Resilient Execution of Data-triggered Applications on Edge, Fog and Cloud Resources

Prateeksha Varshney[†,⋆], Shriram Ramesh[‡,⋆], Shayal Chhabra[†,⋆], Aakash Khochare[*] and Yogesh Simmhan[*]
[*]Department of Computational and Data Sciences, Indian Institute of Science, Bangalore 560012, India
[†] Microsoft India R&D Pvt. Ltd., India
[‡] Wells Fargo International Solutions Pvt. Ltd., India
Email: prateeksha.varshney@microsoft.com, shriramr@alum.iisc.ac.in, shachhab@microsoft.com,
aakhochare@iisc.ac.in, simmhan@iisc.ac.in

2022 22nd International Symposium on Cluster, Cloud and Internet Computing (CCGrid) | 978-1-6654-9956-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/CCGRID54584.2022.00057

***Abstract*—Internet of Things (IoT) is leading to the pervasive availability of streaming data about the physical world, coupled with edge computing infrastructure deployed as part of smart cities and 5G rollout. These constrained, less reliable but cheap resources are complemented by fog resources that offer federated management and accelerated computing, and pay-as-you-go cloud resources. There is a lack of intuitive means to deploy application pipelines to consume such diverse streams, and to execute them reliably on edge and fog resources. We propose an innovative application model to declaratively specify queries to match streams of micro-batch data from stream sources and trigger the distributed execution of data pipelines. We also design a resilient scheduling strategy using advanced reservation on reliable fogs to guarantee dataflow completion within a deadline while minimizing the execution cost. Our detailed experiments on over 100 virtual IoT resources and for $\approx 10k$ task executions, with comparison against baseline scheduling strategies, illustrates the cost-effectiveness, resilience and scalability of our framework.**

I. Introduction

The Internet of Things (IoT) is leading to large-scale deployments of sensing, actuation and computing devices at the edge of the network as part of the physical infrastructure. Sensors like smart power meters, pollution monitors, and surveillance cameras are increasingly part of smart cities [1]. Continuous analytics over observations streaming from such sensors enable efficient utility management, interventions for health and safety, and intelligent transportation [2].

Contemporary IoT and smart city applications tend to be *tightly-bound* to consume specific sensor data sources. E.g., a *power utility application* may monitor the *kWh power load* from sensors `cds26kwh` and `cds74kwh` in the *IISc campus* neighborhood, and initiate demand curtailment if the load exceeds a threshold during the peak periods of *9AM–7PM* [3]. However, given the vast trove of public observation streams that change over time, such applications are more useful if they can specify "what" data streams they wish to consume and "when" they wish to consume them, rather than "which" specific sensor streams they should consume. E.g., rather than statically bind the power utility application above to the kWh sensor streams `cds26kwh` and `cds74kwh`, we should instead be able to state that the application should trigger for all *kWh power streams* in the *IISc campus spatial region* and during the *9AM–7PM time period*. So, the developer should not need to precisely know the deployed sensors and statically bind to their stream endpoints, but allow for their *automated discovery and use, based on semantic needs*. There has been recent interest in using declarative queries over IoT event streams to trigger such applications [4].

Another opportunity for such IoT applications is the availability of *edge computing* resources as part of the infrastructure. Sensor deployments are typically connected through edge gateways such as Raspberry Pi which offer non-trivial compute resources. Rather than use them to just move data from the sensors to the Cloud for processing, their captive compute capacity can be used to host light-weight applications [5]. However, *reliability* is a challenge. Further, cities are also explicitly deploying *fog computing* resources with server-class or accelerated resources [6] to handle compute-heavy applications such as video surveillance, and allow *multi-tiered management* of edge devices [7]. Edge and fog are *cheap or free captive resources*, and can serve as the first-line of computing for smart city applications. However, their resource capacities need to be supplemented on-demand by pay-as-you-go cloud resources [8]. At the same time, *platforms* to ease the development and robust deployment of applications on edge, fog and cloud are still evolving [9].

In this paper, we leverage the twin-opportunities of *ubiquitous data streams* and *edge and fog computing resources* in smart cities, to ease the specification of declaratively-triggered applications over such streams, and execute them reliably on edge, fog and cloud resources. We make these contributions:

1) We propose a novel *trigger-based model* for dataflow execution based on *declarative queries* specified over the attributes of micro-batch streams generated from wide-area sensor sources (Sec. II, III).
2) We develop a unique *resilient decentralized scheduling strategy* for *dynamically instantiated* dataflows on unreliable edges, and reliable fog and cloud resources, using *advanced slot reservation* on the fog, while meeting the dataflow deadline and minimizing the cost (Sec. IV).
3) We offer *detailed experiments on 100+ emulated edge and fog resources* for realistic dataflows with $10k$ task executions to validate the lower cost, higher resilience and

[⋆] Based on work done as a graduate student at the Indian Institute of Science, Bangalore, India

978-1-6654-9956-9/22/$31.00 ©2022 IEEE
DOI 10.1109/CCGrid54584.2022.00057

scalability of our approach relative to baselines (Sec. V). In addition, we also discuss related works (Sec. VI) and offer our conclusions (Sec. VII).

II. System and Data Model

A. System Model

Edge and fog are emerging resource abstractions that complement the well-established cloud computing [8]. Given the diverse definitions for the edge and fog paradigms, we state our assumptions on their characteristics [10].

Edge resources are distributed across a metropolitan area network (MAN) using wired, cellular or *ad hoc* network connectivity. They are often co-located with sensors that serve as sources of data streams. Edges have low-end computing and memory capacity, e.g., a Raspberry Pi with a multi-core ARM processor and 1-2 GB RAM. These devices may be *unreliable*, e.g., due to mobility, intermittent network, on-field failures, energy constraints, etc. Typically, these are captive and free resources as part of a city infrastructure, but finite in number. They may use containers for application sandboxing.

Fog resources are also distributed across a MAN, but *reliable* and connected to a high-speed broadband or cellular network. They offer a workstation or server-class performance and optionally have accelerators. They are also captive, finite and as cheap as or costlier than the edge resources. They may use containers or hypervisors as the application environment.

Public cloud resources at data-centers are accessible over the Wide Area Network (WAN). They offer unlimited on-demand access to *reliable* virtual machines (VM) with server-grade CPUs and diverse capacities. Their pay-as-you-go pricing may be costlier or cheaper than fog resources.

Management. Since Edge devices are constrained, we assume that the edge and fog resources are organized hierarchically to ease their management. Specifically, the resource and application management for each edge is done by its *parent Fog*, and a parent fog and all its edge children form a *fog partition* [7]. This grouping may be based on network or spatial proximity, or other organizational domains. Any interaction with an edge, say to schedule a task or to transfer data, is only through its parent fog which serves as a gateway. We expect the network performance between edge–parent fog to be high, and from fog–fog and fog–cloud to be lower.

Definitions. Let $r_i^E \in \mathbb{R}^E$, $r_j^F \in \mathbb{R}^F$ and $r_k^C \in \mathbb{R}^C$ be the set of edge, fog and cloud resources, respectively, with $\mathbb{R} = \mathbb{R}^E \cup \mathbb{R}^F \cup \mathbb{R}^C$ as the set of all resources in the system. $C(r_i^F) \subset \mathbb{R}^E$ is the set of *edge children* for a parent fog r_i^F, at a given time. The set $\mathbb{R}^E$ can vary over time. ϵ is the *incremental time unit of billing* for a resource with a *price function* $\pi(r_x)$ that gives the fixed-price for the resource r_x for ϵ time units. The *performance scaling*, or compute speed, of a resource relative to a baseline (slowest) resource r_0 is given as the function $\rho(r_x)$, with $\rho(r_0) = 1$. So the time taken to execute a task on r_x will be $\frac{1}{\rho(r_x)}$ of the time taken for the task on r_0. This can be workload-specific. The network bandwidth between resources r_x and r_y is given by β_{xy} and the latency by λ_{xy}; $\beta = 0$ for resources that cannot reach each other. ϕ_{xy} is the monetary cost for unit data transfer between two resources. These coefficients allow the scheduler to take performance and price-aware decisions.

