Received: 15 December 2022 | Revised: 29 April 2023 | Accepted: 30 May 2023

DOI: 10.1002/spe.3231

EXPERIENCE REPORT

WILEY

JAY: A software framework for prototyping and evaluating offloading applications in hybrid edge clouds

Joaquim Silva | **Eduardo R. B. Marques** | **Luís M. B. Lopes** | **Fernando M. A. Silva**

CRACS/INESC TEC & DCC/FCUP, University of Porto, Porto, Portugal

Correspondence
Eduardo R. B. Marques, DCC-FCUP Rua do Campo Alegre, 1055, 4169-007, Porto, Portugal.
Email: ebmarques@fc.up.pt

Funding information
Augmanity, Grant/Award Number: POCI-01-0247-FEDER046103; COMPETE 2020; FCT, Grant/Award Number: UIDB/50014/2020; SafeCities, Grant/Award Number: POCI-01-0247-FEDER-041435; PORTUGAL 2020

Abstract

We present JAY, a software framework for offloading applications in hybrid edge clouds. JAY provides an API, services, and tools that enable mobile application developers to implement, instrument, and evaluate offloading applications using configurable cloud topologies, offloading strategies, and job types. We start by presenting JAY's job model and the concrete architecture of the framework. We then present the programming API with several examples of customization. Then, we turn to the description of the internal implementation of JAY instances and their components. Finally, we describe the JAY Workbench, a tool that allows the setup, execution, and reproduction of experiments with networks of hosts with different resource capabilities organized with specific topologies. The complete source code for the framework and workbench is provided in a GitHub repository.

KEYWORDS

computation offloading, edge computing, hybrid edge clouds, mobile edge clouds

1 | INTRODUCTION

With the advent of the smartphone, IT infrastructures were gradually flooded with requests from huge numbers of these ubiquitous, resource-limited devices. The increase in numbers was staggering, from only a few million in 2008 to 1.06 billion in 2012 and up to more than 6 billion users worldwide in 2021. This number is estimated to grow to more than 7.7 billion by 2026 according to a report by Ericsson.[1] This growth in numbers has been accompanied by a remarkable increase in computational capabilities, storage, communication interfaces, and energy efficiency. Newer models are provided with special-purpose cores for graphics and for artificial intelligence applications such as speech recognition and computer vision.

As is usual with new technology, developers pushed the limits of hardware by implementing more and more demanding applications. However, given the processing and battery life limitations of the devices, sometimes local computation was not an option. Infrastructure clouds came to the rescue and provided the non-local computational and storage resources required to run more demanding applications and/or to store overwhelming large data volumes produced at the edge, a technology that became known as mobile cloud computing (MCC).[2]

With MCC, mobile devices offload applications or jobs therein to remote cloud servers where they are executed and the results returned. MCC is not without drawbacks, however. When using an infrastructure cloud, a stable, high-bandwidth Internet link is usually required. Also, geography implies that communication latency may be considerable and this may

Abbreviations: CO, computation offloading; EC, edge computing; HEC, hybrid edge clouds; MEC, mobile edge clouds.

not be acceptable for some applications. Events such as a super bowl can lead to tremendous pressure on the network and cloud infrastructure due to the enormous number of simultaneous users connecting to the cloud simultaneously using the same network infrastructure.[3] This saturation leads to a marked decrease in QoS. And then there is also the cost associated with renting a cloud server, a value that can easily reach hundreds of dollars a month for an average VM. Most cloud providers do not provide a fixed price. They usually charge cloud server owners per data usage, computing usage, data storage, and other features.

In order to address these limitations researchers proposed several models to bring computation and data storage closer to the edge of the Internet, where the data sources are often located. One of the first such proposals was that of Cloudlet servers.[4] Cloudlets are dedicated servers placed right at the edge of the network, allowing for content caching and some computation. Cloudlet servers vary significantly in their capabilities, ranging from simple single-board computers such as a Raspberry Pi, which can be used as an excellent caching server at the edge, to full-fledged dedicated servers that perform demanding computations. Cloud servers are still present in most of these models and more demanding computations can still be offloaded to this infrastructure.

More recently, taking into account the extreme mobility and possible lack of Internet connectivity of mobile devices, proposals to form mobile edge clouds (MEC) have been put forward.[5] These clouds are formed by mobile devices, using D2D communication technologies, such as Bluetooth or Wi-Fi Direct. Often MEC are useful for data sharing. Applications such as Firechat and Bridgefy, for example, make use of MEC for communication in situations where internet access is cut-off (e.g., in the Hong Kong Protests in 2014[6,7] and in the Myanmar coup in 2021[8,9]).

A device that is part of an edge cloud can divide a demanding computational job and offload it to neighbour devices for execution, speeding up its completion and averaging battery impact among the devices. Several systems have been proposed as proof of concept for such untethered computational platforms. With the rise of the 5G and IOT, the need for efficient computation offloading is more than ever necessary according to an ETSI 2015 report.[10] Pressure is also mounting for handling data at the edge, for example, to preprocess it before sending it to cloud infrastructures.

To deal with these challenges, mixing MEC, cloudlets and MCC is currently a hot topic of research known as hybrid cloud computing. Hybrid clouds are networks of heterogeneous devices composed of three network tiers (Figure 1): infrastructure clouds, cloudlets, and mobile edge devices. The basic idea is that all devices in any of the three tiers can communicate between themselves and work together most efficiently. While the goal is easy to state, getting there is another issue. With hybrid clouds, one needs to address a whole new set of issues that arise with these infrastructures. Hybrid clouds have to weigh the cost/benefit factor of using each tier for a specific operation. Energy consumption, latency, and computational power are key factors in making decisions on such clouds, for example, job offloading decisions. Another complex aspect of hybrid clouds is resource management and network formation, as the pool of devices

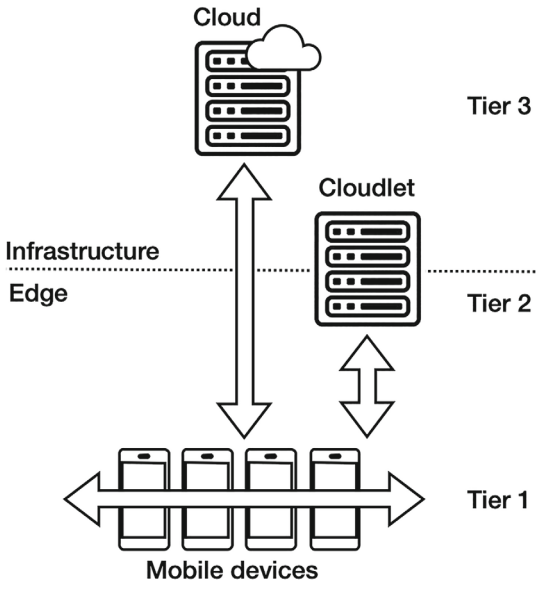

FIGURE 1 Network Tiers: Cloud, cloudlet, and edge.

and servers that compose such a cloud can be highly heterogeneous and use different hardware and software technologies for communication.

In such a complex setting, implementing and testing offloading applications that run over hybrid clouds involves considerable effort from the developer. The strategy used by the application to offload jobs over the tiers should, in principle, optimize runtime metrics such as total execution time, global energy consumption, and fulfilment of QoS requirements. To make a decision, a strategy relies on knowledge of observables such as available network bandwidth, the computational load of devices and servers, and the battery status of devices. This information must be generated at the devices and servers in the different tiers and disseminated throughout the network to provide each application instance with a snapshot (albeit incomplete) of the network status. Naturally, experimenting with different strategies to fine-tune the offloading decisions of the application so that they take the most advantage from the hybrid cloud topology and from the available computational and storage resources at any given instant would go a long way to produce an efficient and robust application. Unfortunately, current hybrid cloud systems do not make this testing and fine-tuning easy for developers.

To address this problem, we designed and developed JAY, a software framework for prototyping and testing offloading applications over hybrid cloud topologies. JAY provides an API and tools that enable developers to implement and test offloading applications using different hybrid cloud topologies, kinds of jobs, and offloading strategies. In previous work,[11] we used JAY to evaluate distinct offloading strategies in different hybrid cloud configurations for a real-world machine learning application. The hybrid clouds were based on combinations of Android devices, cloudlet servers, and Google Cloud servers. Later,[12] we used it to introduce energy-aware adaptive offloading of soft real-time jobs in hybrid clouds.

This article presents JAY in terms of its user API, internal implementation, and support for reproducible experiments. We begin by revising the main traits of JAY in terms of abstract job model and overall architecture,[11,12] then put forward the following main contributions:

- a description of the extensible JAY API with use case examples;
- an overview of the main aspects of JAY's implementation, and;
- the JAY Workbench, a companion tool for automated testing through reproducible experiments.

