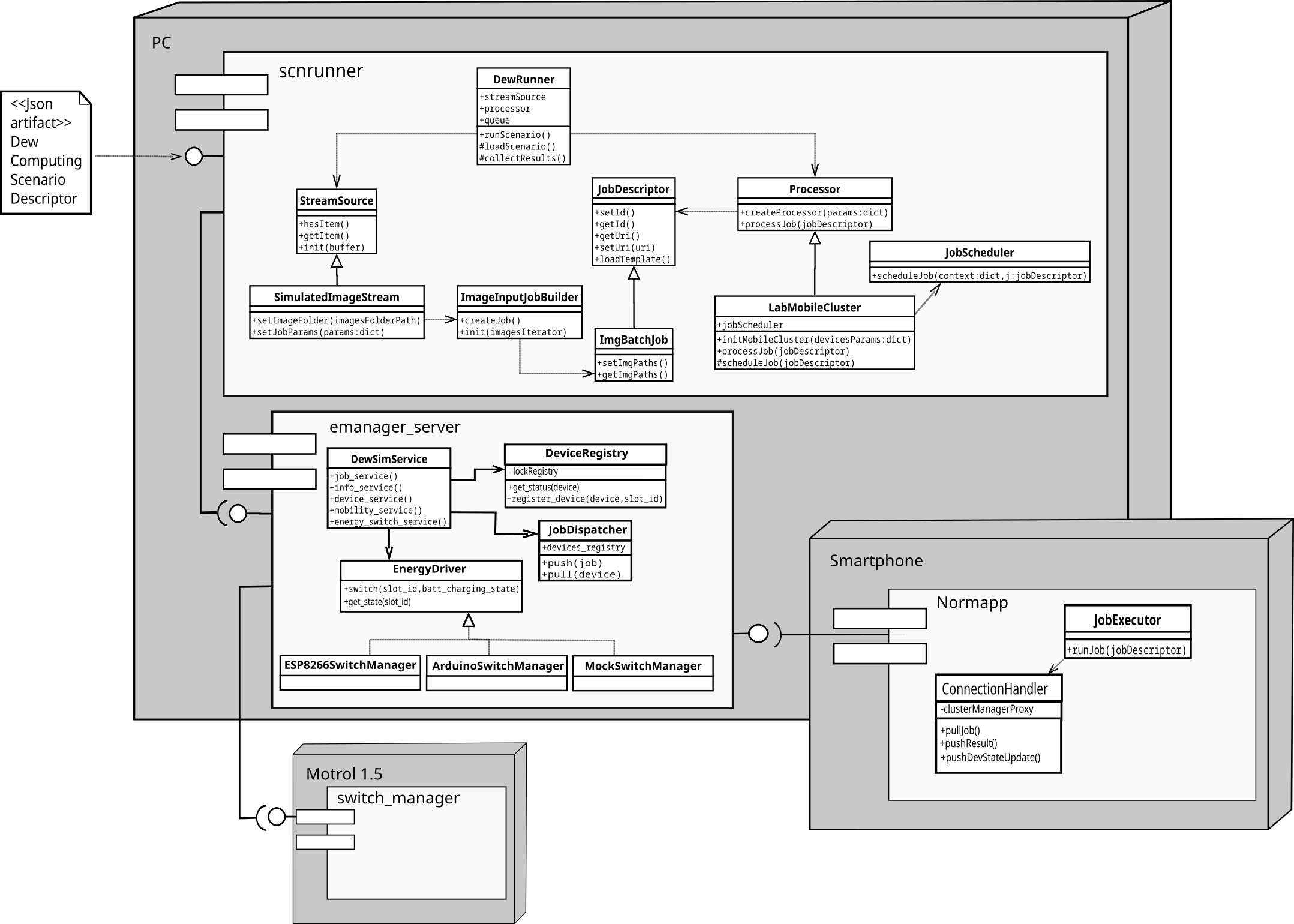
# Since then, there has been a tremendous growth in the amount of resource-rich devices –and hence computational resources- available at the closest edge. According to Statista.com, nearly 84% of the current world’s population owns a smartphone. Modern smartphones contain, on average, more than a dozen sensors, up to eight cores, and powerful GPUs. Likewise, thousands of purpose-specific sensing devices such as surveillance cameras, smoke detectors, noise detectors, and so on are being deployed across buildings and cities around the globe. This reality has led to another paradigm shift by which relying on not-so-close fog servers to support IoT applications consuming and processing the streams of data from such devices might not suffice, whereas massively exploiting hardware at the closer edge is the solution. This is particularly true considering that today’s IoT applications are becoming more commonplace, sophisticated and intelligent, and therefore the timely and efficient execution of increasingly complex tasks and the context-aware processing of larger amounts of data in urban scenarios is needed, which is difficult for centralized clouds and challenging even for fog infrastructures.