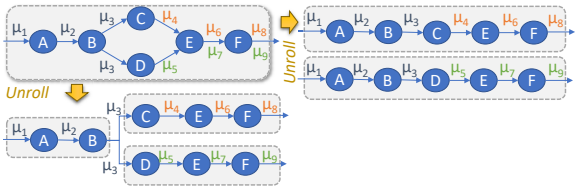


Figure 1: Unrolling of input DAG (top-left) into linear pipelines (top-right) or nested linear pipelines (bottom-left).

B. Streaming Micro-batch Data Model

The primary input source to our applications is data streaming from sensors, typically connected to an edge and sometimes a fog resource. Rather than target real-time event-based streaming applications, we instead consider the streaming *micro-batch model* [11] used by systems like Spark Streaming [5]. Here, a time or a count window of time-series observations is accumulated into a *micro-batch*, which forms the logical unit of execution, data movement and storage. These micro-batches themselves may be constantly generated from the sensor stream, forming a *stream of micro-batches*. This model offers better *throughput* since the data movement, scheduling and execution overheads are amortized across all events in that micro-batch, while bounding the latency overheads to the micro-batch size. The mechanism of acquiring data from the streams and forming micro-batches is outside the scope of this paper [11], [12].

Besides the *content* formed from the window of observations, a micro-batch also has several *spatio-temporal and domain attributes*. This metatata is crucial for automated discovery of such data sources and declarative binding of applications to them. A micro-batch μ_i is defined as the tuple:

$$\mu_i = \langle\, id, sid, \langle t_b, t_e\rangle, \langle lat, long\rangle, \langle key, val\rangle^*, size, content \,\rangle$$

where id uniquely identifies the micro-batch, sid identifies the source from which it was generated, $\langle t_b, t_e\rangle$ are the begin and end timestamps of the contents, $\langle lat, long\rangle$ are the spatial context for the data source, $\langle key, val\rangle^*$ are a set of domain-specific key–value metadata, $size$ is the length in bytes of the content, and $content$ has the actual micro-batched data.

E.g., the following micro-batch represents 8 observations accumulated from a temperature sensor `cds26temp` present in a campus IoT deployment on *Nov 15, 2021*. [1].

```
⟨2021-11-15T09:00:00, 2021-11-15T09:05:00⟩,
⟨13.0165, 77.5706⟩, [⟨units, C⟩, ⟨err, 0.05⟩], 39,
[27.5,27.5,27.6,27.7,27.7,27.7,27.6,27.7]⟩
```

III. Data-Triggered Application Model

We propose a *deadline-driven dataflow model* for application composition with a novel *triggering* of its execution based

on *declarative matching* of user queries with distributed micro-batches. We first specify the dataflow model followed by the query-based triggering approach.

A. Application Definition

We use the common *Directed Acyclic Graph (DAG)* as our application model. Users define their application as a DAG, $\mathcal{D} = (\mathbb{T}, \mathbb{E})$, with a set of vertex tasks $\tau_j \in \mathbb{T}$ and their dataflow edges $\langle \tau_i, \tau_j \rangle \in \mathbb{E} \subseteq \mathbb{T} \times \mathbb{T}$. Each task τ_j consumes a single micro-batch as input, and produces one micro-batch as output on completion. An edge $\langle \tau_i, \tau_j \rangle$ indicates a data dependency between tasks – an output micro-batch generated by the execution of τ_i will be used as input for the execution of task τ_j. Users also provide the scheduler with a *baseline execution time*, θ_j, for each task τ_j when executed on a base resource r_0, and the *expected size* of the output micro-batch from the task, α_j, based on prior benchmarking.

While a DAG offers substantial flexibility, most practical applications tend to be simple linear pipelines with stateless tasks. This is seen in the growing popularity of microservice tasks on the cloud defined as short-running and stateless Functions-as-a-Service (FaaS) and composed as a linear chain of event-driven computing, and executed using patterns such as Saga [13]. As a result, we make several practical simplifying assumptions. Tasks are *stateless* and a micro-batch is its unit of execution on a single resource. Aggregation can be done within events in a micro-batch but not across. In case a task executing a micro-batch fails due to a resource failure, it is *re-executed* for the same micro-batch on another resource.

The dataflow uses *interleave semantics*, i.e., if edges from two upstream tasks are incident on a downstream task, the latter will independently execute once for each output micro-batch from the two upstream tasks. This downstream path-independence lets us easily unroll the DAG into linear chains, converting a task-parallel to a data-parallel execution. E.g., in Fig. 1, the task E of the input DAG in the top-left executes twice for inputs μ_4 and μ_5 and the downstream execution path for their outputs micro-batch μ_6 and μ_7 are independent. This in equivalent to unrolling the DAG into two *sequential pipelines* shown on the top-right. A future optimization can split the common precursor sub-chains hierarchically (Fig. 1, bottom-left) to avoid duplicate execution of tasks, e.g., A and B. This approach eases the execution model while limiting penalties for DAGs without too many branches.

The user also specifies a *deadline* δ for the execution of the DAG for a single input micro-batch from its time of arrival. So all causal micro-batches generated for this input micro-batch should be executed by the DAG tasks within this deadline, i.e., all linear pipelines should complete by this deadline.

B. Application Triggering

Each dataflow's runtime execution is in the context of a single micro-batch. A key challenge is to identify micro-batch streams on which the dataflow should execute. Binding the dataflow to pre-defined stream endpoints requires *a priori* knowledge of all streams, which is difficult in an evolving IoT environment. Filtering micro-batches with certain attributes as part of an initial task of a DAG will still pay the cost for moving the micro-batch to the task even if it does not match. Similarly, publishing the micro-batch to a central event broker for filtering causes unnecessary data movement on the WAN.

Instead, we propose a novel *declarative model* for specifying the characteristics of the input micro-batch, based on its metadata, for a dataflow. Our runtime then automatically matches the generated micro-batches from *various streams across the edge and fog layers* to trigger a DAG execution on any micro-batch. Since the micro-batch attributes include spatial, temporal, sensor and domain attributes, this allows for fairly *complex patterns* to be defined easily by the user and matched at the *granularity of each micro-batch*. The application also implicitly *adapts to streams* entering and exiting as they are co-located with the edge resources. Users can always define a simple query that matches the sensor ID if they wish to statically bind their application to a specific sensor's stream. The matching is *light-weight, distributed across fogs and avoids data movement*, with a throughput of 1000s of micro-batch/sec on just an ARM-based fog.

As part of the DAG submission, users specify a declarative *filter query* $\mathcal{F} = \langle f_s, f_t, f_d \rangle$ where f_s, f_t and f_d are query predicates defined on the *spatial coordinates, time range* and *domain properties or sensor id* metadata of each micro-batch, respectively. This filter is active until the DAG is undeployed. The filters registered for the deployed DAGs are matched by the *query-matching engine*, as discussed next, against all available streams on all the edge and fog resources. So, the users focus on *what* to match rather than *where* the source is present. Any micro-batch matching the filter triggers the execution of an instance of the filter's DAG. The execution of each input micro-batch for a DAG is independent of other matched micro-batches. The same micro-batch can be matched by filters for different DAGs, triggering all of them. There is no ordering guarantee for different micro-batches for the same DAG, making micro-batch execution independent of their generation time. This allows micro-batches to make use of *task, data and pipeline parallelism* to the full extent. The use of micro-batches combined with low-latency matching and immediate deadline-aware scheduling for execution balances the throughput and the execution latency, while allowing maximal utilization of edge and fog resources.

