Vela: A 3-Phase Distributed Scheduler for the Edge-Cloud Continuum

Thomas Pusztai
Distributed Systems Group, TU Wien
Vienna, Austria
t.pusztai@dsg.tuwien.ac.at

Stefan Nastic
Distributed Systems Group, TU Wien
Vienna, Austria
s.nastic@dsg.tuwien.ac.at

Philipp Raith
Distributed Systems Group, TU Wien
Vienna, Austria
p.raith@dsg.tuwien.ac.at

Schahram Dustdar
Distributed Systems Group, TU Wien
Vienna, Austria
dustdar@dsg.tuwien.ac.at

Deepak Vij
Futurewei Technologies, Inc.
Santa Clara, CA, USA
dvij_sj12@yahoo.com

Ying Xiong
Futurewei Technologies, Inc.
Santa Clara, CA, USA
yingx@live.com

***Abstract*—The amalgamation of multiple Edge and Cloud clusters into an Edge-Cloud continuum requires efficient scheduling techniques to cope with high numbers of infrastructure nodes and computing jobs. Since monolithic schedulers typically do not scale well beyond a certain cluster size, distributed scheduling approaches are usually employed to address such scalability issues. Distributed schedulers are often designed for Cloud environments and lack support for the Edge. Conversely, many Edge schedulers focus on single clusters and provide limited support to deal with the scale of the Edge-Cloud continuum. In this paper, we present the Vela Distributed Scheduler, a globally distributed scheduler, which is specifically tailored for the Edge-Cloud continuum. The main contributions of our work include: i) A novel, globally distributed and orchestrator-independent scheduler with a 3-phase scheduling workflow; ii) A two-level, informed sampling mechanism, which reduces latency for globally distributed sampling and leverages job requirements to produce high quality node samples; And iii) a MultiBind mechanism that significantly reduces job evictions and rescheduling due to scheduling conflicts. We implement Vela on top of Kubernetes and evaluate it in a realistic large-scale setup using multiple interconnected, globally distributed, and production-ready MicroK8s clusters with up to 20,000 total simulated nodes. Our results show that Vela's performance scales linearly with infrastructure size and that it reduces scheduling conflicts by a factor of 10.**

***Index Terms*—distributed scheduling, edge computing, edge-cloud continuum**

I. Introduction

When multiple Edge and Cloud clusters meld together they form what is called the *Edge-Cloud continuum*. Executing the microservices of an application on the right nodes allows the application to take advantage of the best of both worlds, i.e., the low latency, proximity to the users, and attached Internet of Things (IoT) devices of the Edge and the powerful compute resources of the Cloud. Placing a workload in the Edge-Cloud continuum, which can often span tens to hundreds of thousands of nodes is challenging for a monolithic scheduler and, thus, often calls for a distributed scheduling approach.

This work is supported by Futurewei's Cloud Lab. as part of the overall open source initiative.
This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871403.

TABLE I: Scheduler Architectures Comparison

Type	State per Instance	State Synchronization	Conflicts Handling	Limitations
Monolithic e.g., [2]–[4]	Entire cluster	Not needed, because single instance only	Avoided by monolithic state	Limited infrastructure size
Two-level e.g., [5]–[7]	Statically or dynamically partitioned by 1st level	Not needed, because state is partitioned	Avoided by partitioning	Local optima and potentially limited infrastructure size if 1st level is monolithic
Shared State e.g., [8]–[11]	Entire cluster	E.g., read-only master state with frequent sync or partitioned sync	E.g., transactions or optimistic concurrency	Limited infrastructure size, since each scheduler maintains entire state
Distributed e.g., [12]	Sampled set of nodes	Sampling	Optimistic concurrency	Local optima
Hybrid e.g., [13], [14]	Depends on combination	Depends on combination	Depends on combination	One part is usually monolithic

There are multiple architectures for designing distributed schedulers, namely two-level, shared state, distributed, and hybrid [1]. We examine their differences from the monolithic architecture and from each other in four major aspects: i) how much of the scheduling-related infrastructure state a single scheduler instance sees, ii) how this state is synchronized, iii) how scheduling conflicts (i.e., two schedulers assign the same resources) are handled, and iv) architecture limitations.

Table I summarizes the scheduler architectures. Monolithic schedulers handle the entire infrastructure state within a single instance, which prevents conflicts, but limits scalability w.r.t. the infrastructure size. Two-level schedulers rely on a hierarchy, where the first level is responsible for the entire infrastructure state and statically or dynamically partitions it among an arbitrary number of second level schedulers. This prevents conflicts and improves scalability, but it may lead to local optima and, if the first level is monolithic, scalability may still be limited. Shared state schedulers operate with multiple schedulers that share access to the entire infrastructure state. Conflicts may occur, especially if the local state is outdated and the scale of the infrastructure is limited, because each scheduler has a copy of the entire state. Distributed schedulers rely on multiple schedulers that have a limited view of the infrastructure state, often obtained by selecting a portion of

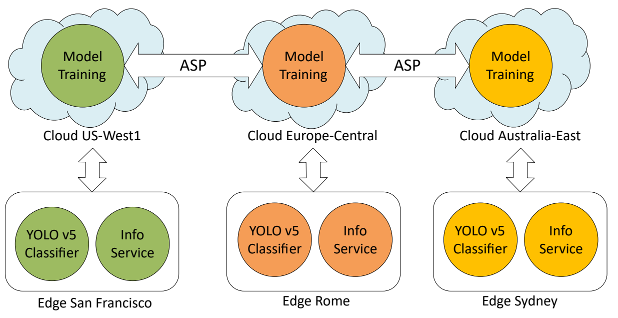

Fig. 1: Globally Distributed Machine Learning.

nodes (sampling), making this architecture highly scalable. The sampling algorithm influences the scheduling decisions' quality and the conflict probability. Hybrid schedulers combine two of the other approaches, usually a monolithic scheduler for one type of jobs and a distributed scheduler for all others.

Edge schedulers typically optimize placement for a set of Edge-specific constraints, such as network latency or geo-location, but they often lack the scalability needed for an online scheduler in the Edge-Cloud continuum, because they rely on computationally intensive algorithms, such as genetic algorithms, or because they focus on a single cluster and, hence, lack a distributed architecture. Those that focus on scalability, e.g., [15]–[20], are often limited to scheduling batch jobs, not microservices, and none of them consider multiple globally distributed clusters. Their evaluations are limited to small clusters with less than 1,000 nodes, which does not allow drawing conclusions on global scalability.

Typically, clusters are managed by an orchestrator, e.g., Kubernetes[1] or Nomad[2], which is responsible for deploying and launching jobs and providing management services. The scheduler is often part of the orchestrator, but it may also be an external component that only interfaces with it to make job placement decisions.

The need for globally distributed scheduling in the Edge-Cloud continuum is exacerbated by novel large-scale applications that often require global deployments, such as general public augmented reality (AR)/Metaverse [21] or geo-distributed machine learning (ML). Such scenarios may also encompass scheduling on heterogeneous devices, like a combination of high-end servers and single-board computers, with the latter being required, e.g., for privacy preserving preprocessing [22].

A use case of globally distributed ML, based on the Gaia ML system [23], is shown in Fig. 1. An AR application for tourists classifies images to display sightseeing information to its users. Classification jobs use the YOLOv5 CNN model to match user videos to sights in a city. An info service provides information to display to the users. Both jobs need to run as services in Edge clusters close to the users, because latency is critical in AR applications [24]. Training jobs to improve the model are run daily in a federated manner in the Cloud, relying mostly on local images from the closest Edge clusters and synchronizing the model globally using the Approximate Synchronous Parallel (ASP) model [23]. With global communications, latency plays a role and demands a reduction of packet round trips between scheduler and target nodes.