The article is complemented by the open-source code for the JAY framework and Workbench, available via Github.[13,14]

The remainder of this article is organized as follows. Section 2 presents JAY's job model followed by the architecture we adopted for JAY instances to implement the model. Section 3 describes the JAY API with examples. Section 4 describes the internal implementation of JAY. Section 5 describes the JAY workbench that allows for the specification and testing of application scenarios. Section 6 gives an overview of the state-of-the-art with a focus on comparing JAY to related systems. Finally, Section 7 states the final conclusions and puts forward several ideas for future developments in JAY.

2 | OVERVIEW OF JAY

2.1 | Job model

JAY is a framework for implementing and testing job offloading strategies in hybrid clouds. The job model that constitutes its foundation can be described as follows. We consider a set of hosts connected over a network. Each host executes soft real-time jobs that have deadlines, indicating the maximum tolerable completion time for good QoS. The jobs running on a host may be local, spawned by an application running on the host itself, or offloaded, spawned by an application running on another host. The decision to offload jobs to other hosts can be influenced by job-specific requirements and runtime variables such as the availability of adequate CPU, storage, and energy resources, expected deadline fulfilment, available network bandwidth, and estimated financial cost. We assume that a job's code is locally available to all hosts so that, when a job is offloaded, only its inputs must be provided to the executing host. Later, after the job is finished, its outputs must be sent from the executing host to the originating host.

Every job has a deadline (d), a completion time (T) and a energy cost (E). The latter two are estimated at runtime. Offloading policies rely on determining the necessary estimates for T and E. JAY's job model breaks down time (T) and energy (E) costs in terms of three components, illustrated in Figure 2: T_I and E_I, the time and energy costs for transmitting

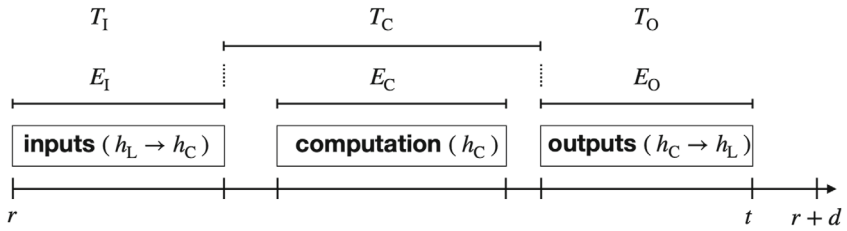

FIGURE 2 Time and energy costs for a job.

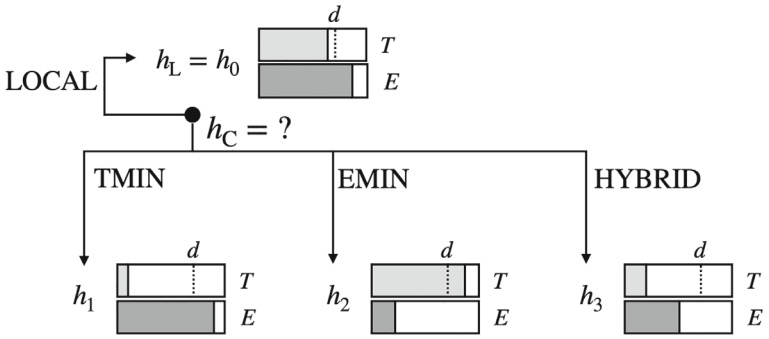

FIGURE 3 Example offloading policies.

the job input data from the originating host to the executing host; T_C and E_C, the costs of actually executing the computation associated with the job, and; T_O and E_O, the costs of transmitting the job outputs (results) from the executing host back to the originating host. Considering the three factors, the overall time and energy cost estimates per host h required by offloading policies can be formulated as $T_h = T_{h,I} + T_{h,C} + T_{h,O}$ and $E_h = E_{h,I} + E_{h,C} + E_{h,O}$.

Energy-wise, this formulation accounts for the energy consumption at the originating host h_L of a job and in the executing host h_C. The energy spent in network I/O expressed by E_I accounts for the energy consumed by h_L in uploading the inputs, and h_C to download them, and vice-versa for E_O in the case of outputs. Another subtle aspect is that the E_C term reflects the energy of executing the job remotely at h_C, but (as explicitly illustrated in Figure 2) only the energy corresponding to the fraction of time that h_C is effectively performing the computation, discarding energy consumption due to the period when the job is pending, for example, when the job is queued due to the existence of other jobs that have precedence over it. In contrast, the total time spent at h_C must be fully accounted for in the case of T_C.

2.2 | Offloading policies

Different offloading policies can be considered, some of which are illustrated in Figure 3 for a network comprising 4 hosts running JAY: $h_L = h_0$, the originating ("local") host, and other hosts h_1, h_2, and h_3. We assume that, for each job, T_h and E_h estimates for each candidate executing host h are available: T_h and E_h. These, respectively, represent the expected completion time and energy consumption of running the job at host h.

Within this framework, one of several offloading policies may be defined to compute h_C, the host chosen to run the job. For instance, as illustrated in the figure: TMIN is a policy that chooses the host that is estimated to minimize execution time, without regard for energy consumption; EMIN, inversely to TMIN, chooses the host that minimizes energy consumption, but does not account for completion time, thus, has no concern also for the job's deadline; HYBRID adopts a compromising strategy as it chooses the host with lower energy cost, but only among those which can fulfil the job's deadlines, or; finally, LOCAL, the void offloading strategy that always chooses $h_C = h_L$, that is, all jobs execute locally. The corresponding mathematical formulations for these example offloading strategies are quite simple:

$$\begin{aligned}
\text{TMIN} &\equiv h_C = \text{argmin}_h\, T_h, \\
\text{EMIN} &\equiv h_C = \text{argmin}_h\, E_h, \\
\text{HYBRID} &\equiv h_C = \text{argmin}_{h\,:\,T_h \le d}\, E_h, \\
\text{LOCAL} &\equiv h_C = h_L.
\end{aligned}$$

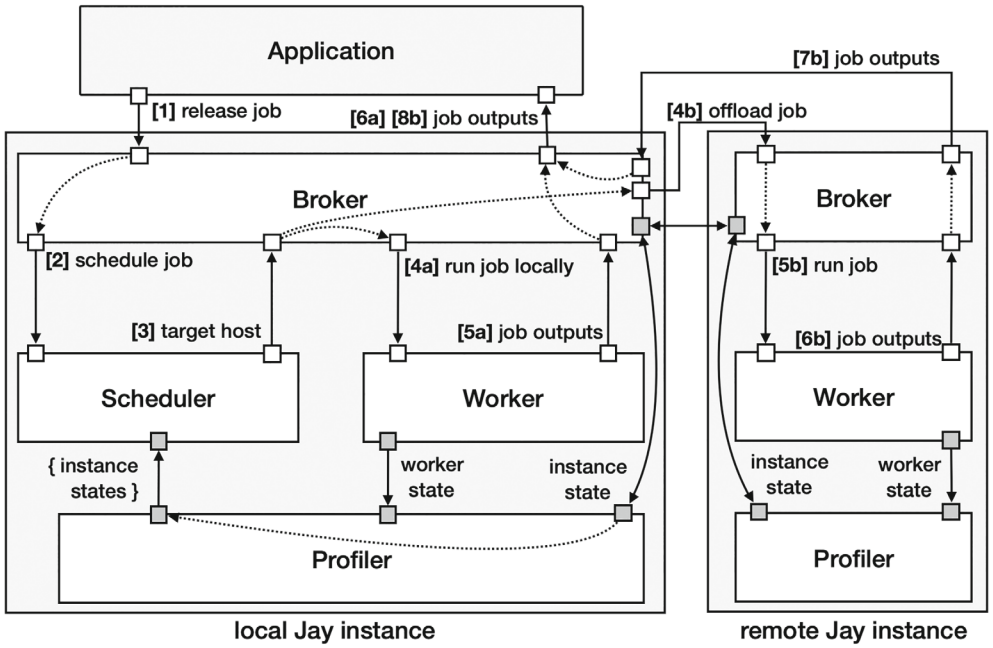

FIGURE 4 JAY architecture.

Note that, each host can run a different offloading strategy, effectively allowing hosts in a hybrid cloud to decide on the best strategy to offload jobs, based on their network context and dynamic estimates for time and energy. Moreover, JAY is agnostic with respect to the type of local scheduler used by hosts that receive offloaded jobs. We assume only that when a job is offloaded to h_C it is executed therein together with a mix of other jobs.