C. Application Execution Engine

We have designed the **Clo**ud, **F**og and **E**dge **E**xecution (CoFEE) Engine for orchestrating the filter matching, DAG triggering, schedule planning, and resilient execution of the user provided dataflows on edge, fog and cloud resources. Here, we describe its execution model; in the next section, we propose heuristics for their deadline-aware and resilient scheduling. The architecture (Fig. 2) consists of edge, fog and cloud resources *(Workers)* that are available for task execution. We also have a separate cloud VM that runs a *Master service* for distributed coordination and holding light-weight state.

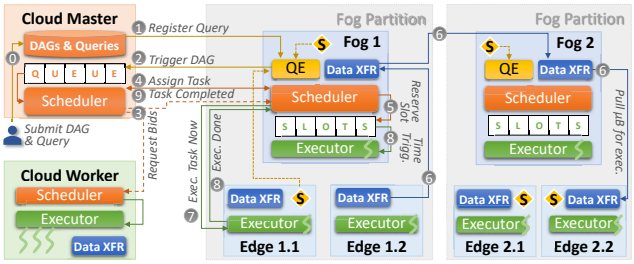


Figure 2: Architecture of CoFEE on Edge, Fog and Cloud

1) Federated Resource Discovery: The cloud and fog workers initially register with the Master. Edge resources bind with a parent fog when they join the system. The parent is chosen by querying the Master based on proximity, domain rules or prior knowledge. The fog schedules tasks and transfers data for edges in its partition. When an edge goes offline, it may explicitly inform its parent fog or implicitly drop off by failing to send heartbeats to it. This *federation* helps scale to a large number of edge and fog resources.

2) DAG Registration and Triggering: *Sensor services* (**S** in Fig. 2) generating micro-batch streams may be present on any edge or fog. *DAGs* are submitted by the user to the Master along with its *filter query* and *deadline* (Fig. 2 ⓪). The Master stores these DAG definitions locally, and registers the query with a light-weight *query-matching engine (QE)* running on each fog (①). The sensors on the edge or fog stream the *metadata attributes* for micro-batches they generate as events to the QE on the parent fog. Each event is ≈ 100 *bytes* in size. The QE matches the metadata events from sensors in this partition against the registered queries. We currently support equality matching for the domain attribute, and equality, intersecting or contains matching on the spatial and temporal ranges, and do an AND on the matching predicates. If a micro-batch's metadata matches a filter, it *triggers* the execution of the filter's DAG on that micro-batch with the Master (②). A micro-batch may match multiple DAGs and trigger multiple executions. By leveraging the fog for querying, we load balance the query cost across partitions and avoid the Master being a bottleneck.

3) DAG Scheduling: We discuss finer details of the scheduling heuristics in Sec. IV, but review the approach here. When a fog matches a micro-batch on its partition and triggers a DAG (②), the Master puts the DAG and the micro-batch ID on its scheduling *queue*. The Master then *unrolls* the DAG into linear pipelines and apportions the DAG's deadline as *sub-deadlines* for the tasks in each pipeline. The source task for each pipeline is first scheduled, followed by each subsequent task after the completion of its predecessor.

Scheduling a task follows a *inquiry, bid and select cycle*. The Master contacts a subset of under-loaded and cheap fogs with the task's baseline execution time, the sub-deadline and location of the input micro-batch (③). Each Fog's *Scheduler* responds with a bid if it can execute the task within the deadline on workers (edges or parent fog) in its partition, and returns the estimated cost based on its resource pricing. The Master also gets a bid from the on-demand Cloud worker. It then selects the cheapest fog partition or the cloud worker which can successfully execute the task, and assigns the task to it (④). When a task completes, the Master is informed of its output micro-batch ID (⑧). The Master then schedules the next task in the pipeline using a similar cycle. The scheduler logic of the Master is stateless and can scale concurrently, while its state and the queue can be replicated for resilience.

4) Data Transfer and Execution: A task that is scheduled on a fog partition for execution may run on an edge or the fog, based on the bid. If the input micro-batch to the task is on a different resource, we use *data transfer services* running on each resource to pull the input micro-batch and stage it locally on the target resource before execution (⑥). The data transfer services of fogs and cloud workers can talk to each other since they are usually on public networks, but edges on private networks use their parent fog's data service to exchange data. E.g., in Fig. 2, *Edge 2.2* pulls an input micro-batch for a local task from *Edge 1.2* through *Fog 2* and *Fog 1*.

5) Resilience: Without loss of generality, only one task executes on an edge or a fog at a time due to their limited resources, but the execution on a cloud worker can be concurrent as its capacity and pricing scales linearly. A task executes on an edge's *Executor* service *immediately* after the fog assigns it (⑦). But an execution on the fog may happen in the *future* (but before its sub-deadline) based on an *advanced slot reservation* decided by the fog (⑤, ⑧). The task completion is made tolerant to edge failures through slot reservation by the parent fog on itself for *fail-over and re-execution* (⑧), and by *caching* the task's input micro-batch (⑥). Once a task completes, the output micro-batch is stored on the executing resource, the parent fog informs the Master of the completion, and passes it the output micro-batch ID to trigger the next task in the pipeline (⑨). These are discussed next in Sec. IV.

6) Implementation: The CoFEE Engine implements all scheduling and runtime logic as *micro-services* in *Python* using *Google RPC* and *Protobuf*. This makes them light-weight and portable to execute even on low-end edge devices. Specifically, we have micro-services for the QE, the scheduler, the data transfer and the task executor, which variously run on edge, fog and cloud. All edge and fog services run within a *container* environment for sandboxing. For simplicity, the task binaries are pre-installed as part of container or VM image creation.

IV. Deadline-aware Resilient Scheduling

The scheduling heuristic that we propose optimizes for *(1) reducing the monetary cost of executing the DAG, (2) while meeting its deadline, and (3) being resilient to Edge failures*. Building upon the high-level objectives above, the specific scheduling strategies are discussed in detail next.

A. Inquiry and Selection of Resource upon Trigger

1) Trigger: When a micro-batch from an edge or fog sensor stream matches a filter query registered with the QE on the parent fog, it triggers the DAG on the Master with the micro-batch ID. The Master unrolls the corresponding DAG into multiple sequential pipelines of tasks.

The deadline for the entire DAG also applies to each pipeline. The *sub-deadline for each task* in the pipeline is calculated by distributing the DAG's deadline δ among the tasks, in proportion to the task's execution time contribution to the pipeline [14]. For a pipeline $\mathbb{P}$ having tasks $[\tau_1, \tau_2, ..., \tau_p]$, the sub-deadline σ_i of a task τ_i is: $\sigma_i = \frac{\theta_i}{\sum_{j=1}^{p} \theta_j} \times \delta$, where θ_j is the baseline execution time for a task τ_j.

2) Inquiry: On a trigger, the Master initiates the *inquiry phase* to find the cheapest fog partitions available to execute the *source task* for a pipeline. Since there may be 10–100s of fog partitions, broadcasting the inquiry to all fogs has a high overhead. But sending the request to more fogs will help discover the cheapest viable one. To balance between these, the Master sends an inquiry request to the *top-n* least-loaded and cheapest fog partitions. For this, each fog periodically reports its *top-k* longest free slots available to the Master. As discussed next, these free slots are a good proxy for the load on the fog partition since tasks running on the edge and scheduled on the fog will reserve these slots. From these, the Master picks fogs with free slots long enough to execute the task within its sub-deadline. Among them, it picks the *top-n* with the lowest resource cost for the parent fog. This increases the chance of getting a viable bid, and also prefers cheaper fogs.

The *inquiry request* is sent to these fogs with details of the source task τ_0 and input micro-batch, $\langle \tau_0, \theta_0, \sigma_0, \mu_x, \alpha_x, r_x \rangle$, which provide the task's ID, baseline execution time and sub-deadline, and the micro-batch's ID, size and the resource it is present on, based on the initial trigger. The Master waits for a pre-defined timeout period, t_{inq}, to receive the *bid* responses from these n Fogs.