[1]https://kubernetes.io

[2]https://www.nomadproject.io

We formulate the following research challenges:

RC-1 *How can a scheduler for the Edge-Cloud continuum handle globally distributed Cloud and Edge clusters and scale reliably with the infrastructure?* As previously mentioned, monolithic schedulers can only handle a limited number of nodes, e.g., Kubernetes officially supports up to 5,000 nodes [25]. But also distributed schedulers may have limitations related to state synchronization, handling of scheduling conflicts, and scalability. However, scalability is an important feature of a scheduler [26], especially when dealing with very large infrastructures that span multiple, globally distributed clusters [27].

RC-2 *How can high-quality samples be collected from globally distributed clusters, while maintaining low sampling and scheduling latency?* Sampling-based schedulers are designed to handle large clusters. They commonly either retrieve samples from a local or shared cluster state, such as Tarcil [10], or contact nodes directly, like Sparrow [12]. The former approach does not work for globally distributed clusters, because maintaining a detailed state of globally distributed nodes is not feasible, nor does the latter, because contacting many globally distributed nodes directly would significantly increase scheduling latency, given global packet round trip times, e.g., 165 ms as per the Verizon SLA for a Europe-USA packet round trip [28] (sum of round trips within Europe, USA, and transatlantic). Additionally, as clusters get more loaded, it has been reported that larger samples are needed to find suitable nodes [10], because the samples are more likely to contain nodes that are full. Such wasted samples increase load on the scheduler. Thus, a sampling mechanism is needed that i) delegates work to the clusters to minimize the latency incurred by network communication and ii) leverages job requirements to return only suitable nodes to avoid an increase in sample size.

RC-3 *How can a distributed scheduler increase job throughput by reducing the number of scheduling conflicts?* The assignment of the same set of resources to two different jobs by two scheduler instances and the resulting conflict is an issue recognized by many distributed schedulers [8]–[11], [13]. Rescheduling the conflicting jobs takes a significant amount of time and reduces the scheduler's job throughput, because the jobs need to traverse the entire scheduling lifecycle again. Reducing conflicts requires careful consideration of the scheduler's inner workings. While a job is being committed to a node, the sampling algorithm may rely on an outdated state and suggest a node, although it will be full after the commit has completed. Accounting for this issue and adding mitigation measures when conflicts do arise can significantly reduce rescheduling and, thus, increase the overall throughput of the scheduler.

In this paper we present the open-source Vela Distributed Scheduler[3], which is part of Polaris SLO Cloud[4], a SIG of the Linux Foundation Centaurus project[5], a novel open-source platform for building unified and highly scalable public or private distributed Cloud and Edge systems. Vela continues our line of research on scheduling in the Edge-Cloud continuum continuum [29] [30]. Our main contributions include:

1) *Vela Scheduler, a novel, globally distributed, orchestrator-independent scheduler with a 3-phase scheduling workflow* to enable optimized scheduling of microservices at global scale within the Edge-Cloud continuum. The workflow is distributed across multiple components to ensure scalability and is comprised of a sampling phase that retrieves node samples from globally distributed clusters, a decision phase that picks the best suitable node, and a commit phase that enforces the scheduling decision in a conflict-aware manner.
2) *2-Smart Sampling, a two-level, informed sampling mechanism that delegates sampling to globally distributed clusters and leverages job requirements to produce samples consisting of nodes that are likely to be suitable*. This reduces scheduling latency and sample wastage. Vela's design for globally distributed clusters delegates sampling to agents in the clusters, which frees the scheduler from communicating with the nodes directly. This delegation greatly reduces network traffic and latency for the scheduler. By leveraging job requirements, the likelihood that the sample contains suitable nodes is greatly increased, while avoiding large sample sizes, which would augment the scheduler's load. To the best of our knowledge, there is no other scheduler that is designed to perform sampling on a global scale or is evaluated in a globally distributed sampling scenario.
3) *MultiBind, a scheduling decision commit phase that automatically retries committing the job to another node if a scheduling conflict occurs*, without rerunning the entire scheduling process. This significantly reduces the number of jobs that need to be rescheduled due to conflicts and, thus, increases the overall throughput of the scheduler.

This paper is structured as follows: Section II examines related work, Section III provides an overview of the architecture of the Vela Distributed Scheduler, and Section IV describes the mechanisms that realize our contributions. Section V evaluates our scheduler on multiple interconnected Kubernetes clusters that represent an Edge-Cloud continuum and Section VI provides and outlook on future work and concludes the paper.

II. Related Work

The default schedulers of Kubernetes [3] and Docker-Swarm [4] suffer from the typical issues of monolithic schedulers that we have previously mentioned. There are many works that focus on Edge-related capabilities for monolithic schedulers, e.g., Rossi et al. [31] propose a latency-aware Kubernetes scheduler for geo-distributed environments and Santos et al. [32] add latency- and bandwidth-awareness to their Kubernetes scheduler extension. Hailiang et al. [33] use a genetic algorithm that aims to reduce the response time for microservice-based Edge applications, but the algorithm runs offline, which inherently prevents it from being scalable. In general, none of these works consider a distributed approach to ensure scalability for the Edge-Cloud continuum, hence they cannot be applied in a globally distributed context like Vela.

Mesos [5] and YARN [6] are two-level schedulers that are frequently used in production [34], [35]. Their top-level is monolithic and the second-level relies on partitioning. For Mesos all scheduling decisions have to pass through the top-level scheduler, which may result in a bottleneck, and YARN's top-level needs to capture the entire cluster state and assign fine-grained resources to the second level, which may be an issue if the entire cluster state gets too big to fit into memory. The Fair Scheduler [36] in YARN allows achieving a fair resource distribution among second-level schedulers and Capacity Scheduler [37] ensures that each tenant of a multi-tenant system gets a minimum share of resources, but both approaches are designed for the Cloud, not the Edge. Epsilon [38] and OneEdge [7] are also two-level schedulers, whose first levels are monolithic. Epsilon's second level utilizes the shared state concept and supports autoscaling of the second-level schedulers. OneEdge uses sharding for the second level schedulers and it supports enforcing and End-to-End (E2E) latency Service Level Objective (SLO). The major issue with these approaches is the monolithic first level, which can hinder scalability – Vela Scheduler aims to avoid this using its fully distributed, sampling-based approach, which does not require scheduler instances to maintain any cluster state beyond the node samples that are retrieved independently for each job. The downside of sampling is that the ideal solution may not be part of the sample, an issue that Vela tries to mitigate using its 2-Smart Sampling mechanism (additionally, we plan further improvements on this using AI-based sampling in future work). Hydra [27] builds on top of YARN and greatly improves scalability by federating multiple two-level clusters across multiple data centers, however it is designed for the Cloud and does not focus on Edge clusters.

Apollo [9], Omega [8], Tarcil [10], and ParSync [11] are shared state schedulers. Apollo's shared state is centralized and treated as read-only for the schedulers; the state can only be updated by status updates received from the cluster nodes. Omega supports different types of transactions to reduce scheduling conflicts. ParSync partitions the state internally and the scheduler instances get updates on different partitions on every synchronization iteration. The schedulers prefer to pick nodes from recently updated partitions to avoid relying on stale state data and, thus, reduce the chance for scheduling conflicts. Tarcil improves speed by sampling nodes from a shared state, but if the cluster is heavily loaded the sample size becomes very large, e.g., 82% of the nodes in one of their examples.

[3]https://polaris-slo-cloud.github.io/vela-scheduler
[4]https://polaris-slo-cloud.github.io
[5]https://www.centaurusinfra.io

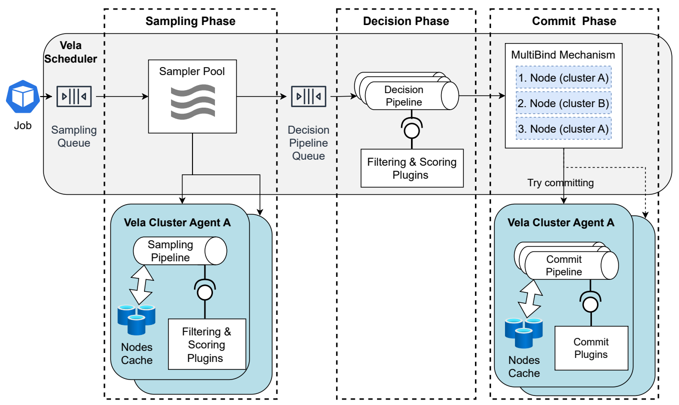

Fig. 2: Scheduling Workflow and Job Lifecycle.

