Oakestra: A Lightweight Hierarchical Orchestration Framework for Edge Computing

Giovanni Bartolomeo, Mehdi Yosofie, Simon Bäurle, Oliver Haluszczynski, Nitinder Mohan, and Jörg Ott, *Technical University of Munich, Germany*

https://www.usenix.org/conference/atc23/presentation/bartolomeo

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Open access to the Proceedings of the 2023 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

Oakestra: A Lightweight Hierarchical Orchestration Framework for Edge Computing

Giovanni Bartolomeo[♮] *Mehdi Yosofie* *Simon Bäurle* *Oliver Haluszczynski*
Nitinder Mohan[♮] *Jörg Ott*
Technical University of Munich, Germany

{firstname.lastname@tum.de}

[♮] *Corresponding Authors*

Abstract

Edge computing seeks to enable applications with strict latency requirements by utilizing resources deployed in diverse, dynamic, and possibly constrained environments closer to the users. Existing state-of-the-art orchestration frameworks (e.g. Kubernetes) perform poorly at the edge since they were designed for reliable, low latency, high bandwidth cloud environments. We present Oakestra, a hierarchical, lightweight, flexible, and scalable orchestration framework for edge computing. Through its novel federated three-tier resource management, delegated task scheduling, and semantic overlay networking, Oakestra can flexibly consolidate multiple infrastructure providers and support applications over dynamic variations at the edge. Our comprehensive evaluation against the state-of-the-art demonstrates the significant benefits of Oakestra as it achieves approximately tenfold reduction in resource usage through reduced management overhead and 10% application performance improvement due to lightweight operation over constrained hardware.

1 Introduction

Within almost a decade since its inception, edge computing has found a wide range of use cases in industry and research, especially for supporting latency-critical services like AR/VR [24], live video analytics [14], etc. [46]. However, despite significant interest, there have only been a handful of real-world demonstrations of edge so far [49]. Reasons for this are manifold and may include, on the technical side, the following. Firstly, resources at the edge are far less capable and more heterogeneous than datacenters [61] – usually of smaller form factor with specialized hardware, e.g., Intel NUCs [36], Coral AI board [20]), Jetson Xavier [50], Raspberry Pis, etc. Many such devices are designed to be deployed in proximity to the users utilizing unreliable (wireless) networks with limited bandwidth and high latency as primary communication mediums [64]. Moreover, the benefits of edge [47,68] are only apparent with the dense availability of computing resources – requiring significant investment and planning [21,46].

Secondly, the majority of popularly used orchestration frameworks, e.g., Kubernetes [34], K3s [29], KubeFed [35], etc., are off-shoot branches of solutions that were inherently designed to perform well in managed datacenter networks. Such frameworks make strong assumptions about the underlying infrastructure's (especially the network's) consistent reliability and reachability, which does not necessarily hold at the edge where resources are more dispersed. For example, recent investigations into Kubernetes' operations uncovered that its reliance on maintaining strong consistency in the datastore via etcd along with its limited scalability results in severe availability and efficiency issues in edge-like environments [37]. Moreover, the core components of such frameworks incorporate many heavyweight operations – limiting their use on constrained hardware. Furthermore, almost none of the existing solutions can currently support the edge's heterogeneity in hardware, networking, and resource availability.

In this work, we present Oakestra, a flexible, hierarchical orchestration framework that overcomes the many challenges just mentioned. Conceptually, Oakestra allows multiple operators over vast geographical regions to contribute their resources to a federated infrastructure – reducing the investment to achieve a dense computing fabric at the edge. Furthermore, Oakestra's implementation is lightweight and extensible, allowing it to manage effectively constrained and heterogeneous edge infrastructures. Specifically, our contributions are:

(1) We consolidate edge infrastructures in a logical three-tier *hierarchy*. With a root orchestrator managing many resource clusters, each controlled by a cluster orchestrator, we enable infrastructure federation. The cluster orchestrator exercises *local* fine-grained control but only sends aggregated cluster usage statistics to the root (§3). By design, Oakestra hides the internal infrastructure details within each cluster, allowing many providers to participate without exposing internal configurations. Application providers can deploy services at the edge by specifying high-level constraints (hardware, latency, geography) at the root. Oakestra uses a *delegated scheduling mechanism* that decouples the task placement by only making coarse-grained cluster choices at the root and

leaving fine-grained resource placement to the clusters.

(2) We design a novel semantic overlay network that transparently enables edge-oriented load balancing policies (e.g., connect to the closest instance), directly addressable using semantically-capable IPv4 addresses and hostnames (§3.4). This supports the portability of cloud-native applications, ensuring flexibility for application developers to optimize the service-to-service interactions at the edge. The overlay also allows `Oakestra` to dynamically adjust communication endpoints in response to infrastructure changes, e.g., migrations, failures, etc., ensuring uninterrupted service interactions.

(3) `Oakestra`'s lightweight and modular implementation is compatible with most popular cloud technologies and allows developers to extend internal components, e.g., schedulers, without much development overhead (§5). Our extensive evaluation in both high-performance computing and edge infrastructures demonstrates `Oakestra`'s capabilities as it consistently (and significantly) outperforms the popular production frameworks (e.g., Kubernetes and its derivatives). Our results show up to 10× lower CPU overhead, 60% reduction in service deployment time, and 10% application performance improvement. Under heavy loads, `Oakestra` reduces resource utilization by ≈ 20% compared to its closest competitor, K3s. `Oakestra` is an open-source project (`https://www.oakestra.io/`), and all its components are available at `https://github.com/oakestra`.

2 Background and Related Work

Kubernetes [34] has emerged as the most popular orchestration system in production, used by ≈ 59% of all respondents in a recent survey [23], and has been touted by many as the primary solution for edge computing. It decouples the execution runtime of the applications (*nodes*) from the global cluster decisions (control plane). Its smallest deployable units of computing are the Pods, which are a group of containerized services. The *nodes* embed the execution runtime of the pods, as well as the networking e monitoring components of the platform. The *control plane* exposes the APIs for developers and external tools, monitors and synchronizes the nodes, and reacts to cluster events, such as deployments, scaling, and failures. Kubernetes is designed for datacenter environments, and it assumes the nodes to be high-end managed resources interconnected by reliable low-latency networks. The platform guarantees strong consistency of the cluster status and resources in replicated control plane setup via the distributed key-value store called etcd. Recent studies have found that the strong consistency requirements of `etcd` are its primary limitation when ported to heterogeneous and diverse edge infrastructures. This has a noticeable impact on scalability when it comes to constrained resources that can slow down the entire infrastructure [37]. Network partitioning and multi-clustering still remain critical even in Kubernetes federation [35] as shown in [45]. In particular, distributed geographical areas lack cooperation and awareness of the remaining infrastructure. The inter-cluster communication then requires additional tools like Submariner [9] that requires global state transfer synchronization and prevents scalability. Lightweight distributions of Kubernetes such as KubeEdge [19], K3s [29], and Microk8s [43] either inherit the strong assumptions of kubernetes [15] or are meant to perform better on small scale clusters as later shown in our evaluation. In general, while extending Kubernetes or rearchitecting its components is a viable option, we argue that the effort for largely redesigning numerous of the core components would be substantial and instead re-formulate some of its base assumptions. Therefore, `Oakestra` pursues a different approach, built ground-up with the edge requirements in mind, it offers a familiar environment for current Kubernetes developers while providing flexibility to exploit the proximity to their clients and geo-distributed multi-owner infrastructure deployment. `Oakestra` does not aim at superseding the feature set of Kubernetes but rather fills the gap identified in those contexts where Kubernetes does not fit. Our ongoing work explores the integration of Kubernetes-based cloud clusters.

In the literature, we can also find other systems that have explored effective edge orchestration natively. CloudPath [48] envisions multi-tier on-path computing for deploying stateless functions closer to the clients. HeteroEdge [69] or SpanEdge [58] cater specifically towards streaming applications, FogLamp [65] focuses on data management and VirtualEdge [42] only considers edge servers within cellular networks. From the task scheduling perspective, we might relate to different hierarchical scheduler approaches that distribute tasks on a cloud-edge continuum [12, 18, 28, 38]. However, while these solutions focus on service scheduling, in our work, we must integrate the scheduling problem in a comprehensive orchestration framework offering both service and resource management. The work closest to ours is OneEdge [60], as it offers a hybrid two-tier control plane for managing geo-distributed edge infrastructures. However, we consider `Oakestra` to be a superset of OneEdge as the former is a general-purpose modular framework that allows developers to express geographic (and other) management constraints as scheduler *plugins*. We demonstrate such extensibility of `Oakestra` through an LDP scheduler plugin (§3.2) that optimizes on geographical and latency constraints – similar to OneEdge. Therefore, integration remains a possibility which we leave out for future work.

3 `Oakestra` Overview

Previous research has shown that both service deployment and resource management in distributed edge infrastructures are non-trivial problems, primarily due to the heterogeneity and dynamicity of the environment [16, 41, 66]. Simultaneously, the application providers are likely to deploy multiple

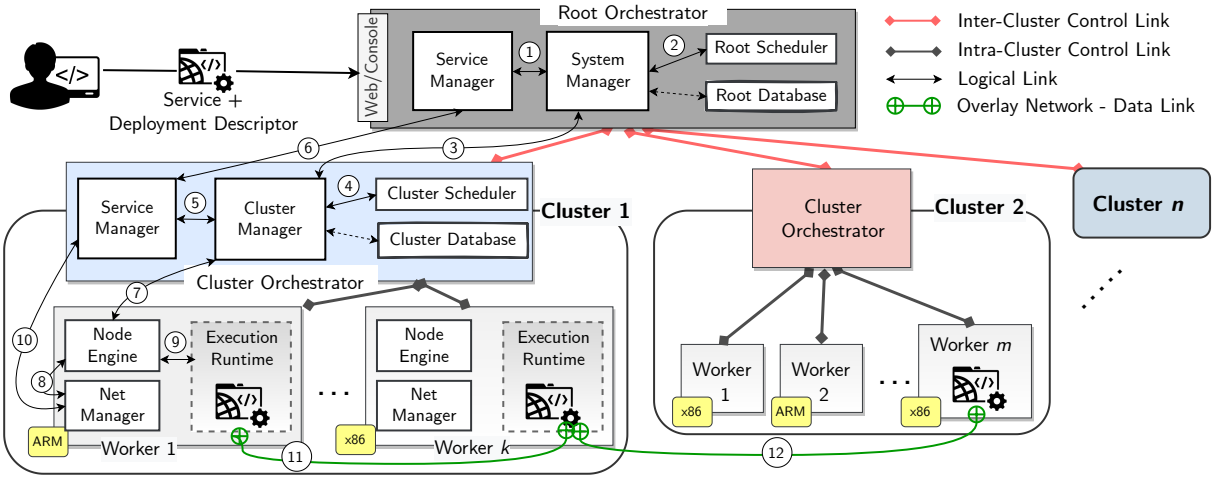


Figure 1: Oakestra Architecture and Workflow.

instances of their services across different edge clusters to have them close to their clients, both in terms of geographic distance and network latency [59]. Additionally, solutions at the edge must remain logically compatible with prevalent cloud techniques such that both existing and novel applications can coexist in edge and cloud realms, thereby providing scalability and flexibility. These and other unique operational viewpoints at the edge impose several design challenges for Oakestra, which we enlist below.

(1) Oakestra must support the infrastructure-at-scale – allowing scaling from thousands to millions of distributed nodes without management overheads. The framework should support a federated heterogeneous infrastructure deployed across geography and controlled by one or more administrative entities and topologies. Furthermore, the framework should allow (a) developers to utilize edge resources regardless of ownership and (b) infrastructure providers to retain management control over their resources.

(2) Extending the infrastructure to the edge of the network requires applications to be able to exploit the proximity with clients. The platform must envision a way for inter-connected microservices to seamlessly communicate and balance the traffic with nearby instances. Therefore, the framework must allow developers to describe the application's requirements with fine-grained SLA primitives (such as specialized hardware requirements, geographical placement, etc.), which must be respected throughout the application lifecycle.

(3) The orchestrator must consider the most up-to-date constraints of edge servers and must adapt to changes in conditions without impacting the applications. Each edge device can contribute with diversified hypervisors, drivers, hardware availability, network, and capacity. The system must abstract the management complexity and autonomously find a compatible node for the deployment issued by the developer.

3.1 System Architecture

Oakestra is a hierarchical orchestration framework for enabling running edge computing applications on heterogeneous resources (Figure 1). Instead of the flat management (inherent to most orchestration solutions [10, 19, 29, 31, 35]), Oakestra organizes the infrastructure into distinct *clusters* (see clusters 1 and 2). We leave the definition of "cluster" purposefully abstract and up to the liking of operators since Oakestra allows multiple edge operators (e.g., ISPs, cloud operators, etc.) to contribute their local deployments towards a shared infrastructure as separate clusters with independent administrative control. Each individual provider then deploys several clusters to segregate its resources, e.g., geographically. The fine-grained control of resources (*workers*) within a cluster is administered by the *cluster orchestrator* (operated by the provider), while *root orchestrator* coarsely controls the global infrastructure. Oakestra can also mimic the single master frameworks (like Kubernetes) with both root and cluster orchestrators deployed on the same machine, albeit with significant performance benefits to the state-of-the-art (§5). The resource and service management responsibilities are separated into independent components, `system manager` and `service manager` (§3.2). We carefully design and implement Oakestra as a modular and extensible framework – allowing the possibility to swap technologies and/or add new features as the requirements of edge computing evolve in the future (§5). Oakestra comprises of three main entities – *root orchestrator*, *cluster orchestrator*, and *worker*.

The **Root Orchestrator** is Oakestra's centralized control plane (analogous to Kubernetes's "control-plane" [10]) and is responsible for managing resource clusters. However, as we explain in §3.2, the root only provides (i) coarse high-level control and (ii) interactions across multiple clusters, as fine-grained control is retained within the cluster boundary. Regardless, we envision the root to be deployed on a machine

reachable from all clusters (in a widescale deployment, e.g., in the cloud). While the root orchestrator may appear to be a central point-of-failure initially, the context separation into fine- and coarse-grained management responsibilities across hierarchy allows Oakestra to continue its operations if the root fails and restarts (see *Fault Tolerance in §3.5)*.

To deploy applications, developers submit the code along with an SLA descriptor to the system manager. The SLA includes high-level operational requirements and constraints for service execution at the edge, e.g., virtualization, required hardware, geolocation, etc. (see §3.2). The system manager notifies the service manager of the new deployment request (step ①), and contacts the root scheduler (step ②) to calculate a priority list of clusters (based on aggregate information) to deploy the application. As such, Oakestra follows a multi-step *delegated service scheduling* approach as the root offloads the fine-grained scheduling operation to selected clusters schedulers (details in §3.2). The system manager is also responsible for registering new clusters and coordinating the control information.

The **Cluster Orchestrator** is a logical twin of the root but with management responsibility restricted to resources within the local cluster. An infrastructure provider registering its resources as a cluster with the root assigns the orchestrator role to a machine that is ideally reachable by all workers. The cluster manager periodically updates the root with *aggregated* statistics of overall cluster utilization and health/QoS of the deployed services (step ⑤, and ⑥) via HTTPS-based *inter-cluster control link* in a *push-based* manner (implementation details in §5). Note that the cluster orchestrator withholds minute information and retains majority administrative control of its member workers. For example, if the cluster orchestrator receives a delegated scheduling request from the root (step ③ and ④), it calculates the optimal resource selection considering the up-to-date availability, utilization, and capability reported by the attached workers. The Oakestra scheduler is designed to be modular and supports several different scheduling algorithms as language-agnostic *plugins*.

Worker Nodes are edge servers in clusters responsible for executing services. Each worker has a distinct capacity and capability, e.g., CPU, GPU, disk, RAM, etc., which it reports to the local cluster orchestrator at registration. If a worker's capacity and capability match the service's SLA constraints, the cluster orchestrator instructs the worker's NodeEngine to deploy the service ⑦, triggering a runtime (and network) instantiation ⑧ and service execution ⑨. Each worker periodically reports its utilization, health of operational services, and (potential) SLA default alarms to its cluster orchestrator via an MQTT-based *intra-cluster control link*. Note that for interconnecting microservices deployed across clusters, Oakestra does not require workers to have public IP addresses as the net manager natively supports both direct ⑪ and tunneled ⑫ communication (see §3.4 for details).

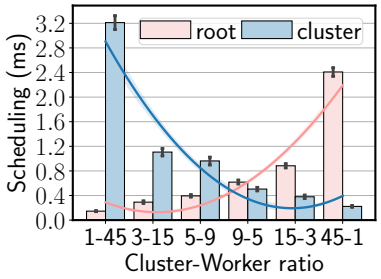


Figure 2: Scheduling time for different cluster sizes.

Why hierarchical orchestration? To understand if a hierarchical management design is inherently better for scalability at the edge, We created an experimental setup with 45 worker node VMs in a compute cluster and configured the orchestration with increasingly different cluster sizes. Starting from a single cluster with 45 nodes, we gradually increased the number of clusters up to 45 with only one worker. For each split, we performed 100 deployments and recorded the time the root and cluster scheduler took for the scheduling decision (see fig. 2). Note that the corner cases of the configuration, i.e., one cluster with 45 workers ($1-45$) and 45 clusters with one worker each ($45-1$), represent the flat orchestration design used in majority state-of-the-art, e.g., in Kubernetes-based solutions. The results indicate that decoupled orchestration reduces scheduling time, primarily since the infrastructure search space is a smaller subset. Optimal performance is achieved when the workers are somewhat balanced across multiple clusters in the hierarchy, and the minimum is around nine clusters with five workers. It is apparent (from the $3-15$ split) that using a two-layer hierarchy can be advantageous, especially for wide-scale infrastructure.

3.2 Resource and Service Management

Resource Management. As discussed in §3.1, edge resources in Oakestra are deployed in distinct clusters, with $R^i = \{R^i_1, R^i_2, \ldots, R^i_n, R^i_{CO}\}$ resources in the ith cluster. Here, R^i_{CO} is the cluster orchestrator of i-th cluster. Each resource R^i_n periodically *pushes* its current utilization (U^i_n) and other characteristics (e.g. location) to R^i_{CO} with update frequency $\lambda(R^i_n)$ over the intra-cluster communication channel. At each update, R^i_{CO} calculates the available capacity of R^i_n by correlating U^i_n to the maximum capacity C^i_n reported at registration. $\lambda(R^i_n)$ can be dynamically tuned for each R^i_n to balance between network overhead and the freshness of the information. We leave the exploration and impact of $\lambda(R^i_n)$ algorithms and the use of techniques such as AoI [62] to future work. Similar to intra-cluster, the update messages over inter-cluster links are also push-based. Each cluster orchestrator periodically sends the aggregate distribution of *available* current capacity, i.e. $\cup(A^i) = \langle \sum(A^i), \mu(A^i), \sigma(A^i) \rangle$ where $A^i = \{A^i_1, A^i_2, \ldots, A^i_n\}$ to the root. The aggregation allows different operators to (i)

```
constraints: [{
    microservice_id: {type: number},
    properties: [{
        memory: {type: integer},
        vcpus: {type: integer},
        vgpus: {type: integer},
        vtpus: {type: integer},
        bandwidth_in: {type: integer},
        latency: {type: number},
        area: {type: string},
        location: {type: string},
        threshold: {type: number},
        rigidness: {type: number},
        convergence_time: {type: integer},
        virtualization: {type: string},
        ... }],
    ...}]
```

Schema 1: Service Requirement Descriptor.

participate in the federated infrastructure while obscuring the minute details of their resources and (ii) freely scale up/down their cluster density without involving the root.

Service Deployment & Scheduling. Developers can deploy their (standalone or multi-microservice) applications by specifying high-level QoS requirements within the SLA description. The deployment descriptor used for submitting applications to `Oakestra` is composed of (i) a description of each microservice of the application (name, namespace, image, etc.) along with communication interlinks, and (ii) a service level agreement (SLA) describing the constraints that the platform must respect for each one of them. Schema 1 shows a fragment of the high-level SLA description supported by our framework. In addition to operational requirements already prevalent in cloud environments, such as processing performance, networking requirements, virtualization needs, etc., the schema allows developers to specify edge-specific restrictions, e.g., geographical location, specialized hardware, etc. Additionally, developers can fine-tune the precision of scheduling heuristics by enforcing *convergence time* and *decision rigidness* metrics. Convergence time specifies the maximum allowed time within which the scheduler should find the suitable edge server that supports the SLA requirements of the service, and rigidness defines the sensitivity for re-triggering service scheduling in case the selected resource violates the SLA (due to environment/infrastructure changes).

As described earlier, `Oakestra` follows a two-step *delegated scheduling* mechanism. Specifically, upon receiving a service deployment request from the developer, the `root scheduler` matches the SLA constraints to the current capacity of each cluster and calculates a priority list of best-fit clusters based on the latest aggregate cluster usage distribution. The root then offloads the deployment request (including SLA and the service) iteratively to cluster orchestrator(s) with decreasing priority. Upon receiving the request, the `cluster scheduler` calculates the optimal service placement within its cluster, leveraging the available schedulers (see §3.2.1). Note that we design `Oakestra`'s scheduling logic to be *language-agnostic* – allowing developers/researchers to implement custom algorithms as *plugins*.

Algorithm 1: Resource-Only Match

Input: A_n : Information about worker n.
$Q_{\tau_{p,i}}$: Requirements of i-th task of p-th service.
$f(A_n, Q_{\tau_{p,i}})$: Resource selection strategy.
Output: Best worker W to run $\tau_{p,i}$.

// Resource selection strategy examples:
// $f(A_n, Q_{\tau_{p,i}}) = *arg\,max_n \big[(A_n^{cpu} - Q_{\tau_{p,i}}^{cpu}) + (A_n^{mem} - Q_{\tau_{p,i}}^{mem})$
// $\wedge\, Q_{\tau_{p,i}}^{virt} \in A_n^{virt}\big]$
// $f(A_n, Q_{\tau_{p,i}}) = *first_n \big[Q_{\tau_{p,i}}^{cpu} \le A_n^{cpu} \wedge Q_{\tau_{p,i}}^{mem} \le A_n^{mem}$
// $\wedge\, Q_{\tau_{p,i}}^{virt} \in A_n^{virt}\big]$
1 $W \leftarrow f(A_n, Q_{\tau_{p,i}})$
2 **return** W

`Oakestra`'s delegated scheduling significantly reduces the search space of the multi-objective task placement problem by considering a subset of resources at each step. In case all microservices of an application cannot be placed within the same cluster, the root scheduler iteratively requests other clusters in the priority list for pending deployment(s). The worker `node engine` also keeps track of the deployed services through their lifecycle (see §3.3). In case of *failures* (resource – if the last update from a worker exceeds a threshold; service – if a worker raises an alarm), the cluster orchestrator marks all affected services as failed and attempts to re-deploy them on another suitable resource within the same cluster. If unsuccessful, the rescheduling request is propagated to the root for system-wide scheduling. Similarly, the cluster orchestrator can trigger *re-deployments* if it observes any SLA violations (exceeding specified rigidity).

3.2.1 Service Schedulers in `Oakestra`

Let $S = \{s_1, s_2, \ldots, s_{|S|}\}$ denote the set of services requested to be deployed by the developers at the root. Each service $s_p \in S$ can be composed of n individual microservices or *tasks*, i.e. $s_p = \{\tau_{p,1}, \tau_{p,2}, \ldots, \tau_{p,n}\}$ where $\tau_{p,i}$ denotes i-th task of p-th service. Each task $\tau_{p,i}$ requires a certain capacity (CPU, GPU, memory), denoted by $Q_{\tau_{p,i}}$. Other considerations like geographical location or virtualization technology, specified by the developer in the SLA, are also part of $Q_{\tau_{p,i}}$. The task of the scheduling components (in both root and cluster) is to find a suitable resource in the infrastructure that supports the requirements in $Q_{\tau_{p,i}}$. In this work, we propose and incorporate two different scheduling approaches.

(1) Resource-Only Match (ROM): As the name suggests, in ROM, the cluster scheduler finds a suitable resource that satisfies the service's capacity requirements (see Algorithm 1). The scheduling approach is analogous to greedy-fit and knapsack-based solutions popularly used for placing VMs on cloud servers in datacenters [63].

(2) Latency & Distance Aware Placement (LDP): LDP (shown in Algorithm 2) builds on the ROM scheduler but additionally considers latency and geographical distance con-

Algorithm 2: Latency & Distance Aware Placement

Input: A_n : Information about worker n.
$Q_{\tau_{p,i}}$: Requirements of i-th task of p-th service.
Output: Best workers W to run $\tau_{p,i}$.

$W \leftarrow \{n \in [1,|A|] \,|\, A_n^{cpu} \geq Q_{\tau_{p,i}}^{cpu} \wedge A_n^{mem} \geq Q_{\tau_{p,i}}^{mem} \wedge Q_{\tau_{p,i}}^{virt} \in A_n^{virt}\}$
if $|Q_{\tau_{p,i}}^{s2s}| \geq 1$ **then**
  for Q_j *in* $Q_{\tau_{p,i}}^{s2s}$ **do**
    $t \leftarrow Q_j^{trg}$
    $W \leftarrow \{n \in W \,|\, dist_{gc}(A_n^{geo}, A_t^{geo}) \leq Q_j^{geo\_thr} \wedge dist_{euc}(A_n^{viv}, A_t^{viv}) \leq Q_j^{viv\_thr}\}$
  end
end
if $|Q_{\tau_{p,i}}^{s2u}| \geq 1$ **then**
  for Q_k *in* $Q_{\tau_{p,i}}^{s2u}$ **do**
    $u \leftarrow Q_k^{lat\_trg}$
    $rtts \leftarrow \{rtt_{i,u} \,|\, i \in rnd(W), rtt_{i,u} = ping(i,u)\}$
    $vivaldiNet \leftarrow \{A_n^{viv} \,|\, n \in [1,|A|]\}$
    $U \leftarrow trilateration(rtts, vivaldiNet)$
    $W \leftarrow \{n \in W \,|\, dist_{gc}(A_n^{geo}, Q_k^{geo\_trg}) \leq Q_k^{geo\_thr} \wedge dist_{euc}(A_n^{viv}, U) \leq Q_k^{lat\_thr}\}$
  end
end
return W

straints for service placement. Since edge applications can be composed of multiple microservices that can either interact with each other (in a chain-like fashion) or directly with end users/devices, we allow the application provider to specify constraints for both service-to-service (S2S) and service-to-user (S2U) links. The root scheduler first filters unsuitable clusters by comparing their resource constraints along with approximate geographical operation zones to the SLA requirements. Within each cluster, the algorithm first creates a list of candidate workers that satisfy the resource constraints. Then, for all S2S constraints $Q_{\tau_{p,i}}^{s2s}$, the algorithm filters out workers that exceed the specified distance $Q_j^{geo\_thr}$ and latency thresholds $Q_j^{viv\_thr}$ to the target service $t = Q_j^{trg}$. LDP estimates geographic distance as the great circle distance ($dist_{gc}$) between the geographic location of worker n (A_n^{geo}) and the location of the target service A_t^{geo}. The approximated latency is the Euclidean distance ($dist_{euc}$) between the location of worker n (A_n^{viv}) and the location of the target service A_t^{viv} in the Vivaldi network [25]. Vivaldi is a network coordinate system embedding networked nodes into a d-dimensional coordinate system such that the Euclidean distance of two nodes approximates their round-trip time. If the developer has specified any S2U constraints $Q_{\tau_{p,i}}^{s2u}$, LDP measures the round-trip times ($rtts$) to the target as $Q_k^{lat\_trg}$ from a set of random workers in the cluster ($i \in rnd(W)$). The measurements approximate the user's position within the Vivaldi network via trilateration. Following that, LDP filters out workers that exceed the distance threshold $Q_k^{geo\_thr}$ to $Q_k^{geo\_trg}$ or the latency threshold $Q_k^{lat\_thr}$ to the approximated user position U.

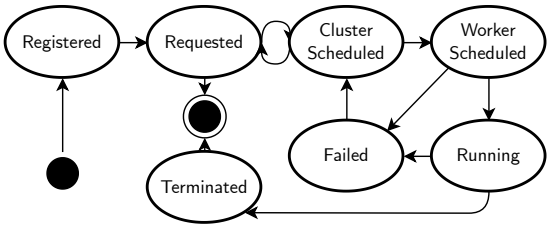


Figure 3: Lifecycle of a service's instance.

3.3 Application Lifecycle

In `Oakestra` an application is composed of multiple services. Each service, in turn, is composed of multiple instances. `Oakestra` keeps track of the instances deployed in the platform through a lifecycle state machine (fig. 3). At any point in time, an instance can have one of the following states.

`Registered`: The developer has submitted an application and its SLA (shown in fig. 1). The service manager in the root saved the service SLA and generated the instance metadata.

`Requested`: After receiving the deployment command, the instance is sent to the root scheduler, and the system manager awaits a suitable cluster. When there is no cluster able to guarantee the SLA, the deployment cannot be carried out.

`Cluster Scheduled`: The root scheduler designated a suitable cluster. The instance SLA is sent to the corresponding cluster orchestrator. The cluster orchestrator is now waiting for the cluster's scheduler decision. If no worker node is capable of hosting the instance, the request is sent back to the root in `Requested` status.

`Worker Scheduled`: The cluster scheduler finds a suitable worker node. The instance binaries (or image download details) are fetched from the root and transferred to the selected worker node while the ports for networking are allocated.

`Running`: The instance is operational as it satisfies the SLA constraints. The worker periodically tracks the current QoS and relays it to the cluster orchestrator along with current utilization in periodic *heartbeat* update messages.

`Terminated`: The service is no longer operational due to an explicit `terminate` command from the cluster orchestrator (issued by the developer at the root). Alternatively, the service gracefully finishes its execution and terminates.

`Failed`: The service execution has stopped with unexpected exit status. A service "fails" if the worker node explicitly reports this state to the cluster orchestrator as a failure alarm or if the node fails. A resource is considered "failed" if it has not sent a *heartbeat* for longer than a pre-defined threshold.

3.4 Service Communication

Supporting intra-service (and service-to-user) networking at the edge can be challenging since (i) infrastructures are susceptible to dynamic changes, (ii) application deployment is

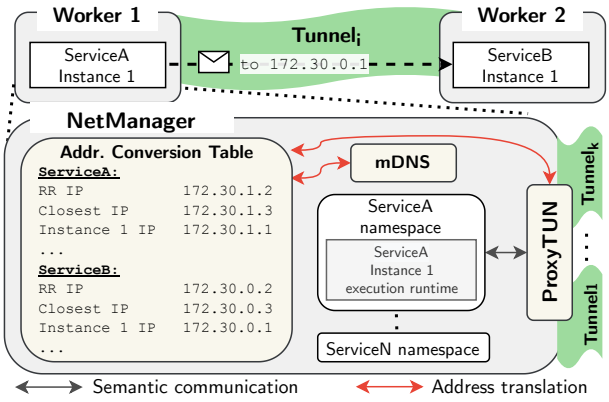


Figure 4: Service communication across edge servers.

fluid to remain close to clients [67], and (iii) several service instances can coexist simultaneously to achieve broader coverage, therefore addressing and balancing must dynamically adapt to the service placement. Moreover, it is impractical to presume that edge servers from multiple participating infrastructure operators can interact over a common/public network – an assumption implicit in the majority of existing solutions [29, 34, 43]. `Oakestra` includes a component called `NetManager` that enables: (a) dynamic routing policies transparently enforced via semantic service addressing to support load balancing catering to edge environments (§3.4.1) and (b) transport layer packet tunneling to interconnect services operating on resources with limited accessibility (§3.4.2).

`Oakestra` separates the bulk of data-plane complexity from the control-plane operations at the worker level. Particularly, the framework utilizes the branches in the multi-tier edge hierarchy only for core control-plane information propagation (e.g., routing updates, service/hardware utilization, and failures, etc.) and designates data-plane communication management to leaf workers using the `NetManager` component. Figure 4 shows the cross-section of `NetManager` with communication between services A and B deployed on workers 1 and 2, respectively. The `proxyTUN` actively maintains and dynamically adjusts endpoints of the tunneled connections to adapt to infrastructure changes, ensuring uninterrupted communication between services. Arguably, the proposed approach resembles some sidecar proxy solutions like Istio [3]. While the set of proposed functionalities might be similar, the `NetManager` does not need to deploy a sidecar along with each deployed application. While the design of solutions like Istio fits the abundance of resources in the cloud context, we propose a lightweight approach with a worker-level proxy used by all the applications, like the native Kubernetes network but featuring additional balancing flexibility and site-to-site tunneling out-of-the-box.

3.4.1 Service Naming and Addressing

Drawing inspiration from semantic routing [39], a `serviceIP` addresses all the instances of the service according to different balancing policies (e.g. *round-robin*, *closest*, etc.). For example, in fig. 4, worker 1 maintains the ServiceA's `closest` and `round-robin` `serviceIP`s – allowing services to select balanced instances through IP addresses. Note that the serviceIP is different from the host IP address as the former is ephemeral and addresses service instances (similar to ClusterIP in Kubernetes [34]). When receiving a packet to an address belonging to the load-balanced `serviceIP`, the `ProxyTUN` uses the conversion table to get the corresponding `Instance IP`. If no conversion entry is found, the `NetManager` registers an interest in that route and enquires the cluster component (see (10) in fig. 1). If the cluster orchestrator does not contain the information, the interest registration is propagated to the root. The root either knows the service route or the service does not exist at all, and no route can be propagated. The interest registration also allows the worker to receive future updates regarding the route. When a route is not used, the conversion entry is erased, and the interest is deregistered.

If properly configured, services can use DNS-based naming schemes which resolve to a *serviceIP* using a **mDNS** [57]. We envision a service naming schema that reflects the hierarchy `<instance_number>.<routing_policy>.<service_name>.<service_namespace>.<app_name>.<app_namespace>`. The `app name` and `namespace` portion of the domain is provided by the developer to uniquely address the application (e.g. videoAnalytics.org). The `service name` and `namespace` address different microservices in the application (e.g. ServiceA.default, ServiceB.default, etc.) while the `instance` number uniquely distinguishes individual service replicas. In case the developer does not care about connecting to a specific instance of the service, `<instance_number>` can be set to "any". The unique aspect of the proposed service naming is the `routing_policy`, as it allows developers to offload connection endpoint selection to `Oakestra` based on the current deployment state. For example, the "closest" policy resolves to serviceIP address representing the nearest service instance while "round-robin" balances connection load across multiple instances. Following the example of fig. 4, suppose *ServiceA* must send a request to the closest instance of *ServiceB*. The request can either be addressed using the semantic IPv4 address representing the *closest* routing policy, therefore `172.30.0.3`, or the name `any.closest.ServiceB.default.videoAnalytics.org`. The *ProxyTUN* component will convert the given semantic address to and *instance address* that represents the chosen instance accordingly to the balancing policy. It will then proceed tunneling the packet towards the destination.

The reachability of the services managed by `Oakestra` from external users can be achieved with standard DNS and API getaway solutions. For future extensions of this work, we are investigating techniques that can support client mobility and dynamicity of the endpoints for first-mile computation and fast handovers. In fact, traditional cloud solutions in edge contexts lead to frequent DNS resolutions and a lack of sup-

port for the discovery of services in close user proximity.

3.4.2 Connection proxying and tunneling

Oakestra enables inter-service communication across workers in different clusters with limited available ports (e.g. behind firewalls) through UDP tunneling. We can again refer to the example shown in fig. 4, where *ServiceA* on *worker 1* needs to communicate with *ServiceB* on *worker 2*. Every packet sent from *ServiceA* to *ServiceB* is handled by the **proxyTun** attached to a virtual bridge in worker 1. The *proxyTUN* component resolves the serviceIP of the destination *ServiceB*, selecting the *Instance IP* accordingly to the balancing policy enforced. If no resolution entry is found, the worker node subscribes to the updates regarding this route to the cluster orchestrator. The resolution entries for a service include a list of the available service instances, their respective IP addresses assigned from each local worker's subnetwork (Instance IPs), and each one of the destination worker's address and port that should be used for the tunneling. The tunneling is performed at L4 using UDP. The L4 implementation allows to transparently support all transport protocols (TCP/UDP/QUIC) out of the box. The only requirement in order to support nodes behind NATs and firewalls is to provide a port for the site-to-site tunneling. An open tunnel (i, j) connecting node i to node j for performance reasons is recycled for all the traffic of all the services communicating with instances belonging to those workers. Therefore, each node only has one ingress tunnel and k egress tunnels, one for each of the workers it's currently exchanging traffic with. To support router configurations without any open incoming ports we envision, in the future, allowing service's connections to transit via the cluster orchestrator. In this case, the *service manager* acts as a VPN server that tunnels the traffic between worker nodes.

If the *NetManager* cannot forward a message (e.g., due to out-of-date information), the route is immediately deleted, and a route refresh is performed. This mechanism avoids imposing a strong consistency requirement on the worker's caches. While this approach might miss the best balancing option when a route update is still propagating asynchronously, it reduces the synchronization effort and, thus, overhead.

We note that by assigning the majority of networking complexity to the worker node, Oakestra dramatically reduces the overhead of orchestration machines. The *service manager* sends only the most relevant routes for each service (according to an internal worker-wise priority list) to increase the scalability and avoid further congestion on the workers. The inherent decentralized networking design of Oakestra is also tolerant towards infrastructure failures. For instance, coupled microservices will continue to communicate with each other even if the cluster (and root) orchestrator becomes unavailable – as long as the host worker node is operational and the endpoints do not migrate (to a different cluster).

3.5 Fault Tolerance.

We now explore the different possible failure cases and how Oakestra manages them. While Oakestra decouples the two-tier *master-worker* orchestration design, prevalent in popular frameworks, in a hierarchical three-tier infrastructure, it is still dependent on the *root*, which may appear as a central point of failure. In case the *root orchestrator* fails, the platform is unable to register new applications as well as schedule new instances to new clusters. Applications that are already part of a cluster can locally replicate, scale, and migrate. The networking is partially affected by root failure since existing tunnels and communication will continue to work, but new inter-cluster links require the root network component for the setup process. Each cluster's aggregated information cannot be propagated, but it will be stored until the root is back online. Since the root is likely to be deployed in the cloud to maximize reachability, it is less frequently affected by failures due to hardware issues. Moreover, in such contexts, traditional redundancy and failover mechanisms can potentially be implemented with redundant replicas, synchronized key-value stores, and L7 load balancing to properly route the cluster's traffic to the active root instance.

A *cluster orchestrator* failure does not affect other clusters' activities. As soon as a cluster stops responding, the root marks the applications deployed within that cluster as failed and attempts to replicate the existing workload to new suitable clusters. The worker nodes will not be able to update their status on the failed cluster orchestrator, but the applications will remain operational. The root will then schedule new backup application replicas in the remaining clusters, and the network routes will reactively change toward the newly managed instances. *Worker node* failures may be frequent and expected at the edge; for this reason, the workloads deployed on a failed node are immediately rescheduled. The network automatically adjusts and balances the traffic to the active instances disregarding the unreachable ones. The cluster orchestrator discontinues aggregating failed node's resources from the cluster's pool in further updates.

In addition to the current fault tolerance strategies and assumptions, in future extensions, we envision having multiple replicas of the orchestrator components that coarsely stay in sync with the primary (similar to the multi-master setup in Kubernetes) and/or a leader election strategy to react to control plane failures. Moreover, we intend to explore the orchestration problem in contexts where byzantine behaviors can be expected at both worker and cluster levels.

4 Implementation

We implement Oakestra and its components (fig. 1) with constraints of heterogeneous edge infrastructures in mind. The implementation, spanning 18000 LOC, is modular, extensible, lightweight and open-source [52]. As we show later,

Oakestra can support production-ready applications at the edge much more effectively than the state-of-the-art.

Root and cluster orchestrators are implemented using Python in a similar structure owing to their architectural similarities. The schedulers, system/cluster manager and service manager are implemented as independent micro-services. The database is implemented as two separate instances of mongoDB [5], one for the service manager and one for the system/-cluster manager. The implementation does not force strong consistency in cluster's or root's data structures. They update asynchronously, improving the overall system's scalability – at the cost of occasionally dealing with application rescheduling. The scheduler microservice uses a celery task queue [1] to pull the incoming tasks and find the best placement asynchronously. Scheduling policies are described in Python but are designed to be language-agnostic. The system and cluster managers communicate with the database services and the service manager via REST APIs. The external root APIs for the infrastructure providers and application developers are implemented according to the Open API Specification [7] and authenticated using JWT tokens [4]. Inter-cluster links are RESTful APIs as well while the intra-cluster control links are implemented using MQTT. With a Mosquitto broker [6] hosted at each cluster orchestrator, a worker can subscribe to network routes updates, and publish internal resource consumption and task status. We also provide an Angular web-based frontend attached to the root that can be used by the developers to facilitate (i) creation of applications and describing the services graphically, (ii) monitor each service instance's status and position (fig. 13b), and (iii) scale the services up and down (see Appendix A).

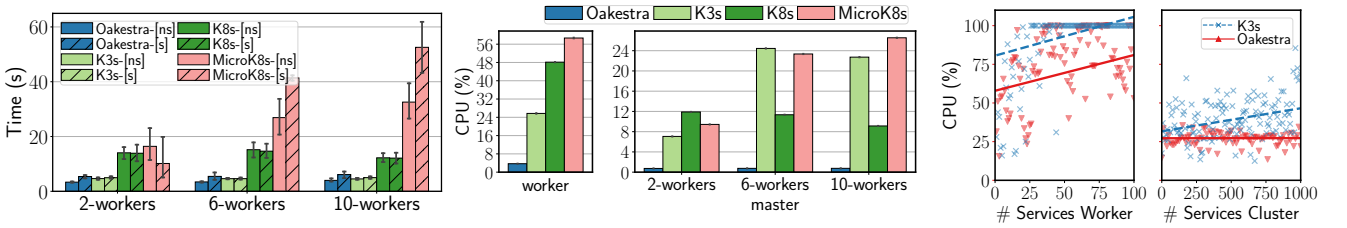
The NodeEngine component as well as the NetManager are implemented as independent services using GoLang to ensure low footprint and maximum support to many execution run-times SDKs. The container's execution runtime is supported with the integration of containerd [2]. Unikernel [40, 44] support is currently under development with QEMU [8] virtualization. The runtime selector is designed to be extended to support even more virtualization technologies in the future, e.g. microVM [11]. The NetManager component uses native Linux virtual network interfaces to create a bridge connected via veth pairs with the TUN interface (namely proxyTUN) and the service's network namespace. The *NetManager* minimizes system context changes and makes extensive use of Goroutines pools to resolve the incoming traffic with high parallelism. The semantic addresses are reserved from a pre-defined 10.30.0.0/16 sub-network. The traffic belonging to the semantic sub-network is forwarded to *proxyTUN*.

5 Evaluation

This section focuses on evaluating Oakestra as compared to the state-of-the-art on edge infrastructures and constrained devices. We use two different testbeds for our evaluation. The *High-Performance Computing (HPC)* testbed is a large, controlled, x86 processor-based cluster, in which we use *S*, *M*, *L*, *XL* VMs with <1,1>, <2,2>, <4,4> and <8,8> <CPU core,RAM (GB)> configurations, respectively. We use this cluster to flexibly spawn resources and emulate a heterogeneous infrastructure. Our *Heterogeneous (HET)* testbed is a local cluster composed of Raspberry Pis [33], Intel NUCs [36], mini-desktops, and Jetson Xavier [50] – representing different edge computing flavors [68]. The HPC cluster is interconnected by 1 Gbps Ethernet, while HET machines connect over Wi-Fi 802.11ac and 1 Gbps Ethernet links. We attempted to compare Oakestra against popular orchestration frameworks. However, despite careful management, KubeFed [35], KubeEdge [19], ioFog [31] and Fog05 [30] experience frequent failures, possibly because they are (i) in early development stages or (ii) not optimized for constrained hardware. Moreover, we could not locate OneEdge's source code [60], which is the only framework architecturally similar to Oakestra. As a result, we compare Oakestra against Kubernetes (K8s) [34] and its two lightweight derivatives, MicroK8s [43] and K3s [29]. All selected frameworks are widely used and have been considered for use with the edge [15, 37]. We use two application workloads, (i) an Nginx web server allowing us to control the operational load dynamically, and (ii) a video analytics application from [13]. The latter is composed of four microservices. The *source* sends a pre-recorded RTP stream [17], *aggregation* stitches and pre-processes each frame, *detection* uses YOLOv3 to detect objects, and *tracking* tracks objects across frames. To remain comparable with the "kube" frameworks, we operate Oakestra in standalone mode, i.e., all workers are deployed within the same cluster. We perform 10 runs for each experiment and clean intermediary files between runs.

5.1 Service Deployment

Figure 5a compares the time taken by each framework to deploy a containerized application on the infrastructure. For this experiment, we configure an *XL* VM as root, an *L* VM as cluster orchestrator in Oakestra (and master for others), and *S* VMs as workers. Oakestra uses the ROM scheduler, which is comparable to the default scheduling policy of the competitors [32]. We increase the cluster size from 2 to 10 workers and measure scheduling overheads by toggling its operation, shown with *s* (with scheduler) and *ns* (no scheduler). MicroK8s performs significantly worse ($\approx 10\times$ slower) than Oakestra, degrading further with increasing infrastructure size. As also noticed in [15], microK8s might easily lead to higher resource usage and generally slower performance. We attribute it to (i) *snap*, which brings extra virtualization overhead, and (ii) microk8s being optimized for single-node deployments. Kubernetes is 2–3$\times$ slower than Oakestra. Its scheduling operation adds almost negligible overhead – this



(a) Deployment time

(b) CPU utilization

Figure 5: Performance comparison for different infrastructure sizes.

Figure 6: CPU usage of worker & cluster orch. in stress (line=median).

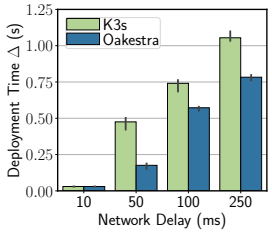


Figure 7: Deployment time with network delay.

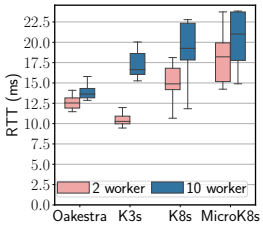


Figure 8: End-to-end latency

result is in line with other recent explorations [15]. One of the reasons why K3s has been generally considered lightweight is its single binary executable that also reduces the internal overhead and synchronization time. This approach, used by `Oakestra` as well, resulted in a comparable deployment time.

Repeating the experiment in our HET testbed, we introduced increasing networking delay using `tc` on the network interface to highlight the overhead due to master-worker synchronization. In fig. 7, we observe that `Oakestra`'s deployment times improve over K3s by a $\approx 20\%$ margin in high delay networks (common for wireless last-mile [22,26,46]). The behavior is similar for lossy networks as `Oakestra` achieved $\approx 50\%$ and 60% deployment time reduction with 20% and 50% losses, respectively (not shown for brevity). The eventual consistent store of `Oakestra` allows the platform to asynchronously manage the deployment with reduced network traffic. As shown later in §5.5, "kube" orchestrators send $\approx 2\times$ more control messages (from both worker and master) on average – hinting at the cause of their degradation to aggravating network conditions. Experiments on the HET cluster typically yielded better results than those on HPC. This is because a more powerful (yet still limited) Raspberry Pi 4 was used as the target device in HET, as opposed to the size *S* worker nodes utilized in HPC.

5.2 Scalability

First, we analyze the idle resource consumption of each framework to estimate the baseline overhead. Lower overhead at the worker indicates a platform's capability to operate on constrained devices. In comparison, lower overhead at the master (at different infrastructure sizes) highlights its ability to handle scale (fig. 5b). We observe that within the chosen competitors, K8s is the one that better handles cluster scalability at the masters, showing no noticeable increase in CPU usage. Meanwhile, K3s and MicroK8s masters show a lower average CPU usage than k8s in two-worker setup but much worse degradation in 6 and 10 workers clusters. At the worker level, k3s is almost 50% faster than K8s. MicroK8s exhibits, again, a higher footprint. We attribute this to the same motivations expressed in §5.1. Due to its asynchronous pub-sub communication and its cluster resource aggregation mechanism, `Oakestra` achieves $\approx 6\times$ and $11\times$ reduction in CPU on the workers and master, respectively. Particularly, an `Oakestra` worker only maintains an MQTT connection to the cluster's broker to periodically report the device resource usage. The orchestrator, subscribed to worker resource topics, only updates the internal database with the latest worker utilization information. Compared to k8s, `Oakestra` maintains a considerably reduced duty cycle while idling.

Since K3s comes closest to `Oakestra` at the worker's level, we perform a stress test comparing both for increasing service deployments. Figure 6 compares the CPU consumption as we increasingly schedule up to 100 Nginx containers on each worker in a 10 node cluster (totaling 1000 containers in the cluster). The left and the right half show the worker and cluster orchestrator (or K3s master) utilization, respectively. `Oakestra` sees negligible overhead performing $\approx$ 10–20% better than K3s – demonstrating its efficacy to support large service volumes. `Oakestra`'s average cluster CPU usage increases by less than 1% during the experiment since each node piggybacks the service status onto the internal resource consumption updates, lowering network usage and processing cycles at the orchestrator. Similarly, `Oakestra` shows significant operational advantages for constrained worker nodes. While K3s exhausted the worker's CPU at $\approx$ 40 services, `Oakestra` deployed the 100 services with a 30% average CPU surplus. Memory utilization showed a similar trend as `Oakestra` achieved $\approx$ 18% and $\approx$ 33% reduction to K3s in worker and master, respectively (not shown).

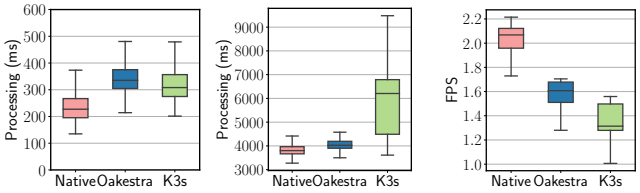
5.3 Networking

We evaluate the round-robin balancing policy of the networking scheme presented in §3.4 against the native balanced *cluster IP* of K3s, K8s, and MicroK8s. First, in a 2-worker setup, we deploy a Python client on the first worker and an Nginx server on the second worker. We then scale the number of workers to 10 and deploy one client and 9 Nginx servers, one on each worker node. The client performs continuous GET requests using a round robin *service IP* on `Oakestra` and a *cluster IP* on the *kube* family with a statically configured round robin policy. Figure 8 shows the average round-trip latency between the client and the closest server. On average, K3s performs better in a 1-to-1 (2 workers) setting (10–20% improvement), while Kubernetes and MicroK8s perform 17% and 30% slower. All competitors' load balancing is significantly worse than `Oakestra` in multiple replica settings, resulting in ≈ 15 up to 35% RTT inflation. The results show the benefits and overhead of the proposed addressing scheme. `Oakestra` performs proxying and site-to-site tunneling for every packet, even in a simple setup with just one client and one server, slightly increasing the overhead even in LAN setups, like the experiment above. On the other hand, this abstraction brings benefits when scaling up the system, introducing minimal additional overhead while balancing with more replicas and outperforming the other systems. In the future, we plan to optimize the 1-to-1 scenario by temporarily disabling the proxy and utilizing VXLAN-based solutions for nodes belonging to the same network.

We evaluated the impact of `Oakestra`'s tunneling on the bandwidth. While the proposed network component is mainly designed to implement semantic addressing, it can also tunnel the traffic between nodes on different networks. For this reason, we test the impact on the bandwidth by comparing it with `WireGuard` [27] – an open-source tunneling solution used by most frameworks. We emulate the network inconsistencies at the edge [46] by gradually increasing the delay between the client and the servers from 10 to 250 ms. Figure 12 compares the time to download a 100 MB file over HTTP using both approaches. We find that, even in high-delay networks, `Oakestra` is always within the competitive range (2-10%) of `WireGuard`. We consider this a promising result given that the proposed networking component performs proxying on top of the "simple" tunneling operations of WireGuard.

5.4 Scheduler Performance

This section shows a preliminary evaluation of ROM and LDP schedulers. Figure 10 depicts both schedulers' performance in a simulated infrastructure with up to 500 edge servers (virtually configured in HPC). We use network latencies between edge servers within 10–250 ms, a typical latency range between users and datacenters globally [21]. We instruct the schedulers (via the SLA) to find workers that satisfy 1 CPU, 100 MB memory, ≈ 20 ms latency (usual for immersive edge applications [46]), and 120 km operational distance. Since ROM only performs a best-fit match for computational requirements, its calculation time does not increase significantly while increasing the number of workers. LDP's calculation time grows increasingly with infrastructure size. However, LDP can effectively support latency-based constraints since it usually satisfies the RTT thresholds even in large edge infrastructures, which implies that the search space increases with the number of nodes. We leave a detailed comparison of different scheduling algorithms to future work as their performance is not the focal point of this paper.



(a) Tracking (b) Detection (c) FPS

Figure 9: Video analytics application performance.

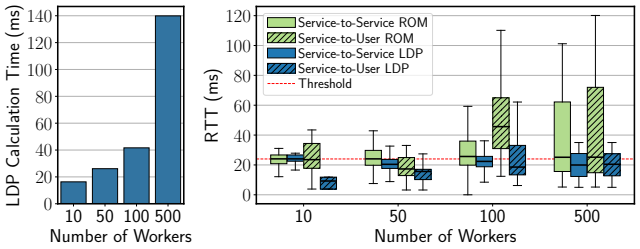


Figure 10: Performance of ROM and LDP schedulers.

5.5 Control Communication Overhead

Figure 11 shows a comparison between the number of exchanged control messages of `Oakestra` and K3s. First, we compare the K3s master and `Oakestra`'s cluster orchestrator for an apples-to-apples comparison. Then, we compare the control messages exchanged at the worker level. We record the network messages using the `strace`. Since K3s is a derivative of Kubernetes, they both use similar control communication. Both `Oakestra` and K3s workers send periodic updates to the master and receive control commands in return. However, the number of control messages ingress/egress on K3s far exceeds `Oakestra` (≈ 2×). These results help explain the observations presented in fig. 7. Moreover, a larger amount of control traffic also influences the time needed to synchronize the infrastructure and perform a deployment. Specifically, the master of "kube" based frameworks sends frequent messages to the workers, including specification of the pods to be attached to the workers, liveness checks, etc., requiring

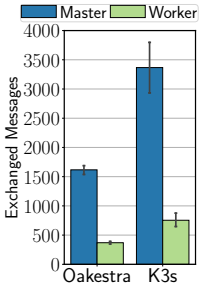


Figure 11: Control message overhead.

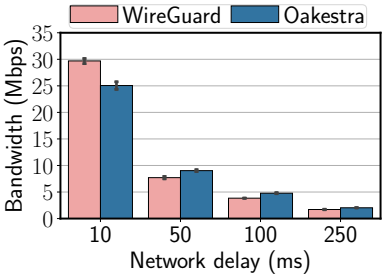


Figure 12: Oakestra vs. WireGuard tunneling overhead.

information/acknowledgment in return. The control communication of Oakestra, on the other hand, is simplified to send periodical aggregated service information from the worker over MQTT messages and keep a minimal footprint. It must be considered that Oakestra still does not provide the guarantees of a production-ready platform. Increasing the monitoring capacity and the update frequency will result in an increase of control traffic. We, therefore, plan on improving the control message communication channel in the future by dynamically tuning the update frequency depending on network conditions.

5.6 Video Analytics Application

We deploy the four microservices composing the video analytics pipeline described earlier on four *S* VMs in the HPC testbed. The provisioned resources do not support GPU acceleration and are single-core machines; therefore, the resulting FPS output is expected to be low. This setup is supposed to stress the platforms into executing this relatively heavy workload on extremely constrained resources. Both K8s and MicroK8s could not support the application since their orchestration components consumed most of the hardware capacity. As a result, we compare application performance over Oakestra, K3s, and without orchestration (native), with native acting as baseline (see fig. 9). Oakestra and K3s exhibit similar performance for object tracking, taking $\approx$ 300-400 ms. However, due to its minimal footprint Oakestra significantly outperforms K3s for supporting the more resource-demanding object detection service, achieving results closer to the baseline. Overall, application performance over Oakestra exceeded K3s by almost 10%. We omit our HET testbed results since they performed similarly to HPC.

6 Conclusion

We presented Oakestra, a flexible hierarchical orchestration framework designed for heterogeneous and constrained edge computing infrastructures. With its logical management hierarchy, Oakestra can sustain a high degree of context separation at scale. The delegated service scheduling of Oakestra reduces the task placement complexity and allows the framework to dynamically adjust to infrastructure changes irrespective of the scale. Furthermore, the proposed networking component enables developers to seamlessly and dynamically adjust the balancing policy with minimal overhead while natively being able to cross different networking boundaries. The lightweight implementation of Oakestra allows it to easily manage constrained resources likely to operate as "edge servers". In such contexts Oakestra superseded the performance of popular production frameworks (Kubernetes and its derivatives), achieving $\approx 10\times$ resource usage reduction and 10% application performance improvement.

We plan to add several feature extensions to Oakestra. For example, we aim to dynamically assign the cluster orchestrator role upon failovers via distributed leader election. We also intend to extend the scheduling and networking capabilities of Oakestra with recent research solutions for the edge, e.g. deadline-aware scheduling, multi-level tunneling, etc. To provide better QoS guarantees, we also aim to support and compare more recent application runtimes such as unikernels, demikernels, or Akka. Finally, we also seek to explore the possibility to incorporate Kubernetes deployments as Oakestra's clusters to achieve integration for existing cloud deployments.

Acknowledgments

We would like to thank the anonymous ATC reviewers and the shepherd for their helpful comments and insights during the review process of this paper. We would also like to thank the ATC Artifact Evaluation Committee for their meticulous examination and efforts to reproduce our results. We are also thankful to Hasso Plattner Institute (HPI) Germany for providing us access to their infrastructure which was instrumental for our experiments. This work was partly supported by the Federal Ministry of Education and Research of Germany (BMBF) project 6G-Life (16KISK002) and EU Health and Digital Executive Agency (HADEA) program under Grant Agreement No 101092950 (EDGELESS project).

References

[1] Celery - distributed task queue. https://docs.celeryq.dev/en/stable/.

[2] Containerd - an industry-standard container runtime with an emphasis on simplicity, robustness and portability. https://containerd.io.

[3] Istio - simplify observability, traffic management, security, and policy with the leading service mesh. https://istio.io.

[4] Jwt tokens. https://jwt.io.

[5] Mongodb documentation. https://www.mongodb.com/docs/.

[6] Mosquitto mqtt broker. https://mosquitto.org.

[7] Openapi initiative. https://www.openapis.org.

[8] Qemu - a generic and open source machine emulator and virtualizer. https://www.qemu.org.

[9] Submariner - a tool built to connect overlay networks of different kubernetes clusters. https://github.com/submariner-io/submariner.

[10] Kubernetes components, Mar 2021.

[11] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[12] Maha Aljarah, Mohammad Shurman, and Sharhabeel H Alnabelsi. Cooperative hierarchical based edge-computing approach for resources allocation of distributed mobile and iot applications. *International Journal of Electrical and Computer Engineering (IJECE)*, 10(1):296–307, 2020.

[13] Simon Bäurle and Nitinder Mohan. Comb: A flexible, application-oriented benchmark for edge computing. In *Proceedings of the 5th International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '22, page 19–24, New York, NY, USA, 2022. Association for Computing Machinery.

[14] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. Ekya: Continuous learning of video analytics models on edge compute servers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 119–135, Renton, WA, April 2022. USENIX Association.

[15] Sebastian Böhm and Guido Wirtz. Profiling lightweight container platforms: Microk8s and k3s in comparison to kubernetes. In *ZEUS*, pages 65–73, 2021.

[16] Antonio Brogi, Stefano Forti, Carlos Guerrero, and Isaac Lera. How to place your apps in the fog: State of the art and open challenges. *Software: Practice and Experience*, 50(5):719–740, 2020.

[17] Tatjana Chavdarova, Pierre Baque, Stephane Bouquet, Andrii Maksai, Cijo Jose, Timur Bagautdinov, Louis Lettry, Pascal Fua, Luc Van Gool, and Francois Fleuret. WILDTRACK: A Multi-camera HD Dataset for Dense Unscripted Pedestrian Detection. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5030–5039, Salt Lake City, UT, June 2018. IEEE.

[18] Claudio Cicconetti, Marco Conti, and Andrea Passarella. A decentralized framework for serverless edge computing in the internet of things. *IEEE Transactions on Network and Service Management*, 18(2):2166–2180, 2021.

[19] Cloud Native Computing Foundation (CNCF). Kubeedge. https://github.com/kubeedge/kubeedge, 2022.

[20] Google Coral. Coral edge. https://coral.ai/products, 2022.

[21] Lorenzo Corneo, Maximilian Eder, Nitinder Mohan, Aleksandr Zavodovski, Suzan Bayhan, Walter Wong, Per Gunningberg, Jussi Kangasharju, and Jörg Ott. Surrounded by the clouds: A comprehensive cloud reachability study. In *Proceedings of the Web Conference 2021*, WWW '21, page 295–304, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Lorenzo Corneo, Nitinder Mohan, Aleksandr Zavodovski, Walter Wong, Christian Rohner, Per Gunningberg, and Jussi Kangasharju. (how much) can edge computing change network latency? In *2021 IFIP Networking Conference (IFIP Networking)*, pages 1–9, 2021.

[23] Michael Cote. Why large organizations trust kubernetes. https://tanzu.vmware.com/content/blog/why-large-organizations-trust-kubernetes, 2020.

[24] Vittorio Cozzolino, Leonardo Tonetto, Nitinder Mohan, Aaron Yi Ding, and Jorg Ott. Nimbus: Towards latency-energy efficient task offloading for ar services. *IEEE Transactions on Cloud Computing*, pages 1–1, 2022.

[25] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. *SIGCOMM Comput. Commun. Rev.*, 34(4):15–26, aug 2004.

[26] The Khang Dang, Nitinder Mohan, Lorenzo Corneo, Aleksandr Zavodovski, Jörg Ott, and Jussi Kangasharju. Cloudy with a chance of short rtts: Analyzing cloud connectivity in the internet. In *Proceedings of the 21st ACM Internet Measurement Conference*, IMC '21, page 62–79, New York, NY, USA, 2021. Association for Computing Machinery.

[27] Jason A. Donenfeld. WireGuard. https://www.wireguard.com/, 2022.

[28] Janick Edinger, Martin Breitbach, Niklas Gabrisch, Dominik Schäfer, Christian Becker, and Amr Rizk. Decentralized low-latency task scheduling for ad-hoc computing. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 776–785, 2021.

[29] Cloud Native Computing Foundation. Lightweight kubernetes | k3s. https://k3s.io, 2022.

[30] Eclipse Foundation. Eclipse fog05. https://fog05.io/, 2022.

[31] Eclipse Foundation. Eclipse iofog. https://iofog.org/, 2022.

[32] Linux Foundation. Scheduling and eviction | kubernetes. https://kubernetes.io/docs/concepts/scheduling-eviction/, 2022.

[33] Raspberry Pi Foundation. Raspberry pi 4. https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/, 2022.

[34] The Linux Foundation. Kubernetes, 2022.

[35] The Linux Foundation. Kubernetes cluster federation | kubefed. https://github.com/kubernetes-sigs/kubefed, 2022.

[36] Intel. Intel nuc. https://www.intel.com/content/www/us/en/products/details/nuc/boards.html, 2022.

[37] Andrew Jeffery, Heidi Howard, and Richard Mortier. Rearchitecting kubernetes for the edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '21, page 7–12, New York, NY, USA, 2021. Association for Computing Machinery.

[38] Youngjin Kim, Chiwon Song, Hyuck Han, Hyungsoo Jung, and Sooyong Kang. Collaborative task scheduling for iot-assisted edge computing. *IEEE Access*, 8:216593–216606, 2020.

[39] Daniel King and Adrian Farrel. A Survey of Semantic Internet Routing Techniques. Internet-Draft draft-king-irtf-semantic-routing-survey-03, Internet Engineering Task Force, November 2021.

[40] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: Fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 376–394, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Juan Liu, Yuyi Mao, Jun Zhang, and Khaled B. Letaief. Delay-optimal computation task scheduling for mobile-edge computing systems. In *2016 IEEE International Symposium on Information Theory (ISIT)*, pages 1451–1455, 2016.

[42] Qiang Liu and Tao Han. Virtualedge: Multi-domain resource orchestration and virtualization in cellular edge computing. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1051–1060, 2019.

[43] Canonical Ltd. Lightweight kubernetes | microk8s. https://microk8s.io, 2022.

[44] Anil Madhavapeddy and David J. Scott. Unikernels: The rise of the virtual library operating system. *Commun. ACM*, 2014.

[45] Karim Manaouil and Adrien Lebre. *Kubernetes and the Edge?* PhD thesis, Inria Rennes-Bretagne Atlantique, 2020.

[46] Nitinder Mohan, Lorenzo Corneo, Aleksandr Zavodovski, Suzan Bayhan, Walter Wong, and Jussi Kangasharju. Pruning edge research with latency shears. HotNets '20, page 182–189, New York, NY, USA, 2020. Association for Computing Machinery.

[47] Nitinder Mohan and Jussi Kangasharju. Edge-fog cloud: A distributed cloud for internet of things computations. In *2016 Cloudification of the Internet of Things (CIoT)*, pages 1–6, 2016.

[48] Seyed Hossein Mortazavi, Mohammad Salehe, Carolina Simoes Gomes, Caleb Phillips, and Eyal de Lara. Cloudpath: A multi-tier cloud computing framework. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[49] Shadi A. Noghabi, Landon Cox, Sharad Agarwal, and Ganesh Ananthanarayanan. The emerging landscape of edge computing. *GetMobile: Mobile Comp. and Comm.*, 23(4):11–20, may 2020.

[50] Nvidia. Jetson agx xavier. https://www.nvidia.com/en-us/autonomous-machines/jetson-agx-xavier/, 2022.

[51] Oakestra. Oakestra - discord. https://discord.gg/7F8EhYCJDf, 2023.

[52] Oakestra. Oakestra - github. https://github.com/oakestra, 2023.

[53] Oakestra. Oakestra artifacts - experiments. https://github.com/oakestra/USENIX-ATC23-Oakestra-Artifacts/tree/main/Experiments, 2023.

[54] Oakestra. Oakestra artifacts - main. https://github.com/oakestra/USENIX-ATC23-Oakestra-Artifacts, 2023.

[55] Oakestra. Oakestra artifacts - network. https://github.com/oakestra/USENIX-ATC23-Oakestra-net-Artifacts, 2023.

[56] Oakestra. Oakestra wiki. https://www.oakestra.io/docs/, 2023.

[57] M. Krochmal S. Cheshire. Rfc 6762 - multicast dns. 2013.

[58] Hooman Peiro Sajjad, Ken Danniswara, Ahmad Al-Shishtawy, and Vladimir Vlassov. Spanedge: Towards unifying stream processing over central and near-the-edge data centers. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 168–178, 2016.

[59] Farah Aït Salaht, Frédéric Desprez, and Adrien Lebre. An overview of service placement problem in fog and edge computing. *ACM Comput. Surv.*, 53(3), jun 2020.

[60] Enrique Saurez, Harshit Gupta, Alexandros Daglis, and Umakishore Ramachandran. Oneedge: An efficient control plane for geo-distributed infrastructures. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 182–196, New York, NY, USA, 2021. Association for Computing Machinery.

[61] W. Shi and S. Dustdar. The promise of edge computing. *Computer*, 49(5):78–81, 2016.

[62] Tanya Shreedhar, Sanjit K. Kaul, and Roy D. Yates. An age control transport protocol for delivering fresh updates in the internet-of-things. In *2019 IEEE 20th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*, pages 1–7, 2019.

[63] Manoel C Silva Filho, Claudio C Monteiro, Pedro RM Inácio, and Mário M Freire. Approaches for optimizing virtual machine placement and migration in cloud environments: A survey. *Journal of Parallel and Distributed Computing*, 2018.

[64] SuperMicro. Outdoor edge systems. https://www.supermicro.com/en/products/outdoor-edge, 2022.

[65] Dianomic Systems. Foglamp – simplifying iiot data management from sensors to clouds. https://dianomic.com/platform/foglamp/, 2022.

[66] Li Tianze, Wu Muqing, Zhao Min, and Liao Wenxing. An overhead-optimizing task scheduling strategy for ad-hoc based mobile edge computing. *IEEE Access*, 5:5609–5622, 2017.

[67] Shanhe Yi, Zijiang Hao, Qingyang Zhang, Quan Zhang, Weisong Shi, and Qun Li. Lavea: Latency-aware video analytics on edge computing platform. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, New York, NY, USA, 2017. ACM.

[68] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P. Jue. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture*, 98:289–330, 2019.

[69] Wuyang Zhang, Sugang Li, Luyang Liu, Zhenhua Jia, Yanyong Zhang, and Dipankar Raychaudhuri. Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1270–1278, 2019.

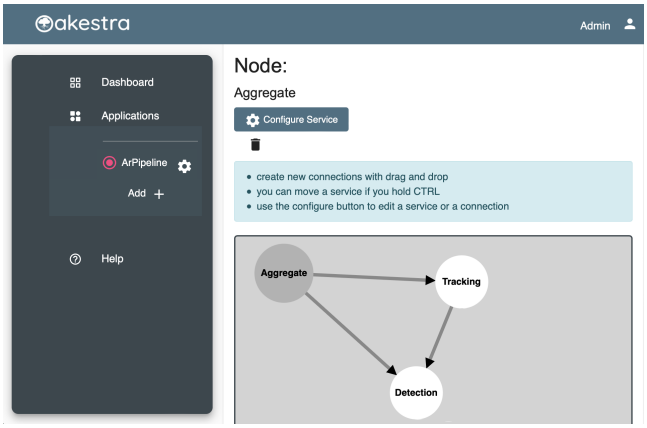
A `Oakestra` Frontend

Oakestra also provides a front-end application that can be used by the developers to create applications and describe the services composing them graphically. A developer can generate the SLA of each microservice through a form. Then, the services connections can be specified using the connection graph (fig. 13a). This graph allows developers to interconnect the services and specify their latency requirements. Once the connection graph is complete, the developer can ask the infrastructure to schedule the desired application. Via the interface, the developer can check each service instance's status and position (fig. 13b); they can also scale the services up, down, or terminate them.

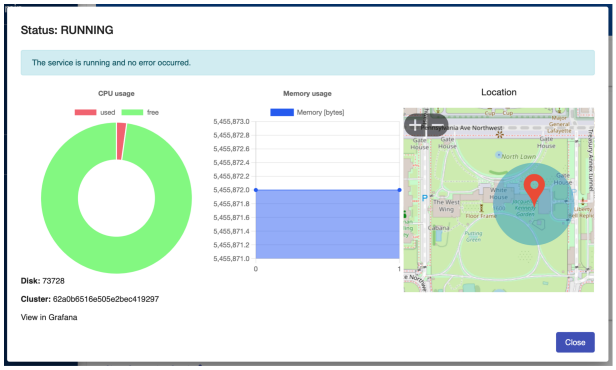
B Artifact Appendix

Abstract

`Oakestra` is an open-source project with all components publicly available on GitHub at https://github.com/oakestra. To ensure the reproducibility of the experiments



(a) Service connection graph



(b) Deployment monitoring interface

Figure 13: `Oakestra` web-based front-end application

conducted within this paper, we fork our project repositories at [54] and [55]. Additionally, we also provide a comprehensive `README`, which includes a `get-started` guide, that can be used for recreating our setup, experiments and familiarizing with the `Oakestra` platform [53].

Scope

The proposed artifacts represents a snapshot of the project that aligns with the paper. The proposed artifacts enable replicating the performance results of `Oakestra` as shown in section §5. However, if the reader is planning to use the latest version of the platform and utilize `Oakestra`'s latest functionalities, we recommend exploring the official website and repository instead.

Contents

Figure 14 shows how the components introduced in §3 can be related to the github repositories. Specifically, the source code, the release binaries and container images are split into the repositories as follows:

- `oakestra/oakestra` [54]: This repository contains the Root & Cluster orchestrators folders, as well as the Node Engine source code for the worker node. Inside the root orchestrator folder, the folders `system-manager-python/` and `cloud_scheduler/` contain the *System Manager* and the *Cloud Scheduler* source code, respectively. Similarly, the *Cluster Orchestrator* folder includes the source of the `cluster-manager/` and the `cluster-scheduler/`. Finally, `go-node-engine/` contains the implementation of the *Node Engine*.

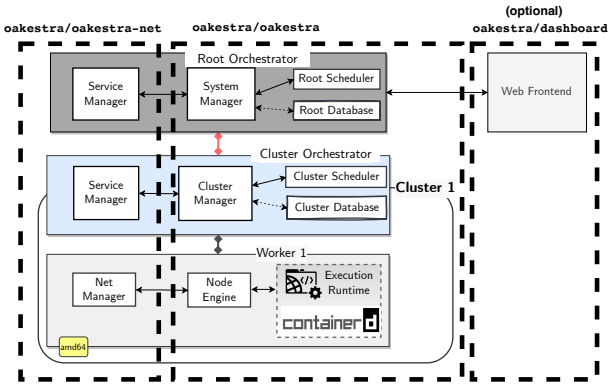


Figure 14: Summary of how the components are split across the repositories

- `oakestra/oakestra-net` [55]: This repository contains the Root, Cluster, and Worker *network* components. Note that the networking stack is not mandatory in `Oakestra`. Without `oakestra-net`, the developer can deploy applications on the infrastructure, but the applications will not be able able to carry out network-related tasks. Since our experiments utilize network-capable applications, our experiment's `README` provides detailed instructions regarding the installation of these components.

- `experiments` [53]: To replicate the experiments, we provide an `Experiments/` folder that includes the setup instructions to create your first `Oakestra` infrastructure and a set of scripts to automate the results collection procedure once the infrastructure is set up as shown in Table 1.

Requirements

We recommend the following minimum requirements for each `Oakestra` component.

1. *Root Orchestrator*: 2GB of RAM, 2 Core CPU, ARM64 or AMD64 architecture, 10GB disk, docker compose installed. Tested OS: Ubuntu 20.20, Windows 10, MacOS Monterey.

Folder	Description
`Test 1`	Deployment overhead calculation (figs. 5 and 7)
`Test 2`	Network overhead measurements (fig. 8)
`Test 3`	Bandwidth measurements (fig. 12)
`Test 4`	Control message measurements (fig. 11)
`Test 5`	Scalability stress test experiments (fig. 6)
`Test 6`	AR pipeline experiments (fig. 9)

Table 1: Experiments test folders

2. *Cluster Orchestrator*: 2GB of RAM, 2 Core CPU, ARM64 or AMD64 architecture, 5GB disk. Tested OS: Ubuntu 20.20, Windows 10, MacOS Monterey.

3. *Worker Node*: Linux Machine, 1 Core CPU, ARM64 or AMD64 architecture, 2GB disk, `iptables` utility.

Beyond the Paper

The AE repository only contains the performance evaluation of `Oakestra`. In addition to this appendix and the repo, the `Oakestra` project provides extensive documentation [56] on how to use `Oakestra` in a variety of infrastructure configurations and applications. In addition, interested researchers are welcome to join the community via GitHub [52] or Discord [51].