Flocky: Decentralized Intent-based Edge Orchestration using Open Application Model

Tom Goethals[1], Merlijn Sebrechts[1],Mays Al-Naday[2],Filip De Turck[1],Bruno Volckaert[1]

***Abstract*—Continuum computing has emerged as a paradigm to improve various aspects of service orchestration by offloading computation from the cloud to the network edge. However, edge orchestration poses two significant challenges compared to cloud computing. On one hand, cloud software scheduling algorithms make suboptimal decisions when applied to the network edge, as edge devices and networks are more hetereogeneous than cloud data centers, and orchestration requires different parameters. On the other hand, most orchestration platforms assume highly centralized cloud data centers, with each server running many easily migrated software instances, whereas edge devices have limited hardware capabilities and migration of tasks between devices is significantly slower than in the cloud. As a result, there is a need for a decentralized orchestration platform that allows scheduling algorithms to take into account a wide variety of device properties and deployment requirements in placement decisions. This article presents Flocky, a decentralized device discovery and service orchestration framework based on Open Application Model (OAM), to address this gap. The architecture of Flocky is elaborated, showing how OAM enables flexible intent modeling in the edge, and combined with a Gossip-like algorithm allows individual edge devices to discover devices in their neighborhoods, map their capabilities, and optimally deploy parts of applications to individual nodes. Evaluation shows Flocky to be highly scalable and mainly dependent on local node density, with nodes discovering over 97% of their viable neighbours on average within two discovery rounds, while using 84% less memory than a centralized orchestrator such as Kubernetes.**

***Index Terms*—edge computing, container networking, pod networking, ebpf, edge kubernetes**

I. Introduction

In recent years, continuum computing has emerged as a paradigm to improve various aspects of service orchestration, such as user experience, green energy use, and hardware resource optimization, by offloading certain computational tasks from the cloud to the geographically widespread edge [1] or vice-versa. Simultaneously, the number of devices available for edge computing has increased exponentially, in environments ranging from industrial sites [2] to city infrastructure and homes [3]. To help developers take advantage of this new paradigm, there is a need for robust orchestration platforms that abstract away the complexities of this infrastructure and provide an easy to use API towards developers and operators. While there are a number of robust and mature orchestration platforms designed for cloud computing, such as Kubernetes, they are not well-adapted to the distributed and heterogeneous nature of the network edge [4].

The first challenge is that cloud software scheduling algorithms make suboptimal decisions when applied to the network edge. This is because cloud scheduling algorithms ignore a great number of network and device properties that are important for optimal placement of workloads in edge networks. For example, the algorithms do not take into account the distributed nature of the network edge and the variety of network conditions between its constituent devices. Unlike cloud networks, fog and edge networks are highly volatile and organic [5], with constantly changing network conditions and topologies. Not only can devices join or leave a topology at any moment, but the physical location of edge nodes such as vehicles and mobile phones can also change rapidly. Furthermore, device metadata is also an important factor to consider in edge workload deployment because it covers a range of functional and non-functional parameters from hardware resources and dedicated computing hardware, to green energy availability and various security features such as encryption, hardened runtimes, etc. Cloud orchestrators such as Kubernetes generally focus on detailed modeling of workload operation, while being mostly agnostic of node properties by default, apart from labels and basic hardware load. Although intents have been proposed as an interesting solution to model the heterogeneous properties of the edge, there has yet to surface a clear standardized way to describe these intents and discover properties at runtime.

The second challenge is that most orchestration platforms assume highly centralized cloud data centers, where each server can run many service or function instances, migrating them seamlessly between machines with little to no service interruption. However, edge hardware is generally resource-limited, and services are more difficult to migrate due to latencies and having to find suitable alternative nodes [6]. As a corollary, scaling in the cloud is very geographically constrained, while the nature of the edge is more suited to scaling across many devices over a wide geographical area, placing service instances close to where they are required to minimize latency. Service scheduling is generally handled by a single, centralized scheduler instance in the cloud; if not, functionally independent clusters are created and federated at a higher level to avoid scaling issues [7]. However, the network edge contains orders of magnitude more devices than data centers, connected at far lower bandwidths. Combined with the volatility of the edge, this makes gathering all the required data for scheduling decisions in a single location unfeasible; a fully decentralized orchestration solution is more suitable.

[1] Ghent University - imec, IDLab,Gent, Belgium, Corresponding author: tom.goethals@UGent.be

[2] University of Essex, Colchester, UK

Despite this, even edge-focussed Kubernetes distributions such as KubeEdge and KubeFed are centralized.

These two challenges combined result in a need for a decentralized orchestration platform that allows scheduling algorithms to take into account a wide variety of both workload, device, and network properties in placement decisions. More specifically, this orchestration platform should allow each node to gather metadata about its neighborhood, match this metadata with a standardized description of workload intents, and make orchestration decisions based on the intersection of the two. Although several studies propose decentralized deployment algorithms for edge computing [8], [9], that work does not address the need for a unifying orchestration framework that enables these algorithms to discover the required information and make informed decisions. While the Open Application Model (OAM)[1] is an interesting contender as a standardized model to describe the properties and requirements of applications and their components, no previous work has investigated how to use it to model workloads, devices and users in the computing continuum. Furthermore, such a unified model could be used as a translational layer between the heterogeneous capabilities of the network edge and high-level user intents to enable intent-driven edge orchestration.

To combine the concepts of decentralization and fine-grained metadata into a framework to enable intent-based orchestration, this article poses the following research questions:

- **RQ1:** How can various sources of intents, e.g. user, workload, and hardware properties, be modeled using OAM?
- **RQ2:** Can a decentralized architecture be devised so that each node gathers metadata about its neighborhood, and makes its own orchestration decisions?
- **RQ3:** How effective is such an architecture, and is the performance overhead acceptable w.r.t. low-resource hardware?

To address these research questions, this article presents Flocky[2], a decentralized OAM-based orchestration framework with a modular and pluggable architecture to allow relevant software (e.g. container engines, VPN software) to advertise node intents and capabilities.

The rest of this article is organized as follows: Section II presents existing research related to decentralized orchestration and workload runtimes, while Section III introduces high level architectural aspects of Flocky, whose theoretical performance is discussed in Section IV. In Section V, the evaluation setup, scenarios and methodology are detailed, while the results are presented in Section VI and discussed in Section VII, which also discusses ideas for future work. Finally, Section VIII draws high level conclusions from the article.

[1]Open Application Model - An open model for defining cloud native apps - https://oam.dev/

[2]https://github.com/togoetha/flocky

II. Related Work

Orchestrators such as Kubernetes, or alternatives derived from it such as K3s[3] or KubeEdge[4], are often deployed for software orchestration in the edge. While Kubernetes and K3s are capable of running on some edge nodes, they are fairly resource intensive, especially control plane nodes [10]. KubeEdge, on the other hand, has specialized components for edge orchestration which communicate with its Kubernetes-based cloud component, making it more suitable for low-resource devices. However, these alternatives are all centralized, leaving orchestration to one or few tightly coupled control plane nodes, thereby forming a bottleneck for large volatile service topologies in the edge. Different ecosystems, such as ioFog [11], are generally based on the same concepts of centralized orchestration despite a more edge-oriented design. Several studies [8], [9] consider the scale of edge computing by constructing and evaluating decentralized deployment algorithms; however these are often focused on specific deployment parameters themselves (e.g. green energy and efficiency [12], latency [13], [14], Quality of Experience [15]) instead of an actual orchestration framework. The solution presented in this article focuses on the latter, while providing the flexibility to implement scheduling algorithms that focus on parameter choices.

Gossip algorithms are often used for decentralized data dissemination, for example [16]. Related alternatives, such as SoSwirly [17], are highly scalable due to a configurable maximum communication range, although it reduces orchestration decisions to a single parameter for efficiency reasons. Flocky, on the other hand, aims to build a multi-tier discovery platform, with plain decentralised network discovery at the bottom layer, and more complex orchestration layers building on top of it, each layer operating on fewer nodes but with more metadata per node to make complex intent-based orchestration decisions.

Most orchestration frameworks are highly container-centric [18]. Various plugins focus on Kubernetes to enable different types of runtimes, for example WebAssembly (WASM) [19], OSv unikernels [20], [21], or Kata Containers [22]. Some alternatives use the flexibility of the containerd shim to implement it for different runtimes such as SpinKube, which leverages WASM [23]. Others, such as Feather, attempt to provide a generic solution for multi-runtime support [24]. The presented framework aims to be both runtime- and workload engine (e.g. kubelet, cloud API) agnostic, leaving the actual choice of how to set up a specific component in its runtime up to the workload engine.

Significant progress has been made in the field of intent-driven network [25] and service management [26], with the aim of using high-level user intents [27], often assisted by Natural Language Processing (NLP), to automate the minutiae of network infrastructure and service orchestration. The OAM-based framework in this manuscript aims to provide a bridge

[3]K3s - The certified Kubernetes distribution built for IoT & Edge computing - https://k3s.io/

[4]KubeEdge - Kubernetes Native Edge Computing Framework - https://kubeedge.io/

between high-level intents and various low-level device- and networking capabilities for edge service orchestration, allowing relevant software to plug in and declare its capabilities in a uniform but extensible structure which can be leveraged to manage service deployments from user intents. Due to the proliferation of Machine Learning (ML) workloads in the cloud-edge continuum, specific frameworks such as KubeFlow[5] have been developed to support ML pipelines on top of orchestration frameworks. Although the proposed framework is not aimed solely at ML orchestration, and does not currently leverage ML models, it does aim to provide the modeling capability to enable detection of ML-oriented traits, and to extend orchestration to ML-specific workflows.

Finally, container images and operation are standardized to a significant degree, through efforts such as the Open Container Initiative (OCI) [28]. Additionally, thanks to its wide support and alternative platforms derived from it, Kubernetes API support is almost a de facto choice for data structures and workflows related to container (or pod) orchestration itself. However, these standards and APIs all focus on low-level orchestration aspects related to containers or pods themselves, with little attention for application-level requirements and node-level capabilities. The Open Application Model (OAM) [29], on the other hand, bridges the gap by focusing on Traits for the definition of specific restrictions or features that should be imposed on deployments (i.e. node capabilities, user intents), and a decomposition of Applications into Components and implementation-specific ComponentDefinitions, leaving the actual workload definition entirely open (e.g. Kubernetes deployment, cloud API objects).

III. Flocky Architecture

This section elaborates the architecture of Flocky, shown at the level of services and components in Fig. 1. The name Flocky is derived from operating at a higher level than Feather, i.e. orchestration, and birds of a Feather Flock together. Flocky allows any device to gather metadata about its environment, and use it to deploy an OAM Application consisting of multiple Components to several target devices, depending on the requirements of each Component and target capabilities. Flocky itself is entirely decentralized, as indicated by the distribution of identical Flocky services among different nodes. Interactions between services are based on an extension of OAM; specific modifications made to OAM are detailed in Section III-A. Flocky itself is comprised of:

- The Discovery service, which is responsible for discovering new nearby nodes and determining whether they are within a certain range.
- The Metadata Repository and Repository service, which are used to extract useful orchestration intent metadata from discovered nodes. Intents in this context can be interpreted as enablers that allow services to be deployed with certain technical intents (e.g. security features, acceleration) or functional intents (e.g. service availability, sidecars, dependencies).
- Swirly and Deployment services, which are respectively the client and server portions of Application deployment; the Swirly service makes orchestration decisions while the Deployment service interacts with Feather to fulfill any deployment requests it receives.

These services are further explained in Sections III-B through III-C.

A. Open Application Model

Apart from a few endpoint adapters, Flocky is orchestrator agnostic and makes use of an extended version of OAM. Technically, this allows translating deployments to any platform including Kubernetes, although Feather [24] is chosen for Flocky due to its low resource requirements and mixed-runtime capabilities. The extensions to OAM are shown in Fig. 2, marked in light green. Importantly, OAM separates object definitions from references, e.g. Components refering to ComponentDefinitions, Traits refering to TraitDefinitions. This is mainly to allow platform operators to configure objects and functionality, while users need only refer to them. The basis of OAM is the Application, which primarily consists of metadata and several Components, i.e. individual (micro-)services. For the purpose of Flocky, Applications are the analog of Kubernetes pods, while Components are the equivalent of containers; the latter are declared in Application deployments similar to containers in Kubernetes Deployment manifests, and contain a reference to a ComponentDefinition, along with Traits and Scopes:

- A ComponentDefinition contains a schema (i.e. analog of a Kubernetes container spec) and all required details for its deployment, including a reference to its WorkloadType. The latter is an abstract representation of functionality or behavior; an official example is the Server[6] WorkloadType for long-running tasks.
- Traits refer to TraitDefinitions, which are intended to apply specific features or restrictions to Components, comparable to metadata labels and Operators in Kubernetes.
- Scopes refer to ScopeDefinitions, and are primarily used to logically group Components for access to shared resources, similar to how Volumes are defined in Kubernetes deployments.

The extensions to OAM are used by Flocky internally and not exposed through orchestrator endpoints:

- The ComponentDefinition Schematic is the Flocky implementation of an unstructured field in the OAM standard, comprising a Kubernetes "Container" description and parameter data. Importantly, Flocky is workload runtime-agnostic; the Kubernetes "v1.Container" struct is used for convenience, and the current implementation also allows for OSv unikernels. The "Schematic" contains a reference to "BaseComponent" which allows several ComponentDefinitions to implement the same base component under the same reference. Application Components should use this name rather than a specific ComponentDefinition

[5]KubeFlow - https://www.kubeflow.org/

[6]https://github.com/oam-dev/spec/blob/master/core/workloads/server.md

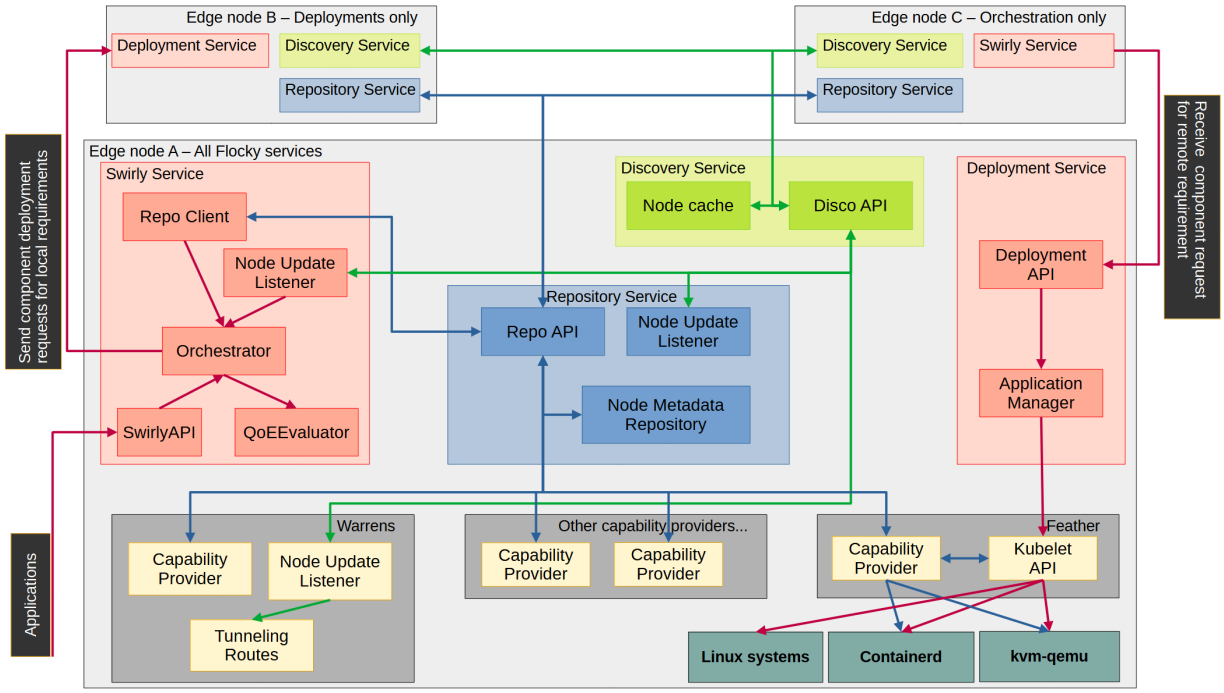


Fig. 1: Architecture overview of Flocky, showing its constituent services and their main components, along with any interactions with external frameworks, runtimes, and Linux systems. The decentralized nature of Flocky is illustrated by the presence of some services on Remote node x/y.

name, which allows Flocky to select the optimal one based on other requirements (e.g. Traits).

- WorkloadType is reinterpreted to serve as an indication of which runtime a ComponentDefinition requires. Basic workload types for OSv unikernels and Docker containers are implemented, while the unikernel workload type is linked to a SecureRuntime Trait, which nodes can advertise if they are capable of running unikernels.
- Several Traits are defined that are useful to describe edge node capabilities and workload requirements: GreenEnergy, SecureRuntime, SoftDistanceLimit, Attestation, and NetworkEncryption. Some traits apply to WorkloadType selection (e.g. a unikernel workload uses SecureRuntime), while others apply to nodes (e.g. limiting QoE impact with SoftDistanceLimit, require a secure node with Attestation) or node properties (e.g. NetworkEncryption, GreenEnergy).
- NodeSummary and NodeCapabilities respectively are used to build the Metadata Repository; these structures allow the Repository service to request the (available) hardware resources, Traits, supported WorkloadTypes and running (Sub)Applications of another node.
- Applications consist of Components, however, several Components of the same Application may be simultaneously deployed on a single node if it fulfills all their requirements. Additionally, Flocky requires a data structure to communicate orchestration decisions and Application level information to deployment nodes. To break down Applications into smaller deployable units while allowing Component grouping, SubApplication is used to deploy subsets of an Application. A SubApplication contains orchestration information such as specific ComponentDefinitions to be deployed for each component, and any Scopes that should be applied to its specific collection of Components.

As such, this Section answers **RQ1** by showing how OAM can be extended for, and applied to, node metadata and intent-based orchestration.

B. *Discovery Service*

The Discovery service is responsible for discovering nearby nodes running the Flocky framework through a REST API hosted by each node, and is based on earlier work in SoSwirly [17]. The Discovery API mainly uses two operations; a ping operation which periodically checks the existence of a remote node and its network latency, and an operation which requests the list of nodes known by a remote node. Using these operations, the Discovery service maintains a cache of nodes within a configurable maximum network latency, by recursively contacting nodes and requesting their node lists; the algorithm uses a depth-first approach, and stops exploring a specific path when response latency is too high, continuing with the next eligible node. Lists of visited and out-of-reach nodes are kept during each iteration, minimizing network traffic and processing overhead, and optionally only nodes running a Deployment service are actually stored in the node cache. Network latency is chosen as a basic metric at this level as the Discovery service is only interested in exploring the actual network topology; advanced metrics are reserved for higher level metadata and decisions. However, like SoSwirly

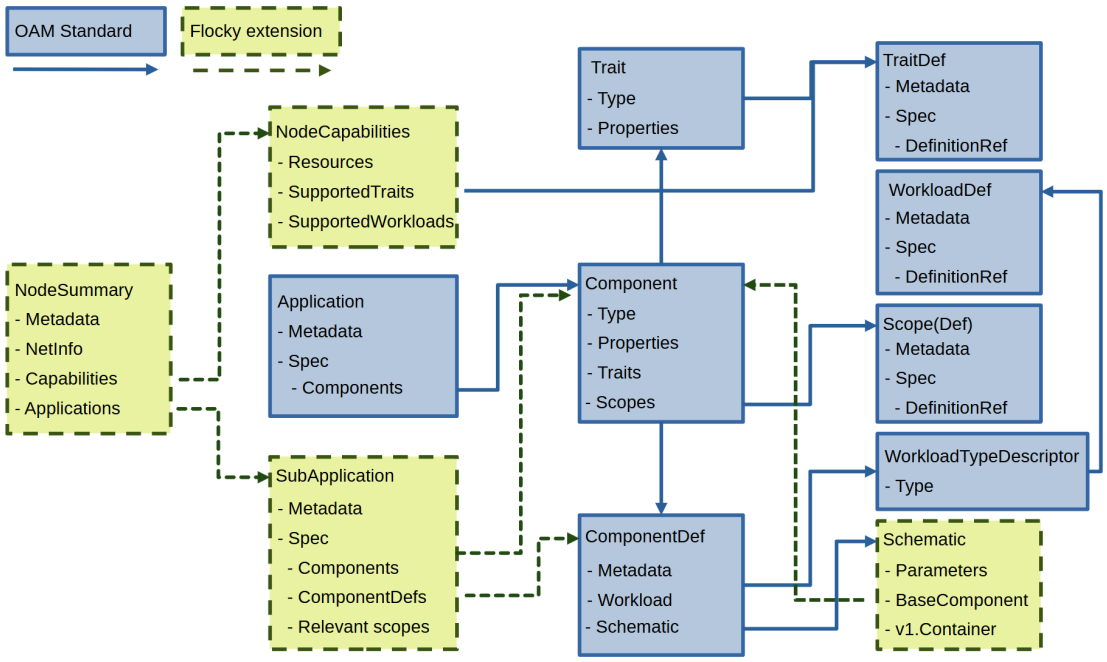


Fig. 2: High level overview of the Open Application Model, and modifications made to enable metadata gathering and deployments in Flocky. Note the reference from the Schematic BaseComponent to Component, indicating an ”implementation of” relation to enable multiple implementations of a Component.

the algorithm evaluates local node density, and expands its search range if it does not find sufficient nodes within the preconfigured range. The discovery algorithm is elaborated in Algorithm 1.

Convergence depends on the initial configuration of a node and maximum discovery latency. Consider a node A, newly deployed in neighborhood α, the set of nodes within a specific latency of A. At least one known node is required when the algorithm starts; if this initial node B is in α, the algorithm will discover its entire neighborhood with as little as a single discovery call, confirming all of α within seconds depending on individual node latencies. However, if B is, for example, an order of magnitude of latency further removed from α, it will take A a considerable time to backtrack to its own neighborhood; this may take in the order of minutes. Only once A finds another member of α can the rest of that subset be informed of its existence. A number of edge cases are discussed in SoSwirly [17], and convergence over time for Flocky is evaluated in Section VI-D. Finally, the Discovery API also provides operations which allow subscribers to receive periodic updates on node topology changes.

C. Metadata Repository

The Metadata Repository is maintained by the Repository Service, which subscribes to the Discovery service for node updates. The repository is not updated on each node update; rather, the service integrates all updates and only periodically contacts known nodes for additional metadata through the Repository API.

One part of this metadata is provided by remote nodes as a NodeSummary (see Fig. 2), which contains node metadata, network interface information, capabilities and any running applications deployed through Flocky. Every node maintains its own NodeSummary through the