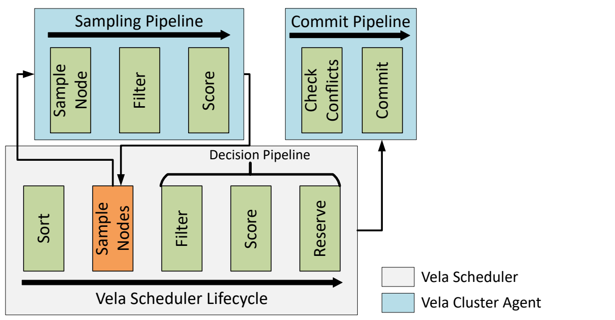

Fig. 3: 3-Phase Scheduling Workflow with Sampling, Decision, and Commit Pipelines.

Arktos [39] improves on the scalability of the Kubernetes scheduler by turning it into a shared state scheduler. All shared state schedulers suffer from the issue that the entire cluster state may become too large to be handled by a single scheduler instance and from the occurrence of scheduling conflicts. Our approach avoids the former issue by being fully distributed and drastically reduces conflicts using the MultiBind mechanism.

Sparrow [12] is a distributed scheduler designed for batch jobs that relies on sampling to collect nodes. The nodes are contacted directly, which is not feasible with globally distributed nodes. A late-binding mechanism is used to ensure that a job starts as quickly as possible: a job is assigned to the queues of all eligible nodes and the first node that dequeues the job gets to execute it. Sparrow cannot have scheduling conflicts, because jobs can always be queued on a node, an assumption that is only valid for batch processing systems. While Sparrow supports constraints for its sampling phase, they are evaluated in a centralized fashion. Vela Scheduler avoids contacting nodes directly to allow for global distribution and it specifically addresses scheduling conflicts, because it is not restricted to batch jobs.

Mercury [13] and Hawk [14] are hybrid schedulers that combine a monolithic scheduler for one type of jobs with a distributed scheduler for other jobs. Mercury divides the two scheduling approaches between “guaranteed” and “queueable” jobs, while Hawk divides them between “long“ and “short“ jobs respectively. Mercury solves conflicts by terminating queueable jobs in favor of guaranteed jobs, while Hawk avoids conflicts by queuing. Naturally, the monolithic part can become a bottleneck and many systems have a single job type, so these approaches are not always applicable. Vela Scheduler does not have this bottleneck and, while being primarily designed for microservices, it can support any job type through appropriate plugins.

III. Vela 3-Phase Scheduling Workflow

The Vela Distributed Scheduler is designed to manage multiple, globally distributed Edge and Cloud clusters. It consists of two components, the *Scheduler* and the *Cluster Agent*. The scheduler can be deployed with an arbitrary number of instances, which are independent of the infrastructure, i.e., clusters, they need to manage. Due to its orchestrator-independent design, clusters may be operated by different orchestrators, e.g., one cluster might use Kubernetes, while another cluster might use Nomad. The exact definition of a cluster node depends on the respective orchestrator – typically, a node will be either a VM, a bare-metal server, or a single-board computer. Every node can host multiple jobs, as long as it has sufficient resources to accommodate them. To become enabled for the Vela Scheduler, each cluster only needs to deploy the Cluster Agent, typically as a controller.

The 3-phase scheduling workflow (see RC-1) is the logical centerpiece of Vela. The workflow and the lifecycle of a job within it are shown in Fig. 2. Each of the three phases, i.e., **sampling**, **decision**, and **commit**, contains a pipeline; these pipelines are shown in Fig. 3. Each pipeline consists of multiple stages. The business logic within each stage is realized through plugins, which facilitates the implementation of diverse scheduling policies.

The 3-phase scheduling workflow starts when a user or another system component, such as an autoscaler, submits a job to an arbitrary instance of the Vela Scheduler. The scheduler instance sorts incoming jobs, e.g., based on priority, in its Sort stage and then adds them to its *Sampling Queue*.

Once the scheduler dequeues the job, it enters the *sampling phase* with the 2-Smart Sampling mechanism – the sampler pool can process multiple jobs in parallel in this phase. 2-Smart Sampling consists of two steps, the first one is executed by the Sample Nodes plugin. It selects a random subset of all configured clusters to be used for sampling and requests a sample from their respective Cluster Agents, passing the job's requirements along to ensure that only nodes that fulfill these requirements are returned.

Each cluster's Cluster Agent, then, executes the second step of 2-Smart Sampling. The agent maintains a frequently updated cache of its cluster's nodes – the exact implementation depends on the underlying orchestrator, e.g., Kubernetes provides a watch mechanism that notifies the agent on nodes list changes. The agent selects a set of nodes from this cache and executes the *sampling pipeline*, which employs a multi-criteria decision making (MCDM) approach, consisting of the Filter and the Score stages. *Filter* plugins remove nodes that are not suitable for hosting a job and *Score* plugins assign scores from 0 to 100 to the nodes that have survived filtering, based on how suitable they are. A higher score indicates better suitability for the job, e.g., empty nodes may score higher than partially loaded ones. The sampled nodes are then returned to the Vela Scheduler, which places them, together with the job, in the *decision pipeline queue*. The presence of this queue ensures that sampling, which may consume some time, can be executed on different threads from the decision pipeline. This allows avoiding situations where all threads might be blocked waiting for samples, while the CPU remains idle, even though it could be used for the decision pipeline. If desired, a timeout can be configured for each sampling request to a Cluster Agent – this can be used if a use case has stringent requirements on scheduling latency.

When the job exits this queue, it enters the decision phase. The *decision pipeline* further evaluates the sampled nodes for their suitability using another set of Filter and Score plugins that allow enforcement of global policies. Multiple decision pipelines, each responsible for a single job, are executed on concurrent threads. Since the Cluster Agent's sampling pipeline also includes scoring, each node already has a list of scores produced by the sampling Score plugins. The scores computed by the scheduler's Score plugins are added to this list. After all eligible nodes are scored, the decision pipeline accumulates the scores and picks the top m nodes with the highest score, with m being determined by the configuration of the MultiBind mechanism. The Reserve stage can be used by plugins to update internal data structures. At the end of the decision pipeline, the scheduler advances the top m nodes to the commit phase. The decision pipeline requires no synchronization with other pipeline- or scheduler instances, because the only point of synchronization is located in the subsequent commit phase and is handled by the Cluster Agent.

In the *commit phase* the MultiBind mechanism instructs the Cluster Agent, responsible for the cluster of the first of the m selected nodes, to commit the scheduling decision to the node. Since scheduling decisions can be made simultaneously by multiple scheduler instances, scheduling conflicts may occur, i.e., two jobs may be assigned to the same node by different scheduler instances, but the node only has enough remaining capacity to host one of them. To handle such conflicts we rely on an optimistic concurrency approach within the Cluster Agent, which checks for each job, if the resources are still available. In case of a conflict, the first job to arrive is committed to the node, the second job is rejected. To this end, the *commit pipeline* first reserves resources in the agent's cache to make them unavailable to the sampling pipeline, then fetches the current information about the node, checks if the constraints are still fulfilled, and, finally commits the decision by binding the job to the node. If the commit pipeline fails, the MultiBind mechanism takes the next best node from the list of m most suitable nodes and tries committing the job to that one. Only if all m nodes fail, will the job be considered as having a scheduling conflict, which requires rescheduling, i.e., running the entire scheduling workflow again. Our experiments show that the MultiBind mechanism reduces the number of conflicts by a factor of up to 10.