To this end, Dew computing proposes to establish clusters of mobile devices at the very edge [3], as a form of “ubiquitous and opportunistic computing” within data sensing contexts. The goal is to exploit in principle the smartphones around us daily to support these applications, specially in public places -public transport, classrooms, coffee shops, and so on- where many nearby devices are present, and hence a cost-effective and performance efficient platform for intelligent IoT applications emerges provided computational resources are managed wisely. Therefore, platforms and tools to study the individual/collective capabilities and limitations of smartphones for intelligent data stream processing are needed.

This paper presents a software platform aimed at supporting experimentation with a specific but broad family of such applications, i.e. those using deep learning over image data streams. The framework allows users -in-lab Dew researchers- to specify automatic, repeatable batch benchmark plans, while indicating specific smartphone-ready deep learning models and task scheduling algorithms for the cluster. Even when we have already proposed a Dew simulation software [3], our framework represents the first step towards “in vivo” benchmarking of mobile devices for such applications, pretty much like commercial platforms such as BrowserStack [4] and LambdaTest [5] allow users to create in-Cloud device farms with the goal of test automation of web and mobile applications. The scientific value of this experimentation platform is threefold, namely to allow researchers to a) to realistically characterize and compare smartphone hardware capabilities when it comes to executing deep learning codes over arbritrary data streams using multi-core CPUs and GPUs, b) to experiment with different cluster settings and task scheduling criteria, and c) to gather smartphone profile data that might be in turn employed to feed back existing Dew simulators, thus creating a virtuous circle in deriving task schedulers [9]. As we support arbritrary streams and tensorflow models, in practice, derived knowledge using our platform might impact many disciplines where Dew computing is the killer computing paradigm, such as the ones illustrated later in the paper, i.e. Smart cities and Agriculture 4.0.

# Software description

# Software architecture

Figure 1: Arquitectural view of LiveDewStream. *The diagram depicts the* UML deployment view, and within each node, the UML components and classes modeling the various entities

From an architectural standpoint, the platform is essentially a client-server software system, complemented with a hardware device called Motrol (see Figure 1). Clients are on one hand the mobile devices being exercised, for which we provide a native Android application called *Normapp*. This application has been tested with devices running Android 6 onwards. The server, called *emanager\_server*, runs on a conventional Linux-powered machine. It is implemented in Python 3.7+ as an HTTP-powered backend. The server implements logging and graceful shutdown using the *logging* and *signal* modules of Python. Finally, another client is a module named *scnrunner*, which is also written in Python 3.7+. This is the module in charge of parsing Dew computing scenario parameters, generating corresponding data streams, deriving deep learning-based jobs, deciding which node executes which job, and submitting jobs to the server and hence indirectly exploiting attached mobile devices.

The server works by assuming an energy managing device -Motrol- for which Python-based drivers are provided. Via its support for dynamic energy supply switching, Motrol allows researchers to automatically repeat/reproduce job set executions involving several smartphones configured with a specific battery level. Currently, we support an USB-interfaced Arduino device called Motrol 1.0 [6], and a WiFi-enabled ESP8266-based microcontroller device called Motrol 1.5 (see Figure 2). A more complex prototype based on the Raspberry Pi 4 Model B, called Motrol 2.0 [7], is under development but it is still not considered in this submission. For example, this model will support USB-charging in addition to AC-charging, a feature missing in Motrol 1.0 and Motrol 1.5.

Internally, Motrol 1.0 and Motrol 1.5 use electromechanical relays and provide low-level operations to control energy supply for attached mobile devices. We also provide a mock energy manager to operate the whole platform without Motrol, which prompts the user to manually plug/unplug a specific smartphone[[1]](#footnote-2) from the power grid during test execution as needed. Naturally, this plug/unplug behavior, when using Motrol, remains automatic.