2.3 | Architecture

We now describe the architecture for our concrete implementation of the model described in the previous section. As shown in Figure 4, each JAY instance takes the form of a set of interacting services running on the same host. An instance gets requests from local applications to execute computational jobs. It is up to JAY to decide whether to execute a job locally or offload it to a remote instance. The process should be transparent to a client application only requiring a job's output to be made available once it is completed. When offloading, JAY instances possibly interact across different cloud tiers.

A JAY instance may offload or locally execute jobs. The Scheduler and Worker services within each instance are kept active for being invoking and taking decisions. The Scheduler is responsible for scheduling decisions, that is, to decide whether to run the job at the local instance, the one invoking the service or to offload it to a remote instance, following a configurable offloading strategy that assesses in runtime the conditions for all instances with an active Worker service. The Worker service manages the actual execution of jobs, regardless of whether they are local or incoming from other hosts through offloading requests. JAY instances running only one of the Scheduler or Worker services merely act as job execution clients or servers, respectively.

To allow for offloading strategies that operate dynamically and adaptively according to runtime conditions, the internal state of each JAY instance is continuously monitored by the Profiler service to reflect aspects such as instance energy consumption, job workload, and network transmission times. In addition, the Profiler gets similar state updates from all other JAY instances in the network, Hence, it is then able to construct a global snapshot of the state of all JAY instances in the network at any given time and supply that information to the local Scheduler service.

Broker service is responsible to glue the global operation in each instance. It acts as a network mediator between an instance and local applications and remote instances, abstracting any interactions with other services on the same local instance.

Summarizing the role of each service during the execution of a job, from release to completion, as indicated by the stage numbers in Figure 4: job release (stage 1)—a job is released when an application requests the local Broker service

to execute it; job scheduling (stages 2 and 3) – the Broker forwards the job to the local Scheduler service (2), which in turn informs back the instance that should run the job (3); local execution (stages 4a to 6a)—if the Scheduler's decision is that the job must execute locally, the Broker instructs the local Worker to execute it (4a). When the job completes (5a), its outputs are delivered to the application (6a), and; offloaded execution (stages 4b to 8b)—on the other hand, the Scheduler's decision may be to offload the job to a remote instance. In this case, it is necessary to transmit the job's inputs over the network (4b). At some point, the remote instance will handle the execution of the job (5b) to produce the job outputs (6b). The outputs are returned back to the originating host (7b) and, finally, delivered back to the application (8b).

3 | JAY PROGRAMMING

We now illustrate the use of JAY in practice, together with the main aspects of the JAY API. JAY is written in the Kotlin programming language,[15] and runs on Android devices or standard computers running Linux. Kotlin may in principle be ported also to other operating systems, as Kotlin compiles to Java bytecode, if some platform-specific modules (e.g., for power estimations) are appropriately implemented for the operating system at stake as it happens already for Android and Linux.

JAY does not restrict the kind of application that can take advantage of job offloading. However, in the contexts described in this article, and in the literature, the most common scenario is that of applications that may be split into multiple independent jobs each requiring significant computational or storage resources.

3.1 | Main API

We illustrate the use of the main JAY API with a simple example concerning the calculation of the Mandelbrot set. Listing 1 shows the steps necessary for an application to start a JAY instance, set up particular offloading strategies and job code, and finally define jobs and execute them.

```
1  // Start Jay instance
2  val jay = Jay()
3  jay.startProfilerService()
4  jay.startWorkerService()
5  jay.startSchedulerService()
6
7  // Configure offloading strategy and task executor
8  val strategy = HybridScheduler()
9  val executor = MandelbrotTaskExecutor()
10 jay.setScheduler(strategy)
11 jay.setTaskExecutor(executor)
12
13 // Define a job
14 val deadline = 4000
15 val job = TaskExecutorManager.generateTask (
16        MandelbrotTask(1920, 1080, 0, 1080, 100),
17        deadline
18      )
19
20 // Execute a job synchronously
21 val result = jay.scheduleTask(job)
22
23 // Execute a job asynchronously
24 jay.scheduleTask(job) {
25   asyncResult -> doSomethingWith(asyncResult)
26 }
```

Listing 1: Example use of JAY.

The initialization of a JAY instance involves the creation of `Jay` object and starting up the required services, as shown in lines 1–5 of Listing 1. The broker service is launched implicitly with the creation of the `Jay` object, and the three other services (worker, scheduler, and profiler) are started manually in the example. Given that the worker and scheduler services are activated, this means that the local JAY instance will be able to execute both local and remote jobs (i.e., received from other hosts).

We then configure the active offloading strategy, and the executor for jobs, that is, the code that will be executed for a job (Listing 1, lines 7–11). In the example, we set the active offloading strategy to `HybridScheduler`, which corresponds to a Jay built-in class for the HYBRID time/energy-aware strategy discussed earlier, and the active executor to an instance of a user-defined class called `MandelbrotTaskExecutor` (more on this below).

After proper initialization of a JAY instance, jobs can be defined and executed (Listing 1, lines 13–26). A job is defined by an instance of an application-defined class `MandelbrotTask`, and a relative deadline for execution in the scale of milliseconds (lines 13–18). In the example, the deadline is set to 4 s. The job can then be executed synchronously or asynchronously (lines 21 and 24), and the application can retrieve the job's output results in either case. In a synchronous execution, the caller application thread blocks until the job completes. Alternatively, in an asynchronous execution, the application is notified of the job's completion through a user-defined callback. In any case, the job may either execute locally or remotely through offloading in a transparent manner to the application code.

3.2 | Coding jobs

The well-known Mandelbrot Set[16] calculation involves the calculation of the set of complex numbers z for which the sequence $f_c(z) = z^2 + c$ does not diverge when iterated from $z = 0$, that is, for which the sequence $f_c(0)$, $f_c(f_c(0))$ and so forth remains bounded in absolute value. When adapting this problem to a computational job, a possible solution is to use the standard escape time algorithm that performs a repeating calculation for each (x, y) point in the region of interest of the complex plane. Points in the plane are assigned colors that indicate whether they are members of the set (usually the color is set to black) or the number of iterations after which the orbit of the point exceeds a threshold and diverges to infinity.

```
1  // Mandelbrot task specification
2  class MandelbrotTask (
3    val width: Int,   val height: Int,
4    val startLine: Int, val endLine: Int,
5    val maxIter: Int ) : Serializable {
6    ...
7  }
8  // Definition of concrete task executor
9  class MandelbrotTaskExecutor (name: String = "Mandelbrot set") : TaskExecutor(name) {
10   override fun executeTask(rawtask: Task, callback: ((Any) -> Unit)?) {
11     val obj = ObjectInputStream(ByteArrayInputStream(rawtask.data))
12     val task = obj.readObject() as MandelbrotTask
13     obj.close()
14     callback?.invoke{ calcMandelbrot(task) }
15   }
16   // Mandelbrot set calculation
17   private fun calcMandelbrot(task: MandelbrotTask): Array<IntArray> {
18     val matrix = arrayOf(intArrayOf())
19     for (col in 0..task.width) {
20       for (row in task.startLine..task.endLine) {
21         val cRe: Double = (col - task.width / 2.0) * 4.0 / task.width
22         val cIm: Double = (row - task.height / 2.0) * 4.0 / task.width
23         var x = 0.0, y = 0.0, iter = 0
24         while (x * x + y * y <= 4.0 && iter < task.maxIter) {
25           val xNew = x * x - y * y + cRe
26           y = 2 * x * y + cIm
27           x = xNew
28           iter++
29         }
30         matrix[row][col] = iter
31       }
32     }
33     return matrix
34   }
35   ...
36 }
```

Listing 2: Computing the Mandelbrot Set.

The code for implementing the Mandelbrot set computation is shown in Listing 2. It involves the definition of inputs for the job through class `MandelbrotTask` (lines 2–7) and the code for execution through class `MandelbrotTaskExecutor` (lines 9–36). An instance of `MandelbrotTask` defines a job's input data, which JAY supplies to the code in `MandelbrotTaskExecutor` to obtain the job's output data, a matrix of integers (defined as an `Array<IntArray>`) as defined by the signature of `calcMandelbrot` (line 17). The core of the Mandelbrot computation, shown in `calcMandelbrot` (lines 17–34), essentially applies the traditional escape time algorithm for a region of the complex plane. Each output position in the integer matrix of a given width and height, both specified as part of the input, indicates the number of iterations after which divergence was detected, up to the `task.maxIter` iteration limit (also an input).