Currently, Vela Scheduler is aimed at stateless microservices. However, its plugin-based design allows adding plugins to support stateful microservices or batch jobs in the future.

Vela is fault-tolerant by design. The failure of a Cluster Agent means that its cluster is not available for scheduling, but does not affect the availability of other clusters. Since no coordination is needed among scheduler instances, the failure of one instance only requires users to submit new scheduling requests to another instance, but has no effect on the operational status of the overall system.

IV. VELA'S MAIN SCHEDULING MECHANISMS

In this section we present the two most important scheduling mechanisms, i.e., 2-Smart Sampling and MultiBind in detail.

A. 2-Smart Sampling

To reduce latency and avoid large sample sizes, even in loaded clusters (see RC-2), Vela Scheduler introduces 2-Smart Sampling, a two-step informed sampling approach, where the scheduler delegates sampling to the Cluster Agents in the selected clusters. This delegation frees the scheduler from communicating with globally distributed nodes directly, which would incur high latency, and allows sampling to take full advantage of the local information that is available within the cluster. Specifically, 2-Smart Sampling executes in two steps:

1) The scheduler picks a random subset of all configured clusters to be contacted for samples. Using only a subset ensures scalability and reduces scheduling conflicts. Then, the scheduler contacts the Cluster Agent of each selected cluster for a node sample, passing along all the job's requirements.
2) Each contacted Cluster Agent runs the sampling pipeline to pick a set of nodes and check them for eligibility for hosting the job. The nodes that are deemed eligible are scored and then returned to the scheduler.

The percentage of clusters to be sampled (C_p) and the percentage of nodes to sample per cluster (N_p) can be configured.

The sampling pipeline in the second step of 2-Smart Sampling consists of three stages (see Fig. 3): Sampling Strategy, Filter, and Score. The scheduling policy of each stage is implemented by one or more plugins. Currently we provide two Sampling Strategy plugins (a sampling request specifies which one to use), one for random sampling and one for Round-Robin sampling, and three Filter plugins: `ReourcesFit` ensures that a node fulfills the job's resource requirements, `GeoLocation` allows a job to specify that it needs to run in a specific location, and `BatteryLevel` allows restricting a job to running on a node that has a minimum battery level (if the node has a battery) – the former two plugins also tie into the Score stage.

For each job 2-Smart Sampling operates as shown in Algorithm 1:

Step 1. Lines 3–9 execute the first step of 2-Smart Sampling, i.e., pick a random subset of all clusters and request a sample from their Cluster Agents. The returned samples are added to the decision pipeline queue, together with the job.

Algorithm 1 Sampling Phase

```
 1: Input: $j$: The job for which to sample nodes;
    $C_p$: Percentage of clusters to sample;
    $N_p$: The number of nodes to sample per cluster;
    $strat$: The sampling strategy to use;
 2: Output: $S_e$: The set of sampled nodes that are eligible for hosting $j$ and
    their scores;

              ▷ The 1st step of 2-Smart Sampling runs within the scheduler
 3: $S_e \leftarrow \{\}$
 4: $C \leftarrow$ PickRandomClustersToSample($C_p$)
 5: for all $c \in C$ do
 6:     $S_{e,c} \leftarrow$ c.RunClusterSamplingPipeline($j, N_p, strat$)
 7:     $S_e \leftarrow S_e \cup S_{e,c}$
 8: end for
 9: AddToDecisionPipelineQueue($j, S_e$)

    ▷ The 2nd step of 2-Smart Sampling, i.e., the sampling pipeline, runs
    within a Cluster Agent
10: function RunClusterSamplingPipeline($j, N_p, strat$)
11:     $sampleSize \leftarrow$ ComputeSampleSize($N_p$)
12:     $S_{e,c} \leftarrow \{\}$                 ▷ The sampled nodes from this cluster

13:     while $|S_{e,c}| < sampleSize$ AND NOT timeout occurred do
14:         $S_i \leftarrow$ SampleNodesWithStrategy($strat$, $sampleSize$)
15:         for all $n \in S_i$ do
16:             if RunAllFilterPlugins($n$) = $true$ then
                        ▷ If the node survives all filter plugins, it is eligible.
17:                 $S_{e,c} \leftarrow S_{e,c} \cup \{n\}$
18:             end if
19:         end for
20:     end while

21:     for all $n \in S_{e,c}$ do
22:         RunAllScorePlugins($n$)     ▷ Run all score plugins and add
    the scores to the node $n$
23:     end for

24:     return $S_{e,c}$
25: end function
```

Step 2. Lines 10–25 execute the second step of 2-Smart Sampling in each involved Cluster Agent. Lines 13–20 constitute the sampling and filtering loop, which proceeds until enough eligible nodes have been found or a timeout is reached. Line 14 gets a set of nodes from the Sampling Strategy plugin, e.g., random sampling (default) or Round-Robin. Lines 15–19 run all Filter plugins on each sampled node to determine if it fulfills the job' requirements. Lines 21–23 execute all Score plugins on each eligible node. Subsequently, the complete cluster sample is returned to the scheduler.

This approach ensures that each cluster's sample only contains nodes that meet the job's requirements, which allows for a smaller sample size. The sampling pipeline plugins need to ensure that the job's resource requirements are met by a node, but they may also implement complex policies that further improve the quality of the sample. The Cluster Agent may also implement cluster-specific scheduling policies.

B. MultiBind Commit Phase

Vela Scheduler relies on an optimistic concurrency approach to deal with multiple decision pipeline or scheduler instances running in parallel. No cluster node resources are locked during the sampling phase, because most of them will not be used – in the end the job is assigned to a single node. This improves scalability, but entails that when committing a scheduling decision, another decision pipeline or scheduler instance may have already claimed some of the required resources on the node, resulting in a scheduling conflict for the current job. This is a common issue in distributed scheduling, which is normally handled by rescheduling the job (see RC-3) [8]–[11]. In Vela Scheduler we mitigate this issue by the randomness in both steps of 2-Smart Sampling. Nevertheless, scheduling conflicts can occur. Note that the number of jobs per node is not limited, i.e., if the selected node has enough resources for both jobs, both are committed and executed – a conflict only occurs, if the node does not have sufficient resources for hosting both jobs.

To further reduce the number of scheduling conflicts that require rescheduling, Vela Scheduler relies on its MultiBind commit phase: instead of trying to commit the job only to the highest scored node and rescheduling it, if a conflict occurs, we use a list of the m highest scored nodes and try committing to the next node. Only if committing to all m nodes fails, the job is considered to have a scheduling conflict and needs to be rescheduled. Our tests in Section V show that a setting of $m = 3$ reduces the scheduling conflicts by factor of 10 compared to not using MultiBind. When trying to commit a job to a node, the MultiBind mechanism contacts the Cluster Agent of the node's cluster to execute the commit pipeline. This pipeline, which supports running multiple instances in parallel, contains two stages, whose logic is implemented using plugins: the *Check Conflicts* stage and the *Commit* stage. The entire process executed by the MultiBind commit phase is shown in Algorithm 2:

Algorithm 2 Commit Phase

```
 1: Input: $j$: The job to commit;
    $N = (n_1, ..., n_m)$: The $m$ highest scored nodes as commit candidates;
 2: Output: $SUCCESS$ or $CONFLICT$;

                   ▷ The MultiBind mechanism runs within the scheduler
 3: for all $n \in N$ do
 4:     if RunClusterCommitPipeline($n, j$) = $SUCCESS$ then
 5:         return $SUCCESS$
 6:     end if
 7: end for
 8: return $CONFLICT$        ▷ There was a conflict for all nodes in $N$.

                   ▷ The commit pipeline runs within the Cluster Agent
 9: function RunClusterCommitPipeline($n, j$)
10:     ReserveResourcesInCache($n, j$)
11:     Lock($n$)

12:     $n^* \leftarrow$ FetchNodeInfo($n$)
13:     $J \leftarrow$ FetchJobsOnNode($n^*$)
14:     $n^* \leftarrow$ ComputeAvailableResources($n^*, J$)

15:     if RunCheckConflictsPlugins($j, n^*$) = $CONFLICT$ then
16:         UnreserveResourcesInCache($n, j$)
17:         $result \leftarrow CONFLICT$
18:     else
19:         Commit($j, n^*$)
20:         $result \leftarrow SUCCESS$
21:     end if

22:     Unock($n$)
23:     return $result$
24: end function
```

Step 1. Lines 3–8 represent the MultiBind mechanism, which executes in the scheduler. It iterates over the list of the m highest scored nodes and tries to commit the job to every node, stopping and reporting a scheduling success if the commit succeeds. If all commits fail, a scheduling conflict is reported. Each commit attempt, triggers the commit pipeline in the respective Cluster Agent.