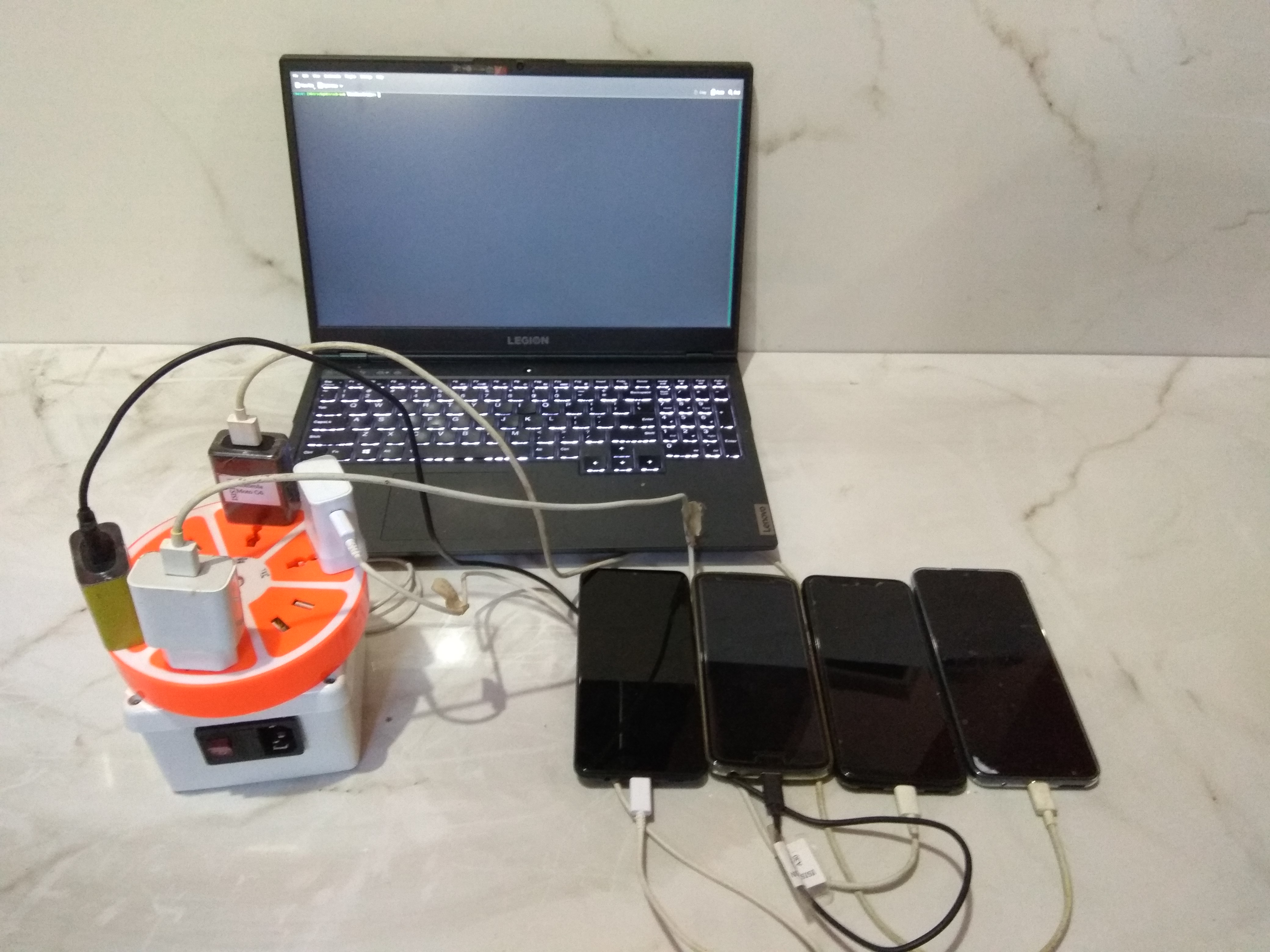


Figure 2: Physical nodes where *LiveDewStream* operates

Motrol 1.5

(optional)

Smartphones

powered by Motrol

Scnrunner and emanager\_server running in a Linux-based laptop

In terms of software design, *emanager\_server* exposes several Rest APIs, which are listed in Table 1. Please refer to the project’s documentation for detailed API specifications in the popular Swagger format [8].