For offloading, JAY requires that `MandelbrotTask` is a `Serializable` object, that is, can be serialized into an array of bytes, sent over the network and then deserialized through the Java serialization API (line 7). Moreover, in `MandelbrotTaskExecutor`, the glue code required by JAY is defined by the `executeTask` method (lines 10–15), which every `TaskExecutor` subclass must define. This method contains the logic for deserializing a `MandelbrotTask` instance and subsequently using it in a call to `calcMandelbrot`.

3.3 | Definition of offloading strategies

Applications have accessible for use a number of built-in offloading strategies in JAY, but they can also define custom ones. Offloading strategies are provided as schedulers in the JAY API. Listing 3 illustrates code snippets for the main programmatic support required for the implementation of offloading policies. As shown, `AbstractScheduler` (lines 2–5) defines the abstract `scheduleTask()` methods that are required to be implemented in subclasses. Types `TaskInfo` (lines 7–10) and `WorkerInfo` (lines 12–24) are used in the input and output types of `scheduleTask` respectively. `TaskInfo` provides information regarding the general characteristics of a job to be scheduled: the size of data to be transmitted, the deadline, and the creation time. `WorkerInfo` instances contain runtime information about available workers (those reported as available by the JAY runtime as in the example of Listing 4). This information—for example, computation state, bandwidth estimates, and power consumption estimates—can be used by the scheduler to make offloading decisions.

```
1  // Abstract class for defining a scheduler
2  abstract class AbstractScheduler(private val name: String) {
3    ...
4    abstract fun scheduleTask(taskInfo: TaskInfo): WorkerInfo?
5  }
6  // Information about a job
7  data class TaskInfo(
8    val dataSize: Long,
9    val deadline: Long?,
10   val creationTimeStamp: Long = 0L ) { ... }
11 // Information about the worker
12 data class WorkerInfo( ... ) {
13   var queueSize
14   var queuedTasks
15   var avgComputingEstimate
16   var bandwidthEstimate
17   var powerEstimations: PowerEstimations?
18   ...
19 }
20 // Power estimation values associated with a worker
21 data class PowerEstimations(
22   val idle: Float, val compute: Float, val rx: Float, val tx: Float,
23   val batteryLevel: Int,
24   val batteryCapacity: Float ) { ... }
```

Listing 3: Base definitions and attributes used in the implementation of offloading policies.

```
1  // Concrete implementation of scheduler
2  class LFHybridScheduler : AbstractScheduler("LFHybrid Scheduler") {
3    ...
4    internal fun expectedCompletionTime(t: TaskInfo, worker: WorkerInfo?): Long {
5      // Account both for computation time and transmission time estimates
6      return (w.getQueuedTasks() + 1) * w.getAvgComputingTimeEstimate()
7           + w.bandwidthEstimate * t.dataSize
8    }
9    internal fun expectedEnergySpent(t: taskInfo, worker: WorkerInfo?): Float {
10     // Account both for computation and transmission energy overheads
11     var estimate = worker.getPowerEstimations()
12     return (w.bandwidthEstimate * t.dataSize) * estimate.tx
13          + w.getAvgComputingTimeEstimate() * estimate.compute
14   }
15   override fun scheduleTask(taskInfo: TaskInfo): WorkerInfo? {
16     // Check if local worker can meet deadline
17     val localWorker = SchedulerService.getLocalWorker()
18     if (expectedCompletionTime(localWorker) < taskinfo.deadline)
19       return localdWorker // local worker can meet deadline
20     var selectedWorker = null
21     var minEnergy = Float.MAX\_VALUE
22     // Offload to deadline-compliant worker that consumes the least energy
23     for (w in SchedulerService.getWorkers(WorkerType.REMOTE)) {
24       if (expectedCompletionTime(w) < taskinfo.deadline) {
25         var energyCost = expectedEnergySpent(w)
26         if (energyCost < minEnergy) {
27          minEnergy = energyCost
28          selectedWorker = w
29         }
30       }
31     }
32     // Default to localWorker if no remote workers are suitable
33     return selectedWorker != null ? selectedWorker : localWorker
34   }
35 }
```

Listing 4: Implementation of the "local-first" HYBRID offloading strategy.

We provide an example of such an (adaptive) offloading strategy in Listing 4. It corresponds to a variant of the HYBRID strategy discussed earlier in Section 2.2. Recall that HYBRID picks the worker for which the estimated job completion time is the lowest and that complies with the job deadline. The variant considered in the code, as expressed by `scheduleTask` (lines 15–34), is a "local-first" offloading strategy that offloads to a remote worker if the local worker cannot meet the job deadline. The offloading decision is informed by estimates of the time and energy costs, as expressed by methods `expectedCompletionTime` (lines 4–8) and `expectedEnergySpent` (lines 9–14). Both estimates in turn take into account both computation and network transmission overheads measured adaptively at runtime by the JAY profiler. The time estimate in `expectedCompletionTime` is calculated from the remote worker's current queue size and average job completion time, while the network transmission overhead is calculated from the bandwidth estimate between the local instance and remote worker instance and the data payload size associated with the job. The energy estimate in `expectedEnergySpent` is very similar but also includes the measured energy spent by the remote worker while computing and receiving data while discarding the energy cost associated with job queuing.

At this stage, we did not experiment with fault tolerance mechanisms in JAY. If required, these mechanisms can be implemented at the application level feeding on the job completion/failure notifications delivered by JAY. However, within JAY, custom offloading policies for fault tolerance can be implemented through the extensible API just described. These could handle aspects such as automated job retries/re-offloading in reaction to errors. Other aspects, such as offloading the same job to multiple hosts to cope with host errors or to increase performance, are possible in principle but require a generalization of the job model.

4 | IMPLEMENTATION

In a typical runtime scenario, a Jay instance runs on each host in a hybrid cloud. Communication between the instances happens when offloading a job or receiving results from a previously offloaded job. It is not necessary to create a new Jay instance to run new jobs, locally or offloaded. Additional messages are exchanged between the hosts to provide the local profilers at each host with runtime information such as CPU loads, memory and storage availability, and battery status.

We now turn to the description of the implementation of a JAY instance as defined in Section 2, with a special emphasis on its four core services: Broker, Scheduler, Worker, and Profiler.

4.1 | Overview

The Broker, Scheduler, Worker, and Profiler components are implemented as independently running services within a JAY instance, as illustrated in Figure 5. For each component, a corresponding service is accessible through remote procedure calls (RPC). This modularity allows for different configurations to be defined. For example, JAY instances that act only as workers will not require a running service for the Scheduler component. Similarly, JAY instances that only offload jobs will not require a service for the Worker component. The broker component is the most complex in terms of interactions, given that it acts as a mediator between the local JAY instance and applications or other JAY instances. Moreover, to fulfil these roles, and in line with the job lifecycle described earlier in Figure 4, the Broker communicates as necessary with all the other architectural components in the local instance (Scheduler, Worker, or Profiler). As for the other components, they have an internal logic closely related to their functionality: a set of offloading policies in the case of the Scheduler (as exemplified in Section 3.3), a set of monitoring subsystems in the case of the Profiler (discussed below in this section), and a job queue in the case of the Worker (also discussed below).

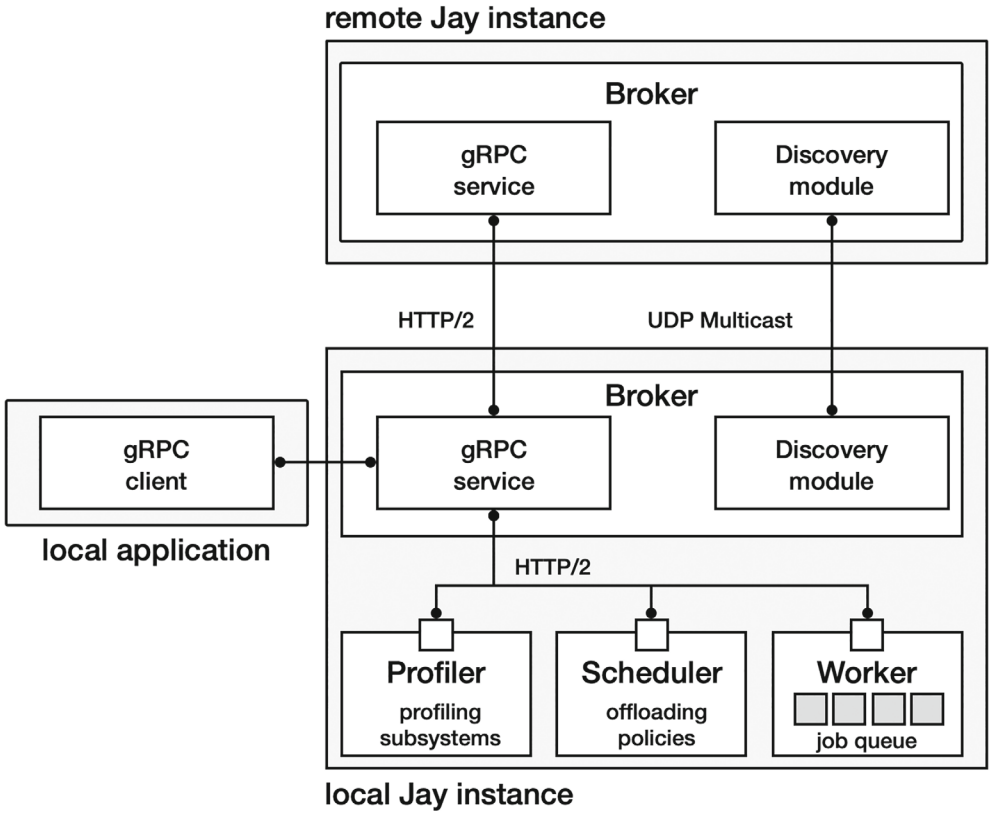

FIGURE 5 JAY components.