Step 2. Line 10 proactively reserves the job's resources in the nodes cache to make them unavailable for sampling requests. Free resources that are not required by the job are still available for sampling.

Step 3. Line 11 locks the target node within the Cluster Agent such that no other commit pipeline can access it. Unreserved resources on the node are still available for sampling.

Step 4. Lines 12–14 fetch the target node and all jobs currently assigned to it from the orchestrator and, together with information from the nodes cache, compute the currently available resources on the node.

Step 5. Lines 15–17 execute the Check Conflicts plugins to check for a scheduling conflict. If there is a conflict, we undo the resources reservation in the nodes cache carried out in step 2 and prepare to report the conflict to the scheduler.

Step 6. Lines 19–20 run the Commit plugin to commit the job to the node.

Step 7. Lines 22–20 unlock the target node in the Cluster Agent to make it available to other commit pipeline instances again and then return the result to the scheduler.

Reserving resources in the nodes cache is a critical step, because otherwise the sampling pipeline would consider them still available, even though they are currently being bound to a job. Fetching the target node and its assigned jobs is needed, because the nodes cache could be outdated. The Commit stage first creates the orchestrator-specific job object and then binds it to the target node, which completes the commit pipeline.

V. Evaluation & Implementation

To evaluate our scheduler we focus mainly on the scalability aspect at a global scale, while keeping low latency and reducing scheduling conflicts, as described in our contributions. All code to run the experiments, as well as, all results can be found in our repository[6].

A. Implementation

Vela Scheduler and its Cluster Agent are implemented in Go; all their APIs are JSON-based REST APIs. The two largest engineering challenges lie within the Cluster Agent. The first one is the nodes cache, which needs to support a very high read frequency from sampling, but also a considerable write frequency stemming from the commit pipeline and orchestrator updates. The cache supports read-write locking, but to avoid holding locks for a long time, we treat all node objects as immutable. Reading is only done at three points: at the beginning of the sampling pipeline, by the Sampling Strategy plugins, and at the beginning of the commit pipeline. Writing is also done at three points: once at the beginning and at the end of the commit pipeline and when there is a node status update from the orchestrator. The second major engineering challenge is to reserve resources in the nodes cache as early as possible in the commit phase. It is critical to do this immediately for all incoming jobs, before locking the node (this locking only applies to the commit pipelines, not the nodes cache), because otherwise sampling would still consider resources as available, which will be consumed by a job waiting to be committed. At the end of the commit pipeline, each resource reservation is either committed or removed, depending on the outcome of the pipeline. Further implementation details can be found in our code repository.

B. Experiments Setup

To evaluate the scalability of Vela Scheduler we set up 10 globally distributed Kubernetes clusters, which vary in size, depending on the experiment. We run a single instance of Vela Scheduler, which, however, does not limit the distributed nature of our scheduler, because i) the 2-Smart Sampling mechanism is fully distributed and ii) each scheduler instance runs multiple sampling, decision, and commit pipelines independently of each other in parallel, which is the same as running multiple scheduler instances.

To set up the clusters in our testbed we use 10 Google Cloud Platform (GCP) VMs of type `c2-standard-8`, each having 8 vCPUs and 32 GB of memory and running on a physical machine with an Intel Cascade Lake or later processor. Every VM is bootstrapped with Ubuntu 22.04, on top of which we install MicroK8s[7] v1.25.6 to initialize a distinct single-node Kubernetes cluster. For all experiments, we rely on `fake-kubelet`[8] to create simulated nodes in each MicroK8s cluster. The resource properties of these nodes can be easily configured and they are treated as ordinary nodes by Kubernetes. However, `fake-kubelet` nodes do not actually execute any pods (i.e., jobs), but this is not needed for our experiments, since we benchmark the scheduling performance, i.e., until a job has been bound to a node. Sampling performance is also not affected by `fake-kubelet`, because our sampling algorithm works against the Cluster Agent's nodes cache, which is maintained in the background. Other than consuming some CPU time on each VM, the use of `fake-kubelet` does not impact the metrics evaluated in this paper.

Since Vela Scheduler focuses on the Edge-Cloud continuum, the 10 clusters are intentionally not homogeneous. We simulate three Cloud and seven Edge clusters with different types of nodes; the hosting VMs are located in different regions. Cloud clusters are made up of a combination of VMs of three different sizes and Edge clusters consist of a combination of Raspberry Pi[9] single-board computers and cloudlet servers. The node details, the percentage of each node type in the composition of a cluster, and the cluster regions are listed in Table II. These node details serve as realistic configurations

[6]https://polaris-slo-cloud.github.io/vela-scheduler/experiments

[7]https://microk8s.io

[8]https://github.com/wzshiming/fake-kubelet

[9]https://www.raspberrypi.org

TABLE II: Node Types in Cloud and Edge Clusters.

	Node Type & Occurrence (%)	vCPUs	RAM	Regions
Cloud	50% cloud-small	2	4	Belgium, Oregon, Finland
	30% cloud-medium	4	8	
	20% cloud-large	8	16	
Edge	20% Raspberry Pi 4B	4	2	Belgium, Netherlands, Frankfurt, Montreal, Oregon, Finland, Iowa
	40% Raspberry Pi 4B	4	4	
	30% Raspberry Pi 3B+	4	1	
	10% cloudlet	4	8	

for the resource properties of the simulated nodes. There is no difference between simulating a cloud node and a Raspberry Pi node using `fake-kubelet`, because for our experiments only the configured resource properties are of interest. Vela Scheduler itself is also deployed on a `c2-standard-8` VM and is located in the Zurich region. All tests use Apache JMeter[10] as a load generator – we run it on a VM with 24 vCPUs and 47 GiB of RAM. The hosting server at our university has an Intel Xeon CPU (Cascade Lake) with a base clock of 2.1 GHz. JMeter does not allow for configuring a specific request rate per second, but instead requires configuring the number threads that generate requests and the approximate timing they should use, e.g., one request every 10 milliseconds.

We run three sets of experiments: i) *configuration tuning* to find optimal settings for Vela Scheduler, ii) *scalability with respect to infrastructure* to assess the performance of our scheduler on an increasing number of nodes, and iii) *scalability with respect to workload* to a assess the performance on an increasing scheduler workload.

Configuration Tuning aims to find optimal values for C_p and N_p, i.e, the percentage of clusters and the percentage of nodes to sample in 2-Smart Sampling. To this end, we evaluate settings of $C_p = \{10\%, 20\%, ..., 100\%\}$ and, for each value, run an experiment iteration with $N_p = \{4\%, 8\%, 12\%, 16\%\}$, for a total of 40 iterations. Each iteration tries to schedule $11,200$ jobs requiring 4 vCPUs and 4 GiB of RAM on clusters of $2,000$ (2K) nodes each, adding up to 20K nodes in total. $11,200$ is the maximum number of jobs of this size that this cluster configuration can support, thus the scheduler must find all available space to avoid scheduling failures. Additionally, 50% of the nodes are too small to host such a job.