3) Selection: After the timeout period expires or when all n fogs respond with a bid, whichever is earlier, the Master picks the viable bid that can complete the task within its deadline and has the cheapest cost, $\kappa_{min} = \min_n(\kappa)$. It sends that fog an *accept bid* message, asking it to proceed with the task execution; it also sends a *reject bid* message to the other viable fogs.

It is possible that none of the fog partitions submit a viable bid for a task inquiry because their edges or fog are busy until the task deadline. We implicitly always include a bid from the Cloud worker in our selection process since it is always available, on-demand. We first estimate if the Cloud VM can complete the task within the sub-deadline, considering the micro-batch transfer time and the task execution time, and if so, we estimate the cost for execution to form a bid. This is considered when selecting the cheapest bid from the fogs. So even if the resource cost for the cloud is cheaper than the fog, our scheduling heuristics will work. If none of the Fog partitions submit a bid and if the Cloud VM cannot finish before the deadline, this task and hence pipeline has failed.

B. Bidding by the Fog for an Inquiry

A fog r_i^F that receives an inquiry $\langle \tau_y, \theta_y, \sigma_y, \mu_x, \alpha_x, r_x \rangle$ checks if it can execute the task τ_y on the input micro-batch μ_x within the sub-deadline on any of its edges, $r_j^E \in C(r_i^F)$, or on itself. For this, it first checks these conditions on each idle edge in its partition not currently executing a task:

$$\omega_j + \frac{\theta_y}{\rho(r_i^F)} \leq \sigma_y \qquad \text{where} \quad \omega_j = t_{inq} + d_{xj} + \frac{\theta_y}{\rho(r_j^E)}$$

This tests if the time taken for: the inquiry and bid phase to complete, the data transfer d_{xj} from the current micro-batch location at r_x, the task execution on the edge and also the task re-execution on the fog parent *if* the edge fails, all together fall within the sub-deadline σ_y. Here, ω_j gives the *latest completion time* on the edge; if the edge fails to complete the task by this time, we will re-execute it on the fog. We use estimates of the inter-resource bandwidths β_{xi} and β_{ij} between the resource r_x hosting the micro-batch to this parent fog r_i^F and from this parent fog to the local edge r_j^E, and similarly their latencies λ_{xi} and λ_{ij}, combined with the micro-batch size α_x, to estimate the incoming micro-batch transfer time as $d_{xj} = \left(\lambda_{xi} + \frac{\alpha_x}{\beta_{xi}}\right) + \left(\lambda_{ji} + \frac{\alpha_x}{\beta_{ij}}\right)$. The task execution time is scaled by the relevant processing speed $\rho(\cdot)$ of the resources.

For every edge device $\widehat{r}_j^E$ which satisfies the above condition, the fog computes the *expected maximum cost*, κ_j, for running the task. This is the cost of executing the task on that edge, and the *probabilistic cost* of having to execute the task on the fog if the edge fails to complete it by the deadline.

$$\kappa_j = \left(k^E + \left\lceil \frac{\theta_y}{\rho(\widehat{r}_j^E)\cdot\epsilon} \right\rceil \cdot \pi(\widehat{r}_j^E)\right) + \left(k^F + P(\widehat{r}_j^E) \cdot \left\lceil \frac{\theta_y}{\rho(r_i^F)\cdot\epsilon} \right\rceil \cdot \pi(r_i^F)\right)$$

The first part is the cost on the edge, including *monetary cost for data transfer*, $k^E = \alpha_x \times (\phi_{xi} + \phi_{ij})$, and the *task execution cost* based on the execution time, the billing increment ϵ, and price of the edge $\pi(\cdot)$. The second part is the probabilistic cost on the fog, which is similar, but includes the probability of failure of the edge, given by $P(\widehat{r}_j^E)$. This probability can be a fixed value provided for the edge, or calculated from its *Mean Time Between Failure (MTBF)*. $k^F = \alpha_x \times \phi_{ji}$ is the cost to copy the input micro-batch to the parent fog from the edge – we always cache a copy of the input micro-batch on the parent fog to allow for re-execution if the edge fails. These micro-batch transfer times and costs are adjusted accordingly if it is already present in the local fog partition.

The fog returns the candidate edge with the cheapest expected maximum cost back to the Master. However, we still need to check if the parent fog has the capacity to re-execute the task if the edge fails. For this, we use a slot reservation strategy discussed next. If none of the edges can meet the deadline, or if the fog cannot reserve a future re-execution slot, we check if the fog can run the task directly and meet the sub-deadline; if so, we return its cost to the Master, and if not we return an empty bid to the Master.

C. Advanced Slot Reservation for Fogs

It may be possible to delay executing a task after its submission and still complete it within its sub-deadline. We use this intuition to enable re-execution of a task on the parent fog if it fails on an edge. As concurrent tasks can execute on different edges of the partition and may fail simultaneously, we must ensure that all of them can re-execute on the parent fog within their sub-deadline. Further, tasks may also be

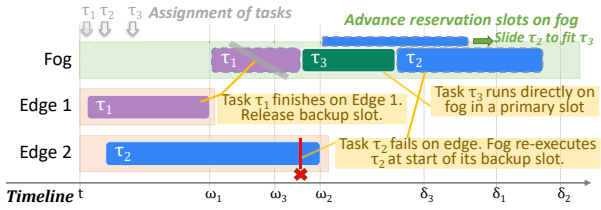


Figure 3: Task reservation and execution in a Fog partition

directly run on the fog. So we need to carefully plan the tasks scheduled on the fog to avoid over-allocation of its resources and guarantee the completion of tasks assigned to its partition.

We use the concept of *advanced slot reservation* on the fog to provide these guarantees [15]. Slots are *slices of resource time* within the fog's future timeline that can be reserved for specific tasks to execute. Slots reserved for re-executing failed tasks from the edge are *backup slots* that may or may not be used, while those reserved for running tasks directly on the fog are *primary slots* that will be used.

Once the fog identifies the candidate edges for an inquiry to run task τ_y, we sort them in increasing order of their κ cost. Incrementally, for each edge, $\widehat{r}_j^E$, we check the parent fog for contiguous free slots of length $\frac{\theta_y}{\rho(r_i^F)}$, starting from the latest completion time on the edge, ω_j, and till the sub-deadline σ_y. If so, the fog can re-execute the task on itself during this slot before the deadline even if the edge fails. We *reserve* these slots as backup slots for the task, and return the edge and its cost to the Master as a viable option.

To check for such contiguous slots rapidly, we maintain the free slots of the fog in an *interval tree* data structure and search for the *worst-fit free slot* that is viable. Say $[t, t']$ is the relative start and end time intervals for the largest free slot. We test if $\max(\omega_j, t) + \frac{\theta_y}{\rho(r_i^F)} \leq t'$ and $\max(\omega_j, t) + \frac{\theta_y}{\rho(r_i^F)} \leq \sigma_y$, i.e., can we complete the task re-execution before the interval's end time and before the task's sub-deadline. The space complexity for the tree is $\mathcal{O}(n)$ where n is the number of free or reserved slots, and the time complexity for most operations is $\mathcal{O}(\log n)$.

If contiguous slots that meet this task's requirement are not available, the fog tries to *reschedule existing slot reservations* to widen the contiguous free slots, while ensuring that the task sub-deadlines for these prior reservations are still met. Intuitively, we use a *defragmentation heuristic* that attempts to consolidate consecutive reserved slots to enlarge the free slot. First, we select the largest available free slot, say, $[t, t']$, between $[\omega_y, \sigma_y]$ and slide the slot reservation for some task τ_z that immediately *succeeds* it as far to the *future* as possible, without violating its sub-deadline, σ_z, or overlapping with its subsequent reservation. This will increase the t' for the free slot, and hence widen the free interval, $(t' - t)$. If this slot is still not large enough for τ_y, we attempt to slide the *preceding* reserved slot for a task τ_w as far to the *past* as possible without dropping below its edge completion time, ω_w, or overlapping with its preceding reservation. This will decrease t and widen the free slot. Lastly, we try to do both. This is repeated from the largest to the smallest free slot until one meets our needs.