These services can have custom, application-specific, modules added. In Section 3, we described how to programmatically add job executors and schedulers to the Worker and Scheduler services, using `registerTaskExecutor` and `registerScheduler`.

Finally, while the Broker service does not allow additions, it is responsible for host discovery in the hybrid network. This is implemented as a mixed static and dynamic decentralized discovery scheme. It uses UDP multicast to find mobile hosts in a network neighbourhood and, in its current incarnation, static URLs/IPs, provided via configuration files from the workbench, to locate cloudlet and cloud servers.

4.2 | Inter-component communication

JAY components interact via gRPC,[17] a very popular framework developed by Google. gRPC is open-source, interoperable in heterogeneous environments (e.g., server machines and Android devices), and has bindings for multiple programming languages (e.g., Kotlin, C/C++, Java, or Python). The HTTP/2 protocol is employed for communication and Protocol Buffers[18] (also by Google) are used as the Interface Definition Language (IDL) to describe the RPC endpoints along with associated data types for call arguments and results. From an IDL description, the Protocol Buffers compiler generates the necessary code for client and server implementation, plus the data structures at stake and the associated support for serialization. We should also note that gRPC supports service discovery but, as mentioned above, we employ our own UDP multicast for dynamic host discovery. The reason is that gRPC requires a centralized registry which is less convenient than UDP multicast for a local mobile edge cloud with high device churn.

A Protocol Buffers IDL fragment for the Broker service is given in Listing 5. It comprises the definitions of datatypes used in RPC calls (lines 15–38) and the actual RPC call signatures (lines 40–49). The message types lead to the generation of data classes that, in addition to their use as RPC call arguments, are used in the JAY API implementation. For instance, the definition of `TaskInfo` shown leads to the data class by the same name illustrated earlier in Listing 3. The `service` section illustrates a subset of the possible RPC interactions with the Broker service: `scheduleTask` can be used by an application to release a job for execution, asking it to be scheduled and later executed; `executeTask` is used between brokers to offload jobs; `setScheduler` sets the active scheduler for the JAY instance; and `stopService` disables the Broker service. Calls like `scheduleTask` can be invoked synchronously, causing the caller to block until the call completes, or, as it happens more often in JAY's implementation, asynchronously through a callback scheme. gRPC also provides for (client-side, server-side, and bidirectional) streaming RPC calls as in the case of `executeTask`.

```
1  // Protobuffer datatypes used by remote API
2  enum StatusCode { Success = 0; Error = 1; Waiting = 2; ... }
3
4  message Status {
5    StatusCode code = 1;
6  }
7
8  message Response {
9    Status status = 1;
10   bytes bytes = 2;
11   string id = 3;
12 }
13 ...
14
15 message TaskInfo {
16   string id = 1;
17   int64 dataSize = 2;
18   int64 deadline = 3;
19   int64 creationTimeStamp = 5;
20 }
21
22 message Task {
23   TaskInfo info = 1;
24   bytes data = 2;
25 }
26
27 message TaskStream {
```

```
  Action status = 1;
  Task task = 2;
  ...
}

message Scheduler {
  string id = 1;
  ...
}

...
// gRPC remote API
service BrokerService {
  // Application interface - job release
  rpc scheduleTask (Task) returns (Response) {};
  // Broker interaction for offloading requests
  rpc executeTask (stream TaskStream) returns (stream Response) {};
  // Configuration and life-cycle
  rpc setScheduler (Scheduler) returns (Status) {};
  rpc stopService (google.protobuf.Empty) returns (Status) {};
  ...
}
```

Listing 5: Fragment of gRPC interface for JAY's broker.

4.3 | Job execution

The worker executes jobs in order of arrival, one at a time, and non-preemptively until completion. Pending jobs are kept on hold in a FIFO queue. In principle, a job execution policy could also be provided as a parameter, as opposed to this fixed scheme, in the same vein as offloading policies. Various aspects could motivate it, for example, the consideration of job deadlines in addition to arrival time as in earliest-deadline-first scheduling, the use of job preemption provided that jobs implement some sort of checkpointing and different queues per job type.

On the other hand, this scheme allows a simple estimation of a job's execution time as $(n+1)\times\tilde{t}$ where n is the number of currently queued jobs and $\tilde{t}$ is a moving average of job execution times. This approximation is used in the method `expectedCompletionTime` from Listing 4, where $(n+1)\times\tilde{t}$ is expressed by `(w.getQueuedTasks() + 1) * w.getAvgComputingTimeEstimate()`. This is important, for example, in the context of experiments (see Section 5) that focus on evaluating offloading policies at the job's originating host.

4.4 | Profiler

The Profiler service is responsible for deriving snapshots of the internal state of the local JAY instance that, combined with similar information from other instances, can guide adaptive offloading policies. As illustrated in Figure 6, the profiler implementation comprises gRPC-based interactions with the other running local JAY services and a number of OS-level monitoring subsystems.

The Profiler requires the gRPC interactions to collect computation and network transmission statistics, specifically job startup and termination at the Worker and network data transmission initiation and termination by the Broker. The

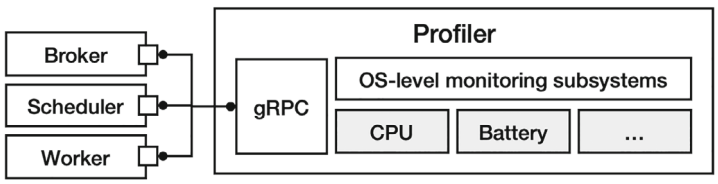

FIGURE 6 Profiler operation.

knowledge of active computation and networking energy consumption drives energy consumption estimation during job computation and transmission of job inputs/outputs. These measurements are then combined with data from OS-level monitoring subsystems, for example, CPU usage and frequency, and energy readings for power and current, to allow adaptive schedulers to make informed decisions about job offloading. Other subsystems in the Android implementation of JAY keep track of hardware activity in the local host device that may also impact energy consumption, for example, subsystems dedicated to monitoring connected sensors or enabled network links.[19] The profiler is not extensible through user code. However, new profiling modules can be implemented directly within the framework. This could be made more flexible, as in task executors and offloading policies, since Jay's profiler is already organized in modules for distinct subsystems.[13,19]

5 | AUTOMATED TESTING USING JAY

5.1 | The JAY Workbench

The JAY Workbench tool was designed and implemented to automate the challenge of orchestrating experiments with multiple devices, namely in what concerns device coordination and the acquisition of runtime analytics. It implements all the logic required to control the multiple JAY instances involved in an experiment, thus allowing it to be executed any number of times with the same inputs, timings, and sequences of events following Poisson distributions. These experiments can be set up seamlessly over manifold hybrid edge-cloud topologies.

The workbench allows individual experiments to be reproduced at any moment for a given hybrid cloud topology and host configuration. This reproducibility is supported by storing all the experimental setup information, including configuration files and seeds for random processes. This capability was of utmost importance for the experimental work described in References 11 and 12. For a given workload configuration, the Workbench needs to determine first if the hybrid cloud topology requirements are met, then set and initiate the workload, and finally collect execution logs for offline analysis. As illustrated in Figure 7, the Workbench takes the form of a Python program that resorts to gRPC to communicate with running JAY instances (once they are active) and the Android Debug Bridge (adb)[20] for configuration actions involving Android devices like rebooting the devices, installing JAY, setting up required permissions, transferring logs and so forth.[19] The hybrid cloud may be composed of a mixture of cloud/cloudlet servers and mobile devices. In general, they will have quite heterogeneous characteristics in terms of CPU, RAM, storage, or power availability and consumption. Cloud and cloudlet servers are identified beforehand explicitly in workload configuration files, while mobile devices are discovered automatically.