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Rest API** | **Purpose** | **HTTP verb** | **Functionality** | **Invoked by** |
| EnergySwitchService | Querying and switching smartphone energy state | GET | Returns energy state (ac\_charging, usb\_charging, discharging) given a slot number. | Normapp |
| PUT | Switches a slot number to a given energy state |  |
| DeviceService | Registering and coordinating smartphones and e\_manager\_server in order to execute jobs | GET | Asks for jobs to execute | Normapp |
| PUT | Updates smartphone info (battery level and RSSI) |  |
| POST | Submits finished job results |  |
| JobService | Managing incoming jobs | PUT | Blanks (reset) a given device job queue | scnrunner |
| POST | Submits jobs for execution in smartphones |  |
| InfoService | Querying information about attached smartphones | GET | Returns current battery level and RSSI, IP, slot number, connection state (see next Service), and pending jobs of attached smartphones | scnrunner |
| MobilityService | Logically connecting/disconnecting smartphones | PUT | Changes the connection state of a smartphone. *Disconnected* means the smartphone is still attached to the server/energy device, but it is not considered for executing incoming jobs (default is *connected*) | scnrunner |

Table 1 – Rest APIs exposed by emanager\_server

In respect to scnrunner, it is a subproject designed to encapsulate logic that facilitates stream-derived jobs modeling and its online execution using real Dew computing testbeds. Modeling stream-derived jobs consist in generating partitions from a continuous flow of data, for example, a set of images captured within a time window, so as its processing is treated by the execution support as an indivisible workload to be run (atomic job). By online execution we mean that once an atomic job is available, a scheduling mechanism is activated to delegate its execution to a worker node within the Dew computing environment (a smartphone within the smartphone cluster). In its current development state, atomic jobs execution mean to perform object recognition and classification operations with deep learning using images as input. By extending the *src/scnrunner/stream.py* module, i.e., providing custom implementations to the *has\_items* and *get\_item* methods of the *StreamSource* class, new stream types can be supported, e.g., audio streams, text streams, etc.

In conjuntion with *emanager\_server*, *scnrunner* eases the reproduction of experimental settings -Dew computing scenarios- in order to study relevant metrics -throughput, battery utilization, latency- as a result of using different scheduling criteria under real Dew computing environments. An experimental setting is mainly characterized by a workload generation and node states reset.

*scnrunner* provides a run.sh script that is the entry point to execute a given set of Dew computing scenarios. The script receives a path to a directory containing Dew computing scenarios descriptor files as the only required argument.

# Software functionalities

To explain the functionality of the platform, let us step into how the user operates the platform. Once installed in a PC, the first step is to start the server, by optionally indicating the total number of mobile devices to be employed in the benchmark session. By default, this parameter is the maximum number of smartphones (connecting *slots*) supported by the configured energy device (please see *src/emanager\_server/serverConfig.json*).

The server will then initialize each device involved in the session, by asking the user to plug, one by one, each device via USB to the PC. This allows the server to gain root access to the device to a) pushing essential configuration such as server IP address and port, and mostly b) copy and run shell scripts directly on the device that are needed for benchmarks to correctly operate, and c) remotely install (or update) and run our Android application on the device. It is worth noting that tasks a) and b) are performed via the ADB (Android Debug Bridge) [10] tool of the Android SDK, which allows a PC to remotely send commands to a daemon which runs on a mobile device. On the other hand, task c) is done by using ADB in conjunction with Monkey [11], a program that emulate streams of mobile user events such as clicks, touches, or gestures, as well as a number of system-level events. Via Monkey, we launch and (if configured to do so in *serverConfig.json*) start the application automatically.

Once Normapp is installed and launched in the participating smartphones, the user unplugs each mobile device from the PC and plugs it using its original charger to the configured energy supply hardware -Motrol 1.0 or Motrol 1.5- or to wall sockets -Mock-. Normapp will periodically poll[[2]](#footnote-3) the server via the *DeviceService* Rest API for jobs to execute, execute individual jobs, and submit the results back to the server. Job creation and result summarization is responsibility of the *scnrunner* module, the second major subproject of the platform, which is illustrated next.