The two scalability experiments use the settings determined by the configuration tuning to evaluate the scalability of Vela Scheduler. Akin to the previous experiment, each scalability experiments uses 10 clusters, each of which contains a tenth of the total nodes in the experiment, i.e., for 1K total nodes each cluster contains 100 nodes and for 20K total nodes each cluster contains 2K nodes.

The experiment on scalability with respect to infrastructure schedules 1K jobs on increasing cluster sizes, specifically 1K, 5K, 10K, 15K, and 20K total nodes (for comparison, Kubernetes officially only supports 5K total nodes [25]). We run three iterations for each of these cluster sizes. The jobs intentionally fit on each node to allow us to focus on measuring the execution performance.

[10]https://jmeter.apache.org

The scalability experiment with respect to workload operates on 20K total nodes (i.e., 2K nodes per cluster) and gradually increases the scheduler workload across 15 iterations, each lasting three minutes. In this experiment the jobs are heterogeneous; specifically each JMeter thread iteration creates one job requiring 1 CPU and 1 GiB, two jobs needing 2 CPUs and 2 GiB, and one job requiring 4 CPUs and 4 GiB of RAM. We intentionally use CPU and RAM requirements only, because adding battery or geo-location requirements would reduce the number of eligible nodes and, hence, saturate the clusters sooner. Each job counts as one scheduling request. We use thread and timing configurations for JMeter to achieve job rates between 15.18 requests/sec and 290.36 requests/sec.

C. Results

1) Configuration Tuning: For this experiment we focus on finding the lowest values for C_p and N_p that yield zero scheduling failures. We aim for the lowest configuration values, because sampling fewer (globally distributed) clusters and fewer nodes within each cluster naturally leads to faster execution times than sampling more clusters and/or nodes. Since rescheduling attempts are common in distributed schedulers, we consider a job to have failed scheduling, only after having attempted rescheduling a total of ten times without success.

Fig. 4 shows the number of scheduling failures as a percentage of the total jobs. It is evident that the number of failures decreases as the number of sampled clusters increases, because the scheduler has more nodes to choose from. The failures first reach zero at $C_p = 50\%$ and $N_p = 4\%$, which is what we will use for the remaining experiments. At $C_p = 60\%$ and $N_p = 4\%$, there is a single failure, but starting at $C_p = 70\%$, there are no more failures, which is why we have excluded larger C_p values from Fig. 4 for clarity. The full set of results, including the number of rescheduling attempts, is available in our repository. For the remainder of this paper we use the above mentioned lowest C_p and N_p values that resulted in no failures, i.e., perfect scheduling, within this experiment. However, future work may investigate dynamic adaptation of these values, because as the utilization of the clusters increases or decreases, different C_p and N_p values may be needed to maintain a low number of failures and scheduling conflicts.

2) Scalability with Respect to Infrastructure: This experiment focuses on evaluating the performance of Vela Scheduler

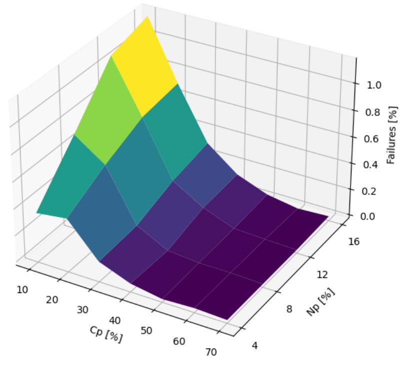

Fig. 4: Scheduling Failure Percentages for Configuration Tuning.

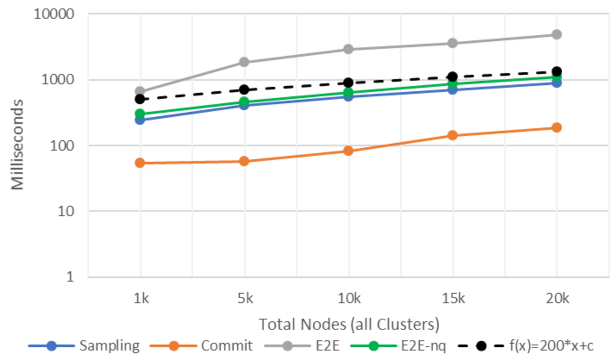

Fig. 5: Mean Scheduling Times (ms) at $C_p = 50\%$ and $N_p = 4\%$ for Total Nodes.

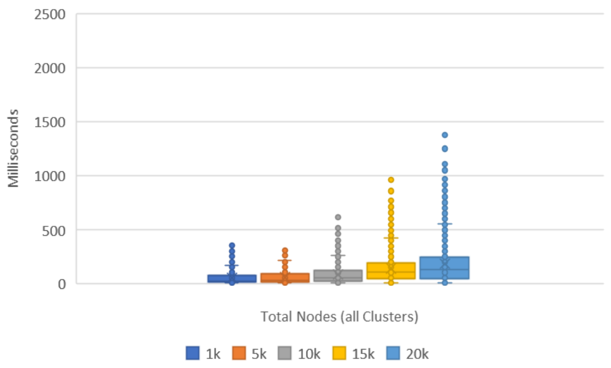

Fig. 7: Commit Times (ms) at $C_p = 50\%$ and $N_p = 4\%$ for Total Nodes.

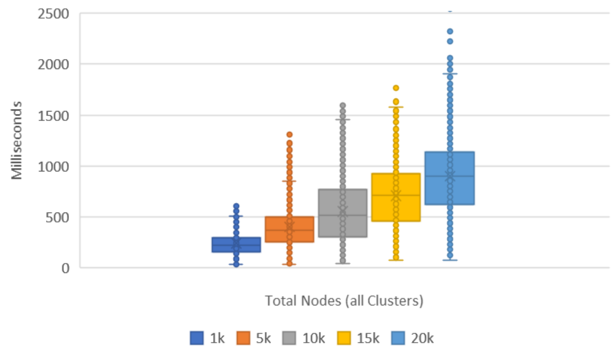

Fig. 6: Sampling Times (ms) at $C_p = 50\%$ and $N_p = 4\%$ for Total Nodes.

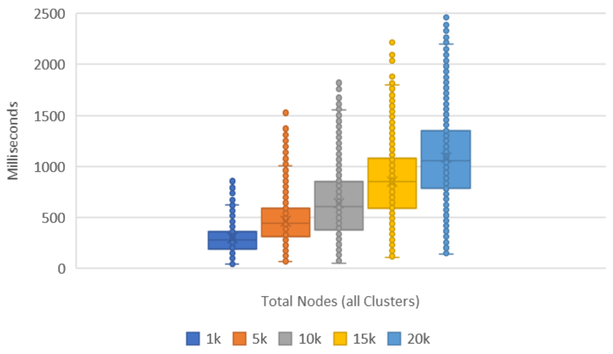

Fig. 8: End-to-End Times (ms), without Sampling Queue, at $C_p = 50\%$ and $N_p = 4\%$ for Total Nodes.

on increasing cluster sizes to show its scalability. We examine execution times of the sampling phase, the commit phase, and the E2E times, i.e., the time from adding a job to the sampling queue until a successful end of the commit phase. Since we noticed a significant latency increase of the MicroK8s API server under high load (e.g., creating a pod object sometimes took about 8 seconds), we do not include the interaction with Kubernetes in the commit and E2E metrics, instead we fetch node information only from our cache and consider the commit pipeline successful once we make the commit in our cache, before we issue a write request to the orchestrator. This allows us to focus solely on the Vela Scheduler performance.

Fig. 5 summarizes the mean execution times in this experiment, showing a linear increase for all of them. We observe two different E2E times: one including time spent in the sampling queue (E2E) and one without sampling queue time (E2E-no-queue or E2E-nq). When including queuing time, E2E time increases much faster, albeit still linearly. This is because as the sampling duration increases, the threads responsible for step one of 2-Smart Sampling in the scheduler are blocked for a longer time. Since we have 80 sampling threads (CPU cores $\times$ 10) in the experiment, these threads are at some point all waiting for responses and thus many of the $1,000$ jobs that arrive in very quick succession need to stay in the queue longer. This could be alleviated, e.g., by running multiple concurrent scheduler instances.