5.2 | Case-study experiments

In previous work,[11,12] we used the JAY Workbench to specify, configure, and automate the execution of multiple experiments over different hybrid cloud topologies, job generation rates and offloading policies. We summarize these experimental setups here to exemplify the ability of the workbench to greatly simplify the planning and execution of the tests involved.

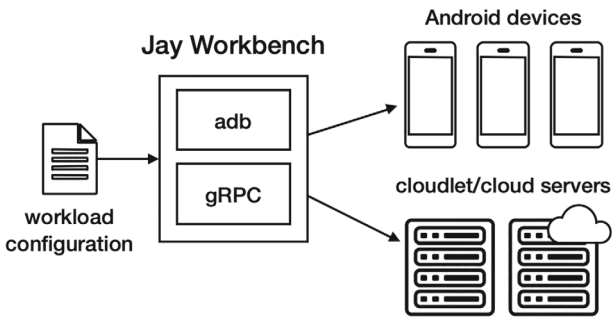

FIGURE 7 The JAY Workbench.

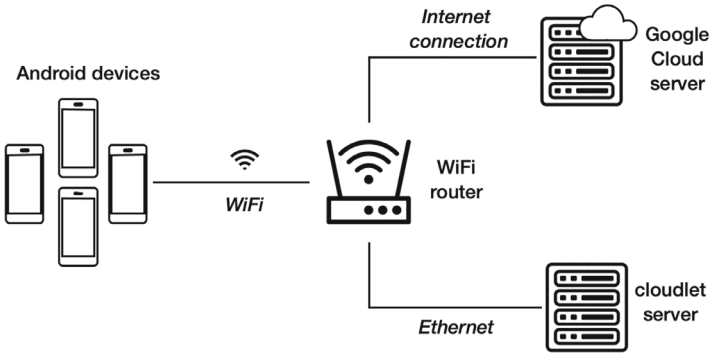

FIGURE 8 Experimental setup for case-study experiments.

In the first scenario,[11] we considered the hybrid cloud topology as shown in Figure 8. It featured 4 to 8 Nexus 9 mobile devices running Android 7, a cloudlet server connected to the local network via a gigabit Ethernet connection, and a Google Cloud server hosted at the `europe-west2-c` Google data center located in London. An Android app that receives images as input and produces reports of the objects detected in them using a machine learning model, more precisely the `ssd_mobilenet_v1_fpn_coco` MobileNet model variant[21] made available by TensorFlow.[22] The computation was performed for three different image datasets, with images of varying resolution/size taken from the UltraEye dataset.[23]

The experiments involved many variations of the cloud topology in which all or just part of the components were used. In all of these, the object detection jobs were only fired by mobile devices, while the cloudlet and cloud servers only acted as workers. For each cloud topology, various parameters were varied to better understand the behavior of the system, namely: number of mobile devices used, number of devices generating jobs, job release rate (defining a Poisson process), job granularity (controlled by using different image resolutions as input to the object recognition jobs), and offloading strategies. Each variation has a corresponding workload specification for automated execution using the JAY Workbench.

The fragment of such a workload specification is shown in Listing 6. As shown, the Workbench reads a text file containing configuration sections, where each section contains a set of key-value pairs that define a workload specification. In the example, the `ESTIMATED_TIME_SCHEDULER_CONFIG` section specifies a workload to be repeated five times (as set by the `Repetitions` parameter), with each iteration lasting 240 s (`Duration`) and using an offloading strategy (`Strategy`) set to `EstimatedTimeScheduler` (the internal identifier of the TMIN strategy discussed back in Section 2.2). Furthermore, the workload uses 8 Android devices (`Devices`), all of which (`Producers`) generate jobs through a Poisson process with 1 job released (`GenerationRateRequests`) every 10 s (`GenerationRateSeconds`) on average. Every device generates jobs and is also able to execute them, along with a cloudlet server (`Cloudlets`) at IP address `127.0.0.1` (the "local host," i.e., the same host as that used to run the Workbench in this case), and a cloud server (`Clouds`) in the Google Cloud namespace `hyrax/europe-west1-b` that is reachable through the `odcloud.duckdns.org` domain name (a dynamic DNS address used for convenience).

```
[ESTIMATED_TIME_SCHEDULER_CONFIG]
Repetitions = 5
Duration = 240
Strategy = EstimatedTimeScheduler
Devices = 8
Producers = 8
GenerationRateRequests = 1
GenerationRateSeconds = 10
Cloudlets = 127.0.0.1
Clouds = hyrax/europe-west1-b/odcloud.duckdns.org
...
```

Listing 6: Workload specification for a three-tier cloud topology.

Subsequently, to the first case study, JAY developments allowed us to consider energy consumption as a metric to be considered for offloading requests, and jobs with explicitly set soft real-time deadlines. In a second case study,[12] we experimented with a 2-tier topology formed by an edge network of heterogeneous Android devices and a cloudlet server, using the same TensorFlow-based object detection application. The cloud tier was not considered due to the inherent impossibility of measuring the energy consumption of a host in a Google Cloud data center. On the other hand, the scenario is richer compared to the first case study in terms of the use of heterogeneous Android devices (e.g., in terms of computational power, battery, memory, or manufacturer), and the fact that energy consumption was measured continuously by the JAY profiler for these devices and also for the cloudlet server. Offloading strategies can then be defined taking into account not only estimates for job completion time but also energy consumption, plus the deadline associated with jobs. For instance, we considered energy-aware strategies such as HYBRID (discussed earlier in Section 2.2), which offloads a job to the instance that is estimated to consume the least amount of energy amongst those whose completion times are estimated to be deadline-compliant.

```
[HYBRID_SCHEDULER_CONFIG]
Strategy = HybridScheduler
TaskDeadline = 9
Duration = 600
GenerationRateRequests = 5
GenerationRateSeconds = 9
Cloudlets = 127.0.0.1
Devices = 5
Producers = 1
PowerDevices = False
MinBattery = 50
...
```

Listing 7: Workload specification with settings for energy and deadline awareness

The workload specifications used by the Workbench reflect the extra parameterization required for the second case study. An example is given in Listing 7. As shown, we have a configuration where the HYBRID offloading strategy (`HybridScheduler`) is employed. Jobs are characterized by a deadline (TaskDeadline) in addition to other parameters discussed earlier related to job generation (`Duration`, `GenerationRateRequests` and `GenerationRateSeconds`). Finally, cloudlet (`Cloudlets`) and Android device (`Devices`, `Producers`) settings are as before, but supplemented by parameters related to battery restrictions: all Android devices are required to not be charging their batteries (`PowerDevices`) and the battery level (`MinBattery`) is required to be at least 50% at the start of the workload execution.

6 | RELATED WORK

In this section, we provide an overview of related work. For an extensive review and comparative discussion to JAY refer to References 19 and 12.

The idea of offloading computations dates back to the early days of mobile devices, well before the advent of smartphones. Limited hardware resources and battery capacity were the chief concerns[24] that researchers naturally tried to handle at the time. With the simultaneous emergence of the smartphone, the cloud computing paradigm,[25] and increasingly fast Internet connections, the MCC[26] paradigm naturally emerged, tethering mobile device applications onto servers in cloud infrastructures. Offloading computation can potentially reduce both battery consumption for mobile devices and the latency of computations. For this, there are ample resources available at cloud data centers. They have various degrees of flexibility such as the possibility of deploying and tearing down resources on the fly, the use of custom computing environments through virtualization, the availability of specialized hardware, and the interaction with a myriad of cloud services for storage and computation. Relevant systems include COSMOS,[27] Cuckoo,[28] MAUI,[29] Phone2Cloud,[30] ThinkAir,[31] ULOOF[32] and, including support for IoT-devices, PMCO.[33]

Variable latency and intermittent connections can be a problem for MCC, though. The computation-communication trade-off may be especially hard to get right, given that on one hand, computationally powerful servers may be available for computation in a centralized cloud, but on the other hand communication latency may be high due to a general-purpose Internet connection. A number of proposals emerged enabling closer-to-the-edge computing, mitigating network overheads, lead up to edge clouds in different forms and designations. For instance, the original concept of cyber-foraging[34] proposed dedicated servers placed in the network vicinity of client devices and found their way into offloading systems for mobile applications, for example, Reference 35. Another popular concept is that of cloudlets,[4] small-scale cloud data centers acting as a middle tier at the edge of the network between mobile devices and centralized cloud infrastructures. With mobile but also IoT applications in mind, fog computing[36] tries to integrate resources for computation and storage in close integration with the network fabric. The distinction between all these approaches and the use of terms is not without some ambiguity. Depending on the authors, the terminology can be contradictory, for example, in regard to the difference between edge and fog computing. Hence, we use the general term edge cloud. Some relevant systems include AIOLOS,[35] EdgeReduce,[37] mePaaS,[38] and Scavenger.[39]