# Sample code snippets

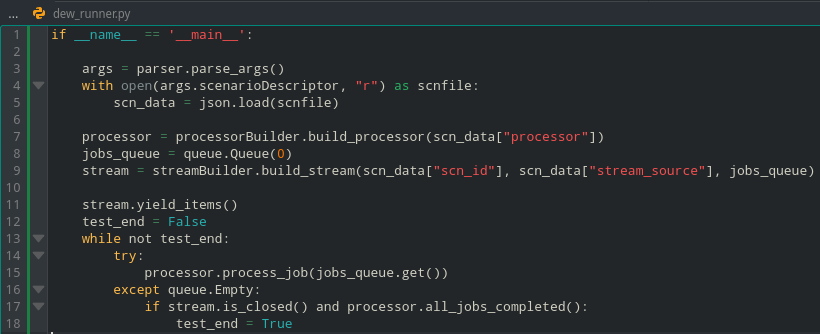
Figure 3: dewrunner.py code invoked from run.sh with logic to parse and create stream and processor entities declared in a Dew computing scenario descriptor file

Figure 3 shows a code snippet of *src/scnrunner/dew\_runner.py*, the module with the main method of *scnrunner*. From line 1 to 9 there is logic to parse the supplied Dew computing scenario JSON file and to create the corresponding stream and processor objects. Besides, at line 8, a shared queue is created (and passed as argument to the *build\_stream* factory method) to enable stream and processor entities to communicate using a producer-consumer pattern. The *Stream* entity, which acts as an items producer, runs in a separate thread that is started upon invocation of *stream.yield\_items()* at line 11. Futhermore, processor entity runs in the main thread consumming items from the shared queue via *jobs\_queue.get()* (line 15).

# Illustrative example

Figure 4 shows an example of a Dew computing scenario configuration file. The *src/scnrunner/run.sh* script accepts a directory path with at least one of these JSON formatted files. The *scn\_id* field identifies the scenario. As can be seen in lines 4 and 5, *scn\_id* is used to name the results directory and an output log file, all created by the platform and where it stores scenario execution related information. From lines 6 to 20, the JSON contains configuration parameters of the stream entity mentioned in Section 2.3. For instance, *img\_folder* indicates the path where input images are located within the filesystem, *per\_job\_frames*, *per\_burst\_jobs*, *millis\_btw\_jobs* and *millis\_btw\_bursts* are parameters used to shape the stream speed and load introduced to the system. In the example, jobs are composed of 30 consecutive frames and these jobs are generated every one second (*millis\_btw\_bursts*). Jobs generation finishes when consumming all images of *img\_folder*. Images are served to *Normapp* through a simple HTTP server which can be accessed through the port specified in line 16. Lines 17 to 20 configure the Python class that is dynamically loaded by the platform whose resposibility is to create the next job item every time the *get\_item()* method mentioned in the previous section is invoked by the platform.

The *processor* key is where parameters of the processor entity are configured (lines 26 to 40). More specifically, *hardsupp.mobile\_cluster.LabMobileCluster* class is loaded and instantiated by the platform to perform nodes initialization and jobs scheduling. To accomplish this, the class uses the *emanager\_server* Rest API. Nodes in this case are four smartphones whose model names and init battery level are indicated under the *devs\_batt\_init* key. Battery level are values in the range [0-1] being 0 equivalent to 0% and 100% respectively. Besides, “-1” is interpreted as “any battery level”. Lastly, desired scheduling logic is configured by providing the path using dot notation of the class that implements a processor logic, in this case, *job.job\_scheduling.RoundRobin*. Notice that paths are indicated relative to where dew\_runner.py is located within the LiveDewStream project.

To conclude this example, Figure 5 shows, on top, a sample directory tree of files generated by the platform after running a test. The *scn\_id* specified in a scenario descriptor file and the system date are used to identify and create a directory where to locate all output files generated by a test (*dogs\_finder\_app/results/cs402\_scn002/jue\_10\_feb\_2022\_15:28:35\_-03*). Such directory includes a plain text log containing chronologically ordered events (see sample log at the middle part of Figure 5). Depending on which entity is logging information, entries of the event log are tagged with [STREAM] or [PROCESSOR]. There is also scenario descriptor file information to easily identify which log corresponds to which input parameters. Lastly, the bottom part of Figure 5 shows an example of results.csv file, a file which is built by the platform by merging individual results sent by each mobile device to the emanager\_server through the corresponding device service. It contains information of which images were sent to which device, how much time the device spent in downloading input image –inputTransferTime(millis) column-- and performing the inference over the image --detectTime(millis) column--. It also records RSSI indicator, battery level at the time of starting downloading images and finish inferences, and the output of the object recognition algorithm itself.