If the slot reservation for a task is successful during an inquiry, the fog returns a viable bid to the Master. But this reservation is *temporary*. If the *bid is accepted* by the Master, it becomes permanent; if the *bid is rejected*, the slot reservation is canceled. Also, if a task completes on an edge, its backup slot reservation is removed and available for future requests.

Example. Fig 3 illustrates this reservation process. Initially all the edges are free. When an inquiry for task τ_1 comes with deadline δ_1, the fog selects the cheapest free edge, *Edge 1* for its execution from now till ω_1. The fog reserves a *backup slot* on itself from time range $[\omega_1, \omega_1 + t_1]$ for this task, where t_1 is the task execution duration on the fog. Similarly, task τ_2 that arrives next is assigned to *Edge 2* since Edge 1 is in use and Edge 2 is the next cheapest. It is initially reserved a backup slot on the fog from $[\omega_2, \omega_2 + t_2]$. Now when τ_3 arrives, all edges are busy and the fog tries to reserve a *primary slot* to directly run it. Since no slot is large enough to fit τ_3 before its deadline δ_3, we slide the slot for τ_2 to the right but before its deadline δ_2 to be able to allocate τ_3 a slot. When τ_1 completes execution on the edge, its backup slot on the fog is released.

D. Reliable Execution of the Task

When a task is assigned to the fog after the Master accepts its bid, the parent fog makes the slot reservation for the task permanent. If required, the fog initiates transfer of the input micro-batch from the host resource to the edge the task will execute on; the fog caches a copy on itself for task re-execution or for direct execution. If the task is assigned to an edge, the fog invokes the task on the edge. If the task completes successfully by its deadline, the fog notifies the Master the output micro-batch ID and deletes the backup slot.

A *timer* on the fog fires if we reach the start timestamp of a reserved slot. This can be a primary or a backup slot. If a primary, the fog invokes the relevant task on the cached input micro-batch to directly execute it within its sub-deadline and returns the output micro-batch ID to the Master. If a backup slot, the *edge has failed to complete the task execution* on the micro-batch before its sub-deadline, either because the edge failed or due to under-performance – a successful completion on the edge would have deleted this backup slot. The fog then re-executes this task using its cached input micro-batch, and responds back to the Master. Importantly, no active notification from the edge is required for re-execution if it fails.

E. Scheduling Downstream Tasks

The above steps are also applicable for successive tasks in a pipeline, with the difference being in how they are triggered. Once the source (or later) task completes execution, the edge, fog or cloud that executed it notifies the Master , and passes the output micro-batch ID and its location. This is equivalent to getting a trigger from the QE, except that we schedule the next task of an ongoing pipeline for the initial micro-batch. Note that multiple copies of a pipeline can be executing for the same DAG but on different input micro-batches. Once the last task for a pipeline has executed before its deadline, the

pipeline has completed; once all pipeline branches for a DAG have completed for an input micro-batch, the DAG execution has completed.

F. Discussion

Using the fog for backup slots allows us to use cheaper edge resources reliably without paying for the fog resource if it is not used. If the edge capacities are all used up, we schedule directly on the fog by reserving primary slots at any point before the task's sub-deadline, to avoid going to the cloud. If edges are highly reliable, the backup slots are rarely used; but their existence guarantees task completion. But the fog cannot use these backup slots for directly executing other tasks and this can lead to under-utilization. We can extend the heuristic to allow over-subscription of fog slots during reservations by a certain ratio $\chi = \frac{\text{allowed load}}{\text{available capacity}} > 1$, i.e., reserve the same backup slots for up to χ tasks. The choice of χ can be decided based on the edge reliability or dynamic conditions. This allows the fog to be fully utilized, and avoids running tasks on the Cloud. But the trade-off is that it increases the chance of a failed task from an edge also failing on the fog if its over-allocated backup slot was used by another edge task that simultaneously failed. The χ ratio helps the user decide this trade-off.

V. Experimental Evaluation

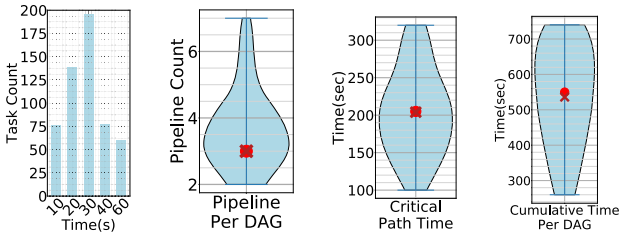
A. System Setup

We use the VIoLET IoT emulation Environment [16] to accurately mimic the behavior of an edge, fog and cloud deployment. VIoLET uses Docker containers hosted on VMs to replicate the compute and network performance of IoT resources. We configure 111 containers representing 100 *Raspberry Pi edges,* 5 *Jetson TX1 fogs and* 6 *16-core cloud resources*, along with one cloud VM running the Master. Each fog partition has one fog and 15–25 edges. These are hosted on 7 Azure D32 VMs, each with a 32-core Xeon CPU and $128GB$ RAM, and match the cumulative compute capacity of the 111 resources. The relative compute performance (ρ) of edge, fog and cloud are 1:8:50, based on real benchmarks.

The *per core-hour billing cost* for the cloud is set realistically to 10¢/hr, with the fog and edge being 10% and 20% cheaper, when normalized for performance. This gives the resource cost for edge as 0.167¢/hr, Fog as 1.467¢/hr. We use a 1 sec billing increment. The bandwidth and latency between Fog–Fog and Fog–Cloud is 100 $Mbps$/5 ms, and for Fog–Edge as 60 $Mbps$/1 ms. The bid timeout is set to $t_{inq} = 1$ sec and the number of fogs to inquire is $n = 2$.

B. Application Workload

We use 30 DAGs sourced from permutations of the RIoTBench IoT dataflow workload [17]. Each DAG has 2–7 unrolled pipelines (Fig. 4b). We use 8 synthetic tasks with *baseline execution time* of θ = 10–60 edge-secs, and their execution time on a resource linearly scales based on its performance scaling. These tasks are mapped to the RIoTBench DAG tasks with a Gaussian distribution (Fig. 4a). The median



(a) Task exec. time histogram (b) Pipelines per DAG (c) Critical path time (d) DAG exec. time

Figure 4: IoT Application Workload Characteristics.

critical path time for a DAG is 204 $secs$ and the total time to execute all its tasks is 537 $secs$ (≈ 9 min to run a DAG on an edge (Figs. 4c, 4d)). Synthetic micro-batches with sizes uniformly from 500–1500 KB are generated from virtual sensors on edges. Their attributes match the filter query's "hit-rate" (*micro-batches triggered/sec*, and hence load on the system) for each experiment.

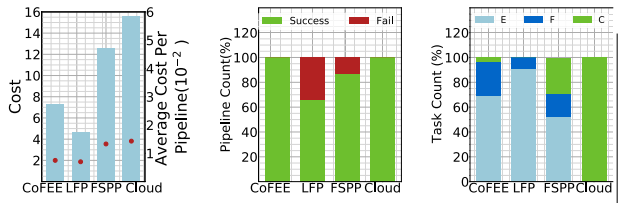
C. Baseline Scheduling Strategies

We compare our proposed scheduling strategy with three baseline algorithms to evaluate their costs and ability to complete the DAGs by the deadline.

1) Cloud-Only (CO) Baseline: In this simple algorithm [18], the edge and fog only generate micro-batches from sensors and transfer them. Every task that is triggered is executed only on the Cloud worker. Sufficient VM capacity is ensured to execute the peak number of concurrent tasks that are present. This makes it *fast and reliable but at a higher cost*. The micro-batch still needs to move from its source resource to the Cloud. This is reflected in the execution time and the resource cost.