Meanwhile, over the last two decades, mobile devices evolved from thin clients to computationally powerful devices like smartphones and tablets. These incorporate relatively powerful multi-core processors, several gigabytes of RAM and storage. They also support multiple types of network connectivity, including in particular D2D communication technologies like WiFi-Direct or Bluetooth that are enablers of proximity-aware applications with very limited or even absent network infrastructure. For these applications, the computation must really happen at the edge. This context led to the consideration of mobile edge clouds,[5,40,41] where nearby devices form ad-hoc networks, via opportunistic D2D communications or via a local network, to form a pool of crowd-sourced computing resources. As with other cloud architectures, computation offloading may aid in reducing latency and energy by moving computation onto nearby devices that can be faster or more energy-efficient. Among offloading systems, a special kind of MECs worth mentioning is known as Femtoclouds.[42] In a Femtocloud, mobile devices form a worker pool to execute jobs offloaded from an external source, enabling for instance applications in the realm of volunteer computing or mobile crowd-sensing. Some relevant systems are Honeybee,[43] Synergy,[44] CWC,[45] and RAMOS.[46]

Applications that make a hybrid use of multiple-tier clouds are emerging.[47-50] The inherent goal is to make the best use of heterogeneous resources for computation, storage, and communication at each tier. The motivating trade-offs are very much the same as discussed above, but the consideration of multiple tiers adds flexibility to deal with issues such as the movement of data to/from the edge, the opportunistic access to communications and edge or cloud resources, including IoT devices, and a dynamic flow of computation from/to the edge according to the congestion level of upper cloud tiers that in turn have an impact on latency or other QoS characteristics. Some relevant systems are mCloud[51] and Drop Computing.[52,53]

The main distinctive traits of JAY are that it is configurable in terms of target cloud architecture and scheduler operation, and can be employed for automatic and reproducible experiments. Thanks to a flexible design, each JAY host may act as a scheduler of jobs (spawns and offloads jobs), a worker for jobs (performs the associated computation), or simultaneously play both of these roles. Moreover, these hosts can be deployed at distinct cloud tiers. One can thus use JAY for offloading over manifold cloud architectures. Moreover, JAY supports per-device schedulers for offloading decisions or centralized ones, as in a Femtocloud setting. This type of flexibility is only partially comparable to (only a few) systems that work over hybrid cloud architectures. All these aspects can be exercised automatically in reproducible experiments using the JAY Workbench.

The offloading strategies we instantiate and evaluate in JAY are partially illustrative of comparable time and/or energy-aware approaches found in other systems. However, JAY is not bound to any particular approach since offloading strategies are configurable, and there is a general design for monitoring the runtime state information. JAY supports job deadlines, as well as a small portion of the offloading systems we surveyed. However, in some systems that do support deadlines, this feature is tied to fault-tolerance mechanisms rather than a QoS factor that may directly influence offloading decisions. On the flip side, in its current form JAY does not have any built-in fault-tolerance mechanisms (if required, these need to be implemented at the application level).

Finally, JAY only supports single-job scheduling granularity, a characteristic that is more in line with the on-the-fly offloading of independent jobs, as seen in most systems discussed. In contrast, multiple-job granularity is usually associated with the use of a centralized scheduler, a Femtocloud architecture, or a computation model that embodies parallelism.

7 | CONCLUSIONS

We presented JAY, a software framework for adaptive computation offloading in hybrid edge clouds that allows the seamless construction and realization of experiments for studying the performance of mobile applications in hybrid clouds. A system profiler, present at every instance of JAY, gathers runtime information that allows state-sensitive, dynamic, offloading policies to be implemented and systematically tested over a parametric space. In previous work[11,12] this capability was used to study performance and energy tradeoffs with respect to parameters like job generation rate, deadlines, and size; overheads due to job computation, network transmission, and associated energy consumption, and; the type of cloud environment, MEC without infrastructural support in the base case, and multi-tier hybrid clouds that also include cloudlet and/or traditional cloud servers.

There are several issues that we consider worth pursuing in future work on JAY.

The first is device churn, that is, the fact that devices may leave and enter a mobile edge cloud environment over time, for instance, due to device mobility or intermittent network connections. Churn may cause jobs to be aborted and MEC to be dismantled. To tackle churn, fault tolerance mechanisms would need to be introduced. Job check-pointing (the ability to save a resumable state of computation for an active job) coupled with migration may allow a job to be resumed in part or in full. When jobs are pending, migration can also leverage the addition of newly arrived devices. Moreover, network formation algorithms at the mobile edge cloud level also play a role in this regard.

A second issue relates to "hierarchy." While JAY works over multi-tier hybrid clouds and is able to differentiate between hosts and make appropriate offloading decisions through runtime profiling information, it still has a "flat" view of the hosts available for computation. Explicit notions of hierarchy and host groups can be relevant. MEC may form dynamic groups enabled by device-to-device communication links, with only one device acting as a "bridge" to upper cloud tiers. Also, cloudlet or cloud servers are often organized in clusters whose size may auto-scale dynamically according to the volume of requests.

Finally runtime resource awareness can also be improved in JAY. In particular, this would be important for energy consumption estimates which, as noted in our experimental results, may sometimes have significant errors. Other aspects include the association of monetary costs due to cloud server offloading or network data through cellular networks, in addition to the ones already considered by JAY. Finally, the available mix of standard CPU, GPU and AI cores at all levels of the network topology—mobile device, cloudlet, and infrastructure cloud—should also be taken into consideration when making job offloading decisions. Data placement awareness may also be relevant for jobs which consume data that may be stored at distinct hosts and/or tiers in the cloud, hence the interplay between computation and data offloading can potentially play a key role. Overall, increased resource awareness may enlarge the type of offloading strategies supported by JAY and enhance their effectiveness.

AUTHOR CONTRIBUTIONS

Joaquim Silva: Conceptualization; investigation; software; validation, experimentation. **Eduardo R. B. Marques**: Conceptualization, software; validation; experimentation, writing-review and editing. **Luís M. B. Lopes**: Conceptualization; validation; writing-review and editing. **Fernando M. B. Silva**: conceptualization, validation; writing-review.

ACKNOWLEDGMENTS

This work was partially funded by projects SafeCities (POCI-01-0247-FEDER-041435), Augmanity (POCI-01-0247-FEDER-046103), both through COMPETE 2020 and Portugal 2020, and by project UIDB/50014/2020 from the Portuguese funding agency, FCT.

DATA AVAILABILITY STATEMENT

Data sharing is not applicable to this article as no new data were created or analyzed in this study.

ORCID

Eduardo R. B. Marques https://orcid.org/0000-0002-6980-6868
Luís M. B. Lopes https://orcid.org/0000-0001-8273-1357
Fernando M. A. Silva https://orcid.org/0000-0001-8411-7094

REFERENCES

1. Ericsson Mobility Report, 2021. https://www.ericsson.com/en/reports-and-papers/mobility-report/reports/november-2021
2. Liu F, Shu P, Jin H, et al. Gearing resource-poor mobile devices with powerful clouds: architectures, challenges, and applications. *IEEE Wirel Commun*. 2013;20(3):14-22.
3. Erman J, Ramakrishnan KK. Understanding the super-sized traffic of the super bowl. *Proceedings of the 2013 Conference on Internet Measurement Conference*. ACM; 2013:353-360.
4. Satyanarayanan M, Bahl P, Cáceres R, Davies N. The case for VM-based cloudlets in mobile computing. *IEEE Pervasive Comput*. 2009;8(4):14-23.
5. Drolia U, Martins R, Tan J, et al. The case for mobile edge-clouds. *2013 IEEE 10th International Conference on Ubiquitous Intelligence and Computing and 2013 IEEE 10th International Conference on Autonomic and Trusted Computing*. IEEE; 2013:209-215.
6. Bland A. FireChat—the messaging app that's powering the Hong Kong protests. *The Guardian*. 2014.
7. Cohen N. Hong Kong protests propel FireChat phone-to-phone app. *The New York Times*. 2014.
8. Albrecht MR, Blasco J, Jensen RB, Mareková L. Mesh messaging in large-scale protests: breaking Bridgefy. In: Paterson KG, ed. *Topics in Cryptology—CT-RSA 2021*. Springer; 2021:375-398.
9. Potkin F, Pang J. Offline message app downloaded over million times after Myanmar coup. *Reuters*. 2021.
10. Hu YC, Patel M, Sabella D, Sprecher N, Young V. *ETSI White Paper #11 Mobile Edge Computing - A key technology towards 5G*. ETSI White Paper; 2015.
11. Silva J, Marques ERB, Lopes LMB, Silva FMA. Jay: adaptive computation offloading for hybrid cloud environments. *2020 Fifth International Conference on Fog and Mobile Edge Computing*. IEEE; 2020:54-61.
12. Silva J, Marques ERB, Lopes LMB, Silva FMA. Energy-aware adaptive offloading of soft real-time jobs in mobile edge clouds. *J Cloud Comput*. 2021;10(1):38.
13. Silva J. The JAY framework; 2020. https://github.com/jqmmes/Jay
14. Silva J. The JAY workbench; 2020. https://github.com/jqmmes/JayWorkBench
15. Kotlin Programming Language. https://kotlinlang.org/ 2011
16. Peitgen HO, Saupe D, Fisher Y, et al. *The Science of Fractal Images*. Springer; 1988.
17. gRPC. A high performance, open source universal RPC framework; 2016. https://grpc.io
18. ProtocolBuffers. https://developers.google.com/protocol-buffers/ 2008
19. Silva J. *Adaptative Computation Offloading in Mobile Egde Clouds*. Ph.D. thesis. Faculty of Sciences, University of Porto; 2021. https://repositorio-aberto.up.pt/handle/10216/139189
20. Android Debug Bridge. 2007. https://developer.android.com/studio/command-line/adb
21. Howard A, Zhu M, Chen B, et al. MobileNets: efficient convolutional neural networks for mobile vision applications, 2017. https://arxiv.org/abs/1704.0486
22. Huang J, Rathod V, Sun C, et al. Speed/accuracy trade-offs for modern convolutional object detectors. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE; 2017:3296-3297. https://github.com/tensorflow/models/blob/master/research/object_detection/
23. Nemoto H, Hanhart P, Korshunov P, Ebrahimi T. Ultra-eye: UHD and HD images eye tracking dataset. *2014 Sixth International Workshop on Quality of Multimedia Experience*. IEEE; 2014:39-40.
24. Satyanarayanan M. Fundamental challenges in mobile computing. *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*. ACM; 1996:1-7.
25. Vaquero LM, Rodero-Merino L, Caceres J, Lindner M. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Comput Commun Rev*. 2009;39(1):50-55.
26. Fernando N, Loke SW, Rahayu W. Mobile cloud computing: a survey. *Future Gener Comput Syst*. 2013;29(1):84-106.
27. Shi C, Habak K, Pandurangan P, Ammar M, Naik M, Zegura E. COSMOS: Computation Offloading as a Service for Mobile Devices. *Proceedings of the 15th ACM International Symposium on Mobile ad hoc Networking and Computing*. ACM; 2014:287-296.
28. Kemp R, Palmer N, Kielmann T, Bal H. Cuckoo: a computation offloading framework for smartphones. In: Gris M, Yang G, eds. *Mobile Computing, Applications, and Services*. Springer; 2012:59-79.
29. Cuervoy E, Balasubramanian A, Cho DK, et al. MAUI: making smartphones last longer with code offload. *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*. ACM; 2010:49-62.
30. Xia F, Ding F, Li J, Kong X, Yang LT, Ma J. Phone2Cloud: exploiting computation offloading for energy saving on smartphones in mobile cloud computing. *Inf Syst Front*. 2014;16(1):95-111.
31. Kosta S, Aucinas A, Hui P, Mortier R, Zhang X. ThinkAir: dynamic resource allocation and parallel execution in the cloud for mobile code offloading. *2012 Proceedings IEEE INFOCOM*. IEEE; 2012:945-953.
32. Neto JLD, Yu SY, Macedo DF, Nogueira JMS, Langar R, Secci S. ULOOF: a user level online offloading framework for mobile edge computing. *IEEE Trans Mob Comput*. 2018;17(11):2660-2674.
33. Yousafzai A, Yaqoob I, Imran M, Gani A, Md Noor R. Process migration-based computational offloading framework for IoT-supported mobile edge/cloud computing. *IEEE Internet Things J*. 2020;7(5):4171-4182.
34. Balan R, Flinn J, Satyanarayanan M, Sinnamohideen S, Yang HII. The case for cyber foraging. *ACM SIGOPS European Workshop 2002*. ACM; 2002:87-92.
35. Verbelen T, Simoens P, De Turck F, Dhoedt B. AIOLOS: middleware for improving mobile application performance through cyber foraging. *J Syst Softw*. 2012;85(11):2629-2639.

36. Bonomi F, Milito R, Zhu J, Addepalli S. Fog computing and its role in the Internet of Things. *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. ACM; 2012:13-15.
37. Pamboris A. *Mobile Code Offloading for Multiple Resources*. Ph.D. thesis. Imperial College London; 2013.
38. Liyanage M, Chang C, Srirama SN. mePaaS: mobile-embedded platform as a service for distributing fog computing to edge nodes. *2016 17th International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE; 2016:73-80.
39. Kristensen MD. Scavenger: transparent development of efficient cyber foraging applications. *2010 IEEE International Conference on Pervasive Computing and Communications*. IEEE; 2010:217-226.
40. Rodrigues J, Marques ERB, Lopes LMB, Silva FMA. Towards a middleware for mobile edge-cloud applications. *Proceedings of the 2nd Workshop on Middleware for Edge Clouds & Cloudlets*. Vol 1. ACM; 2017:1-6.
41. Rodrigues JF. *A Middleware for Mobile Edge-Cloud Applications*. Ph.D. thesis. Faculty of Sciences, University of Porto; 2019. https://repositorio-aberto.up.pt/handle/10216/118307
42. Habak K, Ammar M, Harras KA, Zegura E. Femto clouds: leveraging mobile devices to provide cloud service at the edge. *2015 IEEE 8th International Conference on Cloud Computing*. IEEE; 2015:9-16.
43. Fernando N, Loke SW, Rahayu W. Honeybee: a programming framework for mobile crowd computing. *Mobile and Ubiquitous Systems: Computing, Networking, and Services*. Springer; 2013:224-236.
44. Kharbanda H, Krishnan M, Campbell RH. Synergy: a middleware for energy conservation in mobile devices. *2012 IEEE International Conference on Cluster Computing*. IEEE; 2012:54-62.
45. Arslan MY, Singh I, Singh S, Madhyastha HV, Sundaresan K, Krishnamurthy SV. Computing while charging: building a distributed computing infrastructure using smartphones. *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*. ACM; 2012:193-204.
46. Gedawy H, Habak K, Harras KA, Hamdi M. RAMOS: a resource-aware multi-objective system for edge computing. *IEEE Trans Mob Comput*. 2021;20(8):2654-2670.
47. Garcia M, Rodrigues J, Silva J, Marques ERB, Lopes L. Ramble: opportunistic crowdsourcing of user-generated data using mobile edge clouds. *2020 Fifth International Conference on Fog and Mobile Edge Computing*. IEEE; 2020:172-179.
48. Dreibholz T, Mazumdar S, Zahid F, Taherkordi A, Gran EG. Mobile edge as part of the multi-cloud ecosystem: a performance study. *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*. IEEE; 2019:59-66.
49. Zhang Q, Zhang Q, Shi W, Zhong H. Firework: data processing and sharing for hybrid cloud-edge analytics. *IEEE Trans Parallel Distrib Syst*. 2018;29(9):2004-2017.
50. Liu D, Chen X, Zhou Z, Ling Q. HierTrain: fast hierarchical edge AI learning with hybrid parallelism in mobile-edge-cloud computing. *IEEE Open J Commun Soc*. 2020;1:634-645.
51. Zhou B, Dastjerdi AV, Calheiros RN, Srirama SN, Buyya R. MCloud: a context-aware offloading framework for heterogeneous mobile cloud. *IEEE Trans Serv Comput*. 2017;10(5):797-810.
52. Marin RC, Gherghina-Pestrea A, Timisica AFR, Ciobanu RI, Dobre C. Device to device collaboration for mobile clouds in drop computing. *2019 IEEE International Conference on Pervasive Computing and Communications Workshops*. IEEE; 2019:298-303.
53. Ciobanu RI, Negru C, Pop F, Dobre C, Mavromoustakis CX, Mastorakis G. Drop computing: ad-hoc dynamic collaborative computing. *Future Gener Comput Syst*. 2019;92:889-899.

How to cite this article: Silva J, Marques ERB, Lopes LMB, Silva FMA. JAY: A software framework for prototyping and evaluating offloading applications in hybrid edge clouds. *Softw Pract Exper*. 2023;1-19. doi: 10.1002/spe.3231