More detailed breakdowns of the sampling, commit, and E2E-nq times are shown in Fig. 6, Fig. 7 and Fig. 8 respectively. For 1K total nodes, sampling takes a mean of 243.3 ms, which is a reasonable time for getting samples from five globally distributed clusters, considering global packet round trip times (e.g., the Verizon SLA for a Europe-USA packet round trip, including intra-Europe and intra-US round trips is 165 ms [28]). Sampling times increase linearly with the cluster sizes to a mean of 902.1 ms for 20K total nodes. Since the Cluster Agent performs sampling on its nodes cache, which is regularly updated in the background, no communication within the cluster is necessary in this phase. However, as the cluster size increases, the absolute number of nodes per sample also increases, hence more processing time is needed for larger clusters. Commit times increase linearly as well, ranging from 53.1 ms for 1K nodes to 182.8 ms for 20K nodes. Since the commit phase involves only communication with the target cluster, conflicts checking for a single node, cache operations, and possible MultiBind retries, its contribution to the E2E time is fairly low. E2E-nq times also increase linearly from 297.9 ms for 1K nodes to 1087.1 ms for 20K nodes. This shows that most of the time is spent in 2-Smart Sampling, which is reasonable given that all Filter and Score plugins currently run as part of the sampling pipeline.

The MultiBind overhead when trying to commit to all $m = 3$ nodes, compared to succeeding on the first node, varies depending on the communication latency with the selected clusters. However, it is evident from the execution time results that MultiBind provides considerable time savings over the alternative strategy of rerunning the entire Vela Scheduler lifecycle on every scheduling conflict, because this would encompass not only contacting at least one more cluster for

TABLE III: Scheduling Decisions and Throughput.

Req / sec	Queuing Time (msec)	Scheduling Decisions/sec (SDPS)	Throughput w MultiBind (jobs/s)	Throughput no MultiBind (jobs/s)
54	0	54	54	49
72	1	72	72	62
94	6	95	94	75
99	106	100	98	73
133	30,097	110	107	77
175	35,499	238	96	87
212	35,672	384	99	94
254	32,562	608	116	112
290	30,847	817	134	131

committing, but also running the complete sampling phase again.

3) Scalability with Respect to Workload: In this experiment we evaluate all results with focus on the scheduler's *throughput* in *jobs per second (jobs/s)* and the total number of *scheduling decisions per second (SDPS)*. We calculate the throughput by dividing the number of successfully scheduled jobs by the total time the Vela Scheduler was active. This time is calculated using the difference between the scheduling finish timestamps of the last successful job and the first successful job. We compute this value for every iteration of our experiment and round it to the next integer value, giving us a throughput ranging from 15 jobs/s up to 134 jobs/s. The scheduling decisions per second (SDPS) are the total number number of scheduling attempts irrespective of their results (i.e., success, conflict, rescheduling due to no nodes found, or failure due to too many rescheduling attempts) divided by the total execution time. The SDPS range from 15 to 817. We stopped our experiments at this number, because the simulated cluster resources were getting exhausted, thus, leaving too little space for scheduling other jobs.

Table III summarizes the results of this experiment. It shows the request rate generated by JMeter in requests per second (req/s), the mean queuing time of a job before it is dequeued by the sampling pipeline, the SDPS, and the throughput in successfully scheduled jobs per second with and without MultiBind. The mean queuing time and the SDPS are good indicators of whether the scheduler is able to keep up with the incoming workload. Up until 99 req/s the queuing time is negligible, whereas starting with 133 req/s it suddenly rises to 30 seconds. Likewise, the SDPS are equal to or greater than the request rate up until 99 req/s and start lagging behind at 133 req/s. The throughput with MultiBind remains approximately equal to the input request rate (the difference of 1 in the row with 99 req/s is caused by rounding, the actual difference is less than 0.042), until 133 req/s, where it starts to fall behind. These values indicate that the single-instance configuration of Vela in the experiments can reliably sustain the scheduling of an input workload of approximately 100 req/s. While this is sufficient for our AR use case, Vela is capable of much higher SDPS, as we discuss in the next paragraph. The sudden increase in queuing time is due to the sampling threads waiting for responses from the Cluster Agents. A maximum CPU usage of 93% in the scheduler VM indicates that the current thread configuration is ideal and that the scheduler needs to be scaled out to further increase performance. Conversely, the Cluster Agents show a peak CPU usage of approximately 26%, indicating that thread-level parallelism could be further increased before scaling out, which we defer to future work.

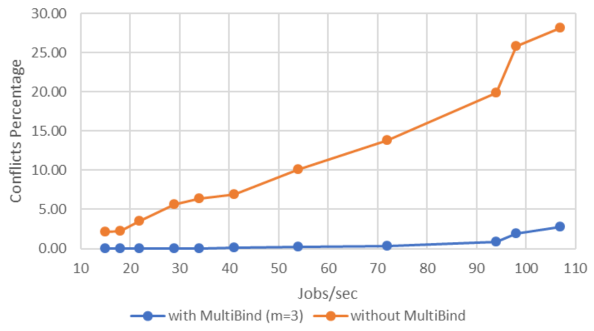

Fig. 9: Scheduling Conflicts with and without MultiBind.

The SDPS show a significant increase after 133 req/s, because of the number of rescheduling attempts, due to not finding suitable nodes. Rescheduling attempts rise from zero until 99 req/s and 0.04% at 133 req/s to 53.4% at 175 req/s and 75.75% at 290 req/s, resulting in up to 817 total SDPS in the last case. This is caused by resources becoming scarce in the cluster, which leads to not finding any suitable nodes during sampling. However, this shows that a single Vela Scheduler instance is capable of supporting high numbers of SDPS, while managing clusters of 20k total nodes.

As previously noted, scheduling conflicts are common in distributed schedulers. Their occurrence rate should be as low as possible to avoid rescheduling jobs, which consumes processing time. In Fig. 9 we examine the percentage of scheduling conflicts of Vela Scheduler with and without the MultiBind mechanism. The number of scheduling conflicts with MultiBind is reported directly by our scheduler, while the number of conflicts without MultiBind is obtained by counting all successful commit phases, where MultiBind retried committing at least once. For the first five experiment iterations there are between zero and two scheduling conflicts with MultiBind. Then, the rate starts increasing gradually, but stays below 1% of the total jobs until a throughput of 94 jobs/s, reaching its highest value of 2.76% at 107 jobs/s. These numbers are very low compared to scheduling without MultiBind. In this case there are 2.09% scheduling conflicts already in the first experiment iteration, gradually increasing up to a maximum of with 28.2% at 107 jobs/s. This clearly shows the benefit of MultiBind; without it, the scheduling time would double or triple for up to 25% of the jobs, because they would need to traverse the Vela Scheduler lifecycle two or three times, due to rescheduling. Altogether the numbers show very promising results, with Vela Scheduler having linear scalability and the MultiBind mechanism being a great improvement over a simple rescheduling on conflict approach.

VI. Conclusion

In this paper we have presented Vela, a globally distributed, orchestrator-independent scheduler for the Edge-

Cloud continuum with a 3-phase scheduling workflow. Its 2-Smart Sampling mechanism delegates sampling to globally distributed clusters, freeing the scheduler from communicating with nodes directly, thus, reducing latency. By considering the requirements of a job during sampling, the clusters produce meaningful samples that only contain nodes that are capable of hosting the job, thus reducing sample wastage and keeping the sample size small. The MultiBind mechanism greatly reduces scheduling conflicts by retrying committing a job to the second or third best suitable node, if the commit to a previous one fails, which significantly increases scheduler throughput. We have evaluated Vela Scheduler on a testbed with 10 clusters with up to 20k simulated nodes, showing its scalability.