2) Local Fog Partition (LFP) Baseline: This greedy algorithm executes all tasks in a DAG on the same fog partition containing the source micro-batch. Within the partition, it picks the cheapest available edge where the task can complete within its deadline. The parent fog is chosen only if all the edges are busy or cannot complete the task within the deadline, and the fog is free. There is no future slot reservations on the fog. A task (and pipeline) fails if the edge executing it fails or if inadequate resources are currently available in the fog partition for scheduling it. *This localizes the scheduling decision and reduces the data movement to the partition, but may be unreliable as the task deadline is not efficiently used.*

3) Fog Service Placement Problem (FSPP) Baseline: This adapts the deadline-aware application scheduling from [19]. Tasks arriving with deadlines on a fog partition's queue are accumulated in a 1 sec window, solved as an Integer Linear Programming (ILP) scheduling problem using IBM CPLEX, and placed on local edges, the fog, the cloud, or a neighboring fog partition. Triggered Triggered DAGs are uniformly spread across the fogs, unrolled, and their tasks placed on the fog queue when ready. Only one task executes on a resource at

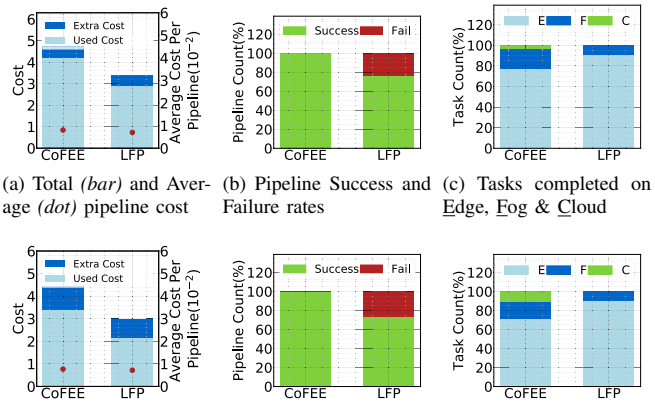


(a) Total Cost *(left bar)* & Avg. Cost for successful pipelines *(right dot)*

(b) Pipeline success rate *(green)* & failure *(red)* rate

(c) Ratio of triggered tasks scheduled on Edge, Fog & Cloud)

Figure 5: Performance of the schedulers with *reliable* edges



(a) Total *(bar)* and Average *(dot)* pipeline cost

(b) Pipeline Success and Failure rates

(c) Tasks completed on Edge, Fog & Cloud

(d) Total *(bar)* and Average *(dot)* pipeline cost

(e) Pipeline Success and Failure rates

(f) Tasks completed on Edge, Fog & Cloud

Figure 6: Performance of CoFEE and LFP on *unreliable edges*, with MTBF of 100 $mins$ (top row) and 40 $mins$ (bottom row).

a time, and the execution time on a resource depends on its performance scaling.

While CoFEE, CO and LFP actually trigger, schedule the DAGs and and execute their tasks for the micro-batch streams within VIoLET containers, we *simulate* this workload for the FSSP scheduler and estimate the task completion and costs.

D. Execution Cost Analysis with Reliable Edges

In these experiments, we evaluate the ability of CoFEE to complete all the DAGs for micro-batches they are triggered for, within the deadline and at a lower cost, compared to the baseline CO, LFP and FSPP schedulers. We deploy the 30 DAGs on the 100 edge, 5 fog and 6 cloud workers with the default pricing. Here, we assume that all resources including edge are *always reliable*. We configure the DAG filter queries such that the *cumulative micro-batch triggering rate* is 15 *micro-batches/min*, with a uniform probability on the edge source that generated the matching micro-batch and target DAG that it triggers. This translates to an *average application load* of 100% of the total edge and fog resource capacities, considering the basic scheduling overheads. We assign a *deadline* that is 110% of the critical path duration for each DAG when executing on edge resources, i.e., 10% extra. We allow CoFEE to *over-allocate* the slot reservations ($\chi > 1$) such that the *full fog capacity* can be used for primary slots, and it can support backup slots for the *cumulative edge capacity* in its partition. The experiments run for 20 $mins$, and a total of ≈ 2740 tasks are executed per run.

Figs. 5 show the various metrics for CoFEE and the baselines. The total monetary cost for executing all the triggered DAGs is shown in Fig. 5a, left Y axis bar. LFP has the smallest total cost, followed by CoFEE, FSPP and CO. LFP appears to be 35% cheaper than CoFEE. However, when we consider the success rate in Fig. 5b that shows the fraction of pipelines that completed within the deadline, LFP a 34.5% failure rate while CoFEE and CO successfully complete all their tasks. This reduced load due to failures leads to a proportionally lower total cost for LFP. FSPP is 19% cheaper than CO but almost twice as expensive as CoFEE, and exhibits a 13.5% failure rate. The cost per *successful* pipeline execution (Fig. 5a, right Y axis red dot) for CoFEE is similar to LFP and 50% cheaper than CO and FSPP, with the latter two comparable.

Fig. 5c shows the fraction of all triggered tasks that ran on each resource type for the three algorithms. CoFEE intelligently uses all three types – edge, fog and cloud – to offer 100% completion within the deadline and at a low cost. Only 3.3% of its tasks run on the cloud while 27.3% on the fog and the rest 69.4% on edge resources. As designed, the cloud worker is used only when the edge and fog do not have the capacity to complete the task within its deadline. There are no edge failures in this setup. Though the average count of DAGs triggered match the cumulative capacities of the edge and fog, the random generation of micro-batch causes load spikes which the cloud handles. While LFP lowers costs by only using the local edge and fog, its reliability is lower by not using the cloud, other fog partitions or future fog slots. Like CoFEE, FSPP also uses all three resource types but runs 28.7% of the tasks on the cloud vs. only 3.3% for CoFEE.

E. Application Resiliency Analysis with Unreliable Edges

One of the key benefits of CoFEE is its ability to schedule applications with a high degree of resilience, even when the edge resources are fault-prone. Its advanced slot reservation heuristics on reliable fogs help achieve this. In these experiments, we evaluate the effectiveness of CoFEE in successfully completing DAG executions when edges have a *low and high rate of failures*. We compare this against the LFP baseline; the CO baseline is not pertinent since it will always complete successfully on the cloud irrespective of the edge failures.

The experiment setup is similar to the above with some key differences. We use two configurations of MTBF for the Edge resources, 100 $mins$ (M100) and 40 $mins$ (M40). This means that with 100 edges and an experiment runtime of 20 $mins$, we expect 20 edge failures ($\approx 1/min$) for M100 and 50 edge failures ($\approx 2.5/min$) for M40. The edges fail *independently* based on a uniform probability across time that matches these MTBFs. We have a mean application load that is 100% of the cumulative capacities of the edge, which initially generates

a cumulative of 10 *micro-batches/min*. This is smaller than earlier, where the application load could also saturate the fog for direct execution. Now, we expect to use the parent fog for backup slots that will be used due to edge failures. Since the edges are the source of the micro-batch, the loss of an edge proportionally reduces the application load and the compute capacity of the system, maintaining the average load at $\approx 100\%$ of the available edges. We retain a DAG deadline of 110% of its critical path using only edge resources.

Fig. 6 plots the various metrics for these two scenarios; M100 is on the top row and M40 on the bottom row. In Figs. 6a and 6d, we report the resource cost to execute tasks that are part of pipelines that eventually complete successfully (light blue) and the extra cost for tasks that execute for pipelines that finally failed and hence wasted (dark blue).