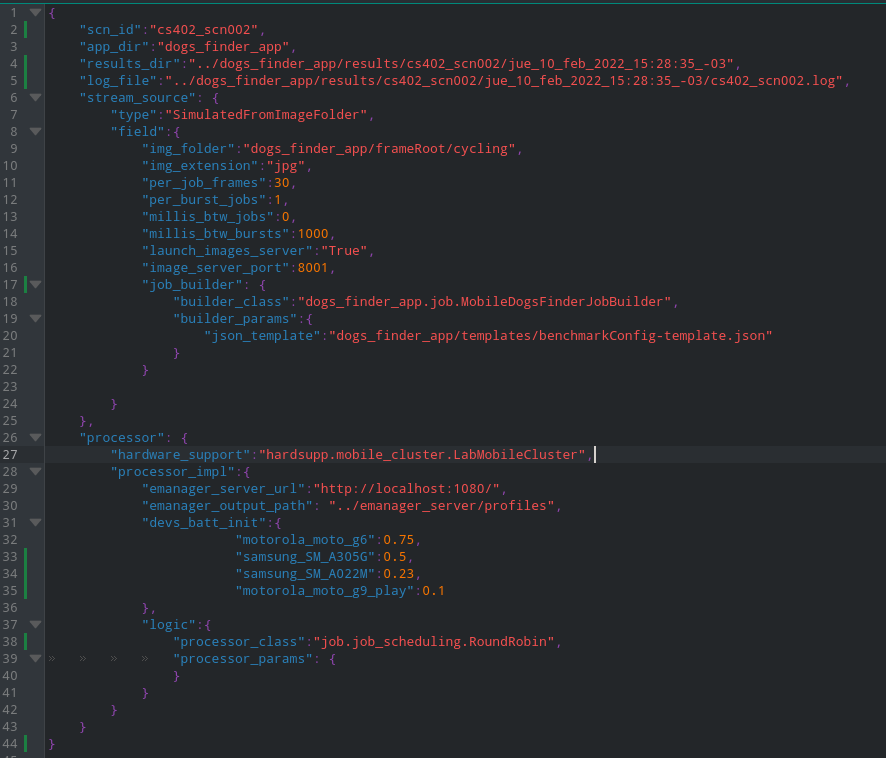
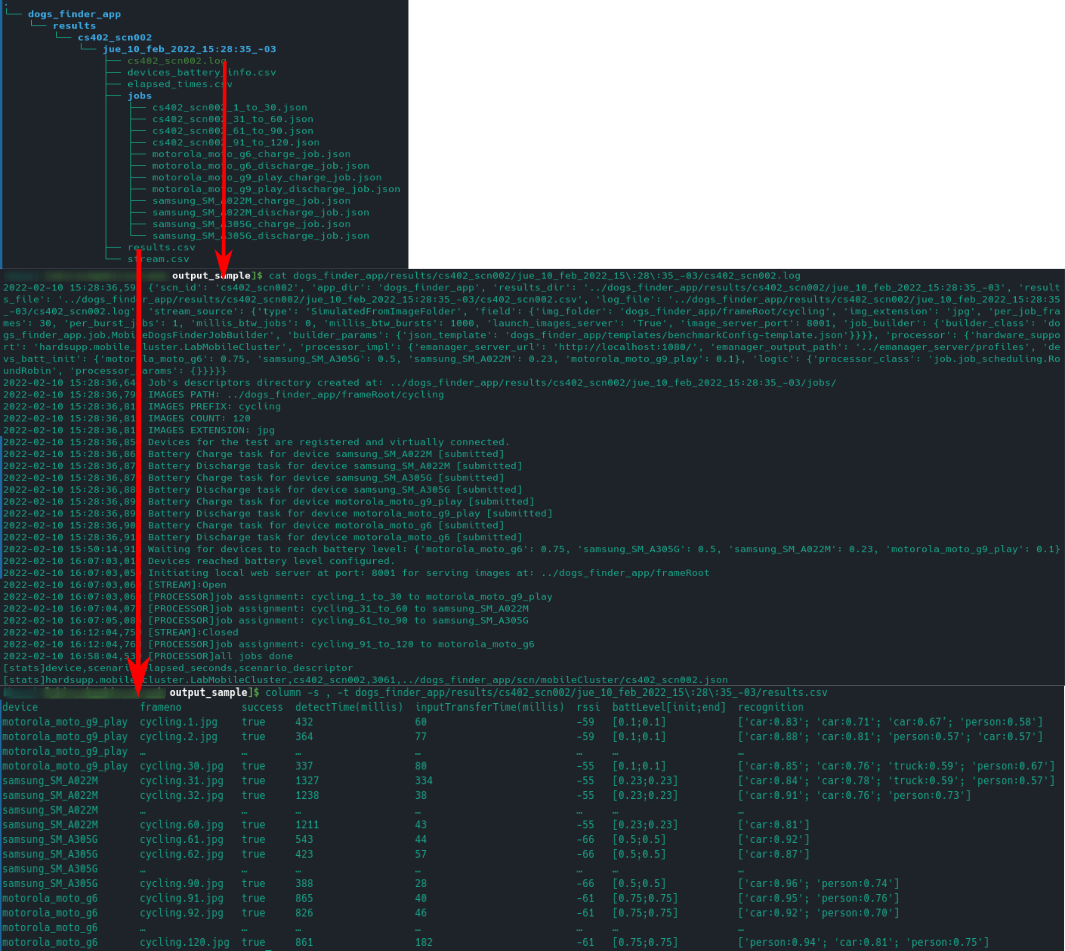


Figure 4: Dew Computing scenario file descriptor

Figure 5: Output sample: Directory tree generated, events log and results file

1. **Impact**

*This is the main section of the article and reviewers will weight it appropriately.*

*Please indicate:*

# *Any new research questions that can be pursued as a result of your software.*

# *In what way, and to what extent, your software improves the pursuit of existing research questions.*

# *Any ways in which your software has changed the daily practice of its users.*

# *How widespread the use of the software is within and outside the intended user group (downloads, number of users if your software is a service, citable publications, etc).*

# *How the software is being used in commercial settings and/or how it has led to the creation of spin-off companies.*

*Please note that points 1 and 2 are best demonstrated by references to citable publications.*

# Conclusions

**Acknowledgements**

We thank Eng. Pablo Sanabria (<https://www.linkedin.com/in/tailmon/?originalSubdomain=cl>) for his assistance regarding running Tensorflow Lite models on Android smartphones.

# References

1. M. Aazam, S. Zeadally, and K. A. Harras. Offloading in fog computing for iot: Review, enabling technologies, and research opportunities. Future Generation Computer Systems, 87:278-289, 2018.

2. F. Bonomi, R. Milito, J. Zhu, and S. 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, pp. 13–16. ACM, 2012.

3. M. Hirsch, C. Mateos, J. M. Rodriguez, and A. Zunino. Dewsim: A trace-driven toolkit for simulating mobile device clusters in dew computing environments. Software: Practice and Experience, 50(5):688-718, 2020.

4. BrowserStack. <https://www.browserstack.com/>

5. LambdaTest. https://www.lambdatest.com/

6. M. Hirsch, C. Mateos, A. Zunino, and J. Toloza. A platform for automating battery-driven batch benchmarking and profiling of Android-based mobile devices. Simulation Modelling Practice and Theory, 109, 102266, 2021.

7. C. Mateos, M. Hirsch, J. Toloza, and A. Zunino. Motrol 2.0: A Dew-oriented hardware/software platform for batch-benchmarking smartphones. IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC), pp. 1772-1777, 2021.

8. Swagger: API Documentation & Design Tools for Teams. http://swagger.io

9. M. Hirsch, C. Mateos, A. Zunino, T. Majchrzak, T.M. Grønli, and H. Kaindl. A task execution scheme for dew computing with state-of-the-art smartphones. Electronics, 10(16), 2021.

10. Android Debug Bridge. https://developer.android.com/studio/command-line/adb

11. Monekey. <https://developer.android.com/studio/test/other-testing-tools/monkey>

1. Under this operation mode, smartphones must be connected directly to the power grid [↑](#footnote-ref-2)
2. A more efficient, push-notification like webhook for Normapp will be developed in the future [↑](#footnote-ref-3)