As future work we intend to add more plugins to the scheduler's pipelines to add awareness of Service Level Objectives, such as network requirements, and awareness of serverless workflows as described in [40]. Furthermore, we plan to implement AI-based sampling strategies that leverage information on the previous execution of similar jobs to produce even better samples and we want to further improve the scalability of our approach by increasing sampling performance and introducing sharding into the Cluster Agents.

Acknowledgment

The authors thank Víctor Casamayor Pujol from the Distributed Systems Group, TU Wien for plotting the 3D graph.

References

[1] M. Schwarzkopf, "The evolution of cluster scheduler architectures," 2016. [Online]. Available: https://www.cl.cam.ac.uk/research/srg/netos/camsas/blog/2016-03-09-scheduler-architectures.html

[2] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems - EuroSys '15*, L. Réveillère, T. Harris, and M. Herlihy, Eds. New York, New York, USA: ACM Press, 2015.

[3] The Kubernetes Authors, "Scheduling framework — kubernetes," 2021. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/

[4] Docker Inc., "Scheduler design," 2017. [Online]. Available: https://github.com/docker/swarmkit/blob/master/design/scheduler.md

[5] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. Boston, MA: USENIX Association, 2011.

[6] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: Association for Computing Machinery, 2013.

[7] E. Saurez, H. Gupta, A. Daglis, and U. Ramachandran, "Oneedge: An efficient control plane for geo-distributed infrastructures," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. New York, NY, USA: Association for Computing Machinery, 2021.

[8] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: Association for Computing Machinery, 2013.

[9] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: Scalable and coordinated scheduling for cloud-scale computing," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/boutin

[10] C. Delimitrou, D. Sanchez, and C. Kozyrakis, "Tarcil: Reconciling scheduling speed and quality in large shared clusters," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15. New York, NY, USA: Association for Computing Machinery, 2015.

[11] Y. Feng, Z. Liu, Y. Zhao, T. Jin, Y. Wu, Y. Zhang, J. Cheng, C. Li, and T. Guan, "Scaling large production clusters with partitioned synchronization," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 2021. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/feng-yihui

[12] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013.

[13] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga, "Mercury: Hybrid centralized and distributed scheduling in large shared clusters," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/karanasos

[14] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel, "Hawk: Hybrid datacenter scheduling," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/delgado

[15] Z. Han, H. Tan, X.-Y. Li, S. H.-C. Jiang, Y. Li, and F. C. M. Lau, "Ondisc: Online latency-sensitive job dispatching and scheduling in heterogeneous edge-clouds," *IEEE/ACM Transactions on Networking*, vol. 27, no. 6, pp. 2472–2485, 2019.

[16] M. Goudarzi, H. Wu, M. Palaniswami, and R. Buyya, "An application placement technique for concurrent iot applications in edge and fog computing environments," *IEEE Transactions on Mobile Computing*, vol. 20, no. 4, pp. 1298–1311, 2021.

[17] N. Potu, C. Jatoth, and P. Parvataneni, "Optimizing resource scheduling based on extended particle swarm optimization in fog computing environments," *Concurrency and Computation: Practice and Experience*, vol. 33, no. 23, 2021.

[18] X. Qiu, W. Zhang, W. Chen, and Z. Zheng, "Distributed and collective deep reinforcement learning for computation offloading: A practical perspective," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1085–1101, 2021.

[19] T. Pusztai, F. Rossi, and S. Dustdar, "Pogonip: Scheduling asynchronous applications on the edge," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021.

[20] S. Tuli, S. Ilager, K. Ramamohanarao, and R. Buyya, "Dynamic scheduling for stochastic edge-cloud computing environments using a3c learning and residual recurrent neural networks," *IEEE Transactions on Mobile Computing*, vol. 21, no. 3, pp. 940–954, 2022.

[21] H. Ning, H. Wang, Y. Lin, W. Wang, S. Dhelim, F. Farha, J. Ding, and M. Daneshmand, "A survey on the metaverse: The state-of-the-art, technologies, applications, and challenges," *IEEE Internet of Things Journal*, 2023.

[22] B. Sedlak, I. Murturi, and S. Dustdar, "Specification and operation of privacy models for data streams on the edge," in *2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC)*, 2022, pp. 78–82.

[23] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu, "Gaia: Geo-distributed machine learning approaching lan speeds," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 629–647. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/hsieh

[24] X. Qiao, P. Ren, S. Dustdar, L. Liu, H. Ma, and J. Chen, "Web ar: A promising future for mobile augmented reality—state of the art, challenges, and insights," *Proceedings of the IEEE*, vol. 107, no. 4, 2019.

[25] The Kubernetes Authors, "Considerations for large clusters," 2023-01-12. [Online]. Available: https://kubernetes.io/docs/setup/best-practices/cluster-large/

[26] C. Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," *ACM Comput. Surv.*, vol. 55, no. 7, 2022.

[27] C. Curino, S. Krishnan, K. Karanasos, S. Rao, G. M. Fumarola, B. Huang, K. Chaliparambil, A. Suresh, Y. Chen, S. Heddaya *et al.*, "Hydra: a federated resource manager for data-center scale analytics," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[28] Verizon, "Ip latency statistics," 2023. [Online]. Available: https://www.verizon.com/business/terms/latency/

[29] S. Nastic, T. Pusztai, A. Morichetta, V. C. Pujol, S. Dustdar, D. Vij, and Y. Xiong, "Polaris scheduler: Edge sensitive and slo aware workload scheduling in cloud-edge-iot clusters," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021.

[30] T. Pusztai, S. Nastic, A. Morichetta, V. Casamayor Pujol, P. Raith, S. Dustdar, D. Vij, Y. Xiong, and Z. Zhang, "Polaris scheduler: Slo- and topology-aware microservices scheduling at the edge," in *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, 2022.

[31] F. Rossi, V. Cardellini, F. Lo Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with kubernetes," *Computer Communications*, vol. 159, 2020.

[32] Santos José, Wauters Tim, Volckaert Bruno, and De Turck Filip, "Towards network-aware resource provisioning in kubernetes for fog computing applications," in *2019 IEEE Conference on Network Softwarization (NetSoft)*, 2019.

[33] Zhao Hailiang, Deng Shuiguang, Liu Zijie, Yin Jianwei, and Dustdar Schahram, "Distributed redundant placement for microservice-based applications at the edge," *IEEE Transactions on Services Computing*, vol. 15, no. 3, 2022.

[34] Apache Software Foundation, "Powered by mesos: Organizations using mesos," 2022. [Online]. Available: https://mesos.apache.org/documentation/latest/powered-by-mesos/

[35] K. Karanasos, A. Suresh, and C. Douglas, "Advancements in yarn resource manager," in *Encyclopedia of Big Data Technologies*, S. Sakr and A. Zomaya, Eds. Cham: Springer International Publishing, 2018, pp. 1–9.

[36] The Apache Software Foundation, "Apache hadoop 3.3.3: Fair scheduler." [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/FairScheduler.html

[37] ——, "Apache hadoop 3.3.3: Capacity scheduler." [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html

[38] Jing Hui Alex Neo and Lee Bu Sung, "Epsilon: A microservices based distributed scheduler for kubernetes cluster," in *2021 18th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, 2021.

[39] P. Huang, Y. Bai, F. Li, X. Ding, Q. Chen, D. Vij, Du Peng, and Y. Xiong, "Arktos: A hyperscale cloud infrastructure for building distributed cloud," in *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, 2022.

[40] S. Nastic, P. Raith, A. Furutanpey, T. Pusztai, and S. Dustdar, "A serverless computing fabric for edge & cloud," in *2022 IEEE 4th International Conference on Cognitive Machine Intelligence (CogMI)*, 2022, pp. 1–12.