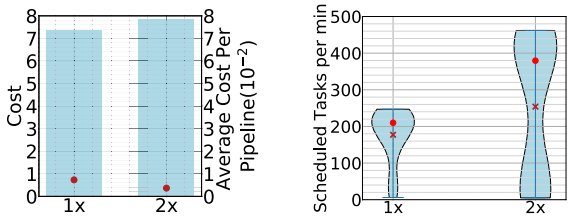
As before, LPF has a lower total cost for execution in both cases (Figs. 6a and 6d), explained by its large number of failures (Figs. 6b and 6e). The fraction of pipelines that fail for LFP has not increased from earlier even though edges fail in this setup. The failure rate for M40 is marginally higher than M100, 26.9% vs. 23.9%, and both are lower than 34.5% seen above. But unlike the previous setup where the application load was 100% of both edge and fog capacities, now it is only at 100% of the edge capacities. This lower load causes fewer failures. The failure rate difference between M100 and M40 is small. In its attempt to greedily schedule the tasks on a current free resource rather than use the deadline slack to defer the execution, LFP is often unable to find a free edge or fog. A small number of additional edge failures does not make this situation much worse. But the extra cost wasted by LFP on tasks of failed pipelines is higher for M40 than M100 since there is a greater chance of progress being lost for M40.

CoFEE is able to successfully complete *all DAGs* in the M100 setup, and all but 5 of the 580 pipelines even in the M40 setup (Figs. 6b and 6e). In the latter, three of the failures happen because the fog was over-booked and executing a primary task during the backup slot, when the edges failed. Two failures happened because the edge was selected for execution but failed before the execution was started.

We see from Figs. 6c and 6f that CoFEE makes greater use of cloud workers as the unreliable edges increase, running 4.1% and 11.3% of tasks on the cloud for M100 and M40. This partly explains the higher cost expended. As the failure rate increases, the edge cost spent on failed tasks also grows (Figs. 6a and 6d). $\approx 12\%$ and $\approx 24\%$ of the total cost is wasted for M100 and M40. However, unlike LFP, these tasks eventually succeed by re-executing on the fog. That said, the mean cost per pipeline remains close to LFP that exclusively schedules on only edge and fog resources.

F. Scalability Experiments

Lastly, we examine the ability for the CoFEE runtime and scheduler to scale with the rate at which the DAGs that are triggered for execution. This reflects both the light-weight nature of our micro-service based implementation and also the limited runtime overheads for the scheduling algorithm.



(a) Total *(bar)* and Average *(dot)* pipeline cost (b) Task Throughput per Minute

Figure 7: Scalability of CoFEE scheduler and runtime.

We use a workload identical to the reliable edge experiments in Sec.V-D, which we call $1\times$WL. We then double the rate at which the micro-batches are generated while halving the execution time per task in the 30 DAGs. This maintains the application computation load on the cumulative edge and fog resources at 100%, and yet doubles the number of tasks that are scheduled per second. We also increase the DAG deadline to 125% since some of the scheduling costs have a fixed time overhead. With shorter tasks, a larger fraction of the critical path time is used for these. This workload is called $2\times$WL.

Fig. 7a plots the total cost for running these two workloads using CoFEE (bar on left Y axis). While the actual compute load for both these workloads are identical the $2\times$WL workload has a marginally higher total execution cost than the $1\times$WL workload. The key reason for this is the billing granularity of 1 sec. For the smaller tasks, it is more likely that the billing round-up will cause excess payment for resources that are not used, and this accumulates across the 9154 tasks that are executed for $2\times$WL. The average per-pipeline cost (red dot on right Y axis) is half for $2\times$WL as that of $1\times$WL. This is as expected as the tasks are half as long. Fig. 7b shows a violin plot distribution of the number of tasks scheduled per minute by the Master. The median task scheduling throughput for $2\times$WL is about twice that of $1\times$WL at ≈ 6 $tasks/sec$. With higher rates of task generation, there is also more variability in the rate of tasks.

In summary, the scalability of the system is limited by DAG triggering and task scheduling. QE can match > 1000 micro-batch-queries per second, per fog, and weakly-scales with the fog and stream counts. The ≈ 360 tasks/min supported by our scheduler can improve by batching task scheduling requests to amortize the inquiry–bid–selection cost between cloud and fogs. These should allow us to scale to 1000s of streams.

VI. Related Work

There is an emerging body of work on scheduling of applications on the edge, fog and cloud ecosystem. Existing conceptual works deal with basic architectures, Application Programming Interfaces (APIs), and data communication [20]. In contrast, the question of how to effectively trigger, distribute and schedule streaming IoT applications across edge, fog and cloud resources has garnered less attention.

Publish-subscribe brokers allow queries to be registered with topics. Published events that match the queries are routed to the subscriber [21]. Brokers like MQTT are commonly used to route streams from sensors to application consumers through a well-defined topic for each sensor [22]. But if every (large) micro-batch is sent as an event to a Cloud broker, it defeats the goal of moving compute to the data on the edge. If no queries match the micro-batch, the data movement has been unnecessary. If just the metadata is sent, we still spend the round-trip latency for triggering. Our application triggering design approaches a *federated* publish-subscribe model, with the subscriber initiating the execution of a DAG. *Complex Event Processing (CEP)* has been used for analytics over sensor events [4]. Here, we adopt it within the QE engine to match the micro-batch metadata and trigger DAGs.

Function as a Service (FaaS) offer a "serverless" application model on the Cloud to execute simple stateless functions over input events [13]. Users also define a resource requirement. Invocations to these functions can then be seamlessly scaled by the provider on multiple VMs. FaaS are also offered on the edge by Azure IoT Edge [23]. Our stateless tasks resemble FaaS but offer dataflow composition, and leverage data locality and lower pricing when deciding where to run the function.

Research literature has examined *programming models* for mobile edge, and on-demand fog and cloud resources along with a scheduling strategy [24]. Here, the applications are strictly wired as a tree, typically rooted in the cloud, and the tasks are deployed prescriptively at specific edge or fog hierarchy levels. We allow placement of tasks on any resource, based on deadline and cost optimization. They perform elastic resource acquisition when a fog is overloaded, and spatially repartition the process state, but do not consider the cost and deadline. CoFEE imposes a makespan deadline constraint with edge and fog having bounded capacities.

There is a large body of work on scheduling applications on edge, fog and cloud [25]. Some consider *scheduling strategies* on fog and cloud [26] for a Bag of Tasks (BoT), where cloud resources are billed while the fog is free. They use fog to execute as many tasks as possible with two minimization goals – the delay to propagate data to the cloud, and the cost. They extended this for scheduling DAGs with deadlines on fog and cloud using ILP [19], which we empirically compare against in Sec. V. Here, if a fog partition is not able to accept a task, it off-loads it to a neighboring fog. Reliability is a non-goal. We instead consider a global view of all available resources and solicit bids to select the cheapest one. Our fog and edge resources are billed, albeit cheaper than the cloud.

Some papers use *hard deadlines* – more appropriate for mission-critical execution on streaming data – and maximize resource usage on free fogs [27]. We use micro-batch and soft deadlines for tunable latency and throughput. Our edge and fog resources have usage costs, and the edge is unreliable. *Energy constraints* are also considered in scheduling. Deng, et al. [28] investigate the power-delay trade-off as a metric for workload allocation in fog and cloud. They schedule transactional tasks on few fog and cloud resources to minimize their power usage while meeting the delay constraint. Others [29] map IoT tasks on Fog and Cloud resources using network bandwidth, latency and energy as constraints. In this paper, we consider energy to be subsumed by the pricing of the edge and fog.

There exists literature on distributed dataflow runtime and *middleware* for edge and/or fog and/or cloud computing [12], [30]. However, they do not focus on application triggering and advanced scheduling. Some also consider migration of tasks and VMs for reliability and locality [31]. We instead focus on light-weight task-level re-execution.

Several dimensions such as flexible dataflow triggering, hybrid resource reliability, and decentralized planning are missing from existing scheduling literature. We address these concerns here. We propose a novel declarative data-triggered approach for instantiating DAGs. These are scheduled on unreliable edge, and reliable fog and cloud resources to minimize the execution cost, and we use advanced slot reservations to meet the deadline and operate at scale.

VII. Conclusions

In this paper, we have presented a novel declarative model for matching micro-batches generated by evolving stream sources with dataflows that are interested in consuming them, allowing developers to intuitively trigger their applications for sources based on their description ("what") rather than their endpoint ("where"). Our micro-batch model balances throughput against latency to scale to 1000s of streams. We have proposed a scheduling strategy to minimize execution costs on unreliable edges, and reliable fog and cloud resources using a federated bid–inquiry price discovery mechanism that scales with the resource count. It is also able to ensure reliable dataflow completion before its deadline using advanced slot reservations on the fogs. Our CoFEE platform implements these and is validated through comparative experiments on 30 DAGs, $10k$ task runs and > 100 edge, fog and cloud resources.

As future work, we plan to examine models that avoid complete unrolling of the DAG for efficiency. We also plan better recovery strategies based on keeping track of the execution lineage of tasks to avoid having to cache micro-batches, and to support triggering over micro-batch generated in the past. More detailed experimental setups will be useful as well.

Acknowledgement

This work was supported by grants from the Department of Science & Technology, India under the Internet of Things (IoT) Research of Interdisciplinary Cyber Physical Systems (ICPS) Programme. We thank the members of the DREAM:Lab at IISc, including Prashanthi S.K. and Deepsubhra Guha Roy, for their feedback on the CoFEE platform and the paper.

References

[1] Y. Simmhan, P. Ravindra, S. Chaturvedi, M. Hegde, and R. Ballamajalu, "Towards a data-driven iot software architecture for smart city utilities," *Software: Practice and Experience*, vol. 48, no. 7, pp. 1390–1416, 2018.

[2] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, *Fog computing: A platform for internet of things and analytics*. Springer, 2014, pp. 169–186.

[3] S. Aman, M. Frincu, C. Chelmis, M. Noor, Y. Simmhan, and V. K. Prasanna, "Prediction models for dynamic demand response: Requirements, challenges, and insights," in *2015 IEEE International Conference on Smart Grid Communications (SmartGridComm)*. IEEE, 2015, pp. 338–343.

[4] P. Kolios, C. Panayiotou, G. Ellinas, and M. Polycarpou, "Data-driven event triggering for iot applications," *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 1146–1158, 2016.

[5] H. Nasiri, S. Nasehi, and M. Goudarzi, "Evaluation of distributed stream processing frameworks for iot applications in smart cities," *Journal of Big Data*, vol. 6, no. 1, pp. 1–24, 2019.

[6] B. Amrutur, V. Rajaraman, S. Acharya, R. Ramesh, A. Joglekar, A. Sharma, Y. Simmhan, A. Lele, A. Mahesh, and S. Sankaran, "An open smart city iot test bed: street light poles as smart city spines," in *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation*, 2017, pp. 323–324.

[7] J. He, J. Wei, K. Chen, Z. Tang, Y. Zhou, and Y. Zhang, "Multitier fog computing with large-scale iot data analytics for smart cities," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 677–686, 2017.

[8] R. Buyya, S. N. Srirama, G. Casale, R. Calheiros, Y. Simmhan, B. Varghese, E. Gelenbe, B. Javadi, L. M. Vaquero, M. A. Netto *et al.*, "A manifesto for future generation cloud computing: Research directions for the next decade," *ACM computing surveys (CSUR)*, vol. 51, no. 5, pp. 1–38, 2018.

[9] L. F. Bittencourt, J. Diaz-Montes, R. Buyya, O. F. Rana, and M. Parashar, "Mobility-aware application scheduling in fog computing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 26–35, 2017.

[10] P. Varshney and Y. Simmhan, "Demystifying fog computing: Characterizing architectures, applications and abstractions," in *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, 2017, pp. 115–124.

[11] M. D. de Assuncao, A. da Silva Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," *Journal of Network and Computer Applications*, vol. 103, pp. 1–17, 2018.

[12] P. Ravindra, A. Khochare, S. P. Reddy, S. Sharma, P. Varshney, and Y. Simmhan, "Echo: An adaptive orchestration platform for hybrid dataflows across cloud and edge," in *International Conference on Service-Oriented Computing*. Springer, 2017, pp. 395–410.

[13] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski, "Status of serverless computing and function-as-a-service (faas) in industry and research," Tech. Rep., 2017.

[14] S. Abrishami, M. Naghibzadeh, and D. H. Epema, "Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds," *Future Gener. Comput. Syst.*, vol. 29, no. 1, Jan. 2013.

[15] L. Ramakrishnan, C. Koelbel, Y.-S. Kee, R. Wolski, D. Nurmi, D. Gannon, G. Obertelli, A. YarKhan, A. Mandal, T. M. Huang *et al.*, "Vgrads: enabling e-science workflows on grids and clouds with fault tolerance," in *Proceedings of the conference on high performance computing networking, storage and analysis*, 2009, pp. 1–12.

[16] S. Baheti, S. Badiger, and Y. Simmhan, "Violet: An emulation environment for validating iot deployments at large scales," *ACM Transactions on Cyber-Physical Systems*, vol. 5, no. 3, pp. 1–39, 2021.

[17] A. Shukla, S. Chaturvedi, and Y. Simmhan, "Riotbench: A real-time iot benchmark for distributed stream processing platforms," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 21, 2017.

[18] S. Abrishami, M. Naghibzadeh, and D. H. Epema, "Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds," *Future generation computer systems*, vol. 29, no. 1, pp. 158–169, 2013.

[19] O. Skarlat, M. Nardelli, S. Schulte, and S. Dustdar, "Towards qos-aware fog service placement," in *IEEE International Conference on Fog and Edge Computing (ICFEC)*, 2017, pp. 89–96.

[20] A. V. Dastjerdi, H. Gupta, R. N. Calheiros, S. K. Ghosh, and R. Buyya, "Chapter 4: Fog computing: principles, architectures, and applications," in *Internet of Things*, R. Buyya and A. Vahid Dastjerdi, Eds. Morgan Kaufmann, 2016, pp. 61–75.

[21] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM computing surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[22] A. Banks and R. Gupta, "Mqtt version 3.1.1," *OASIS standard*, vol. 29, 2014.

[23] Microsoft, "Deploy azure functions as iot edge modules," https://docs.microsoft.com/en-us/azure/iot-edge/tutorial-deploy-function, 2021.

[24] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe, "Mobile fog: A programming model for large-scale applications on the internet of things," in *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing*, 2013, pp. 15–20.

[25] P. Varshney and Y. Simmhan, "Characterizing application scheduling on edge, fog, and cloud computing resources," *Software: Practice and Experience*, vol. 50, no. 5, pp. 558–595, 2020.

[26] O. Skarlat, S. Schulte, M. Borkowski, and P. Leitner, "Resource provisioning for iot services in the fog," in *IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, 2016, pp. 32–39.

[27] L. Yin, J. Luo, and H. Luo, "Tasks scheduling and resource allocation in fog computing based on containers for smart manufacturing," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 10, 2018.

[28] R. Deng, R. Lu, C. Lai, T. H. Luan, and H. Liang, "Optimal workload allocation in fog-cloud computing toward balanced delay and power consumption," *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 1171–1181, 2016.

[29] A. Brogi and S. Forti, "Qos-aware deployment of iot applications through the fog," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1185–1192, 2017.

[30] V. Issarny, G. Bouloukakis, N. Georgantas, and B. Billet, "Revisiting service-oriented architecture for the iot: a middleware perspective," in *International Conference on Service-Oriented Computing*. Springer, 2016, pp. 3–17.

[31] L. F. Bittencourt, M. M. Lopes, I. Petri, and O. F. Rana, "Towards virtual machine migration in fog computing," in *IEEE International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, 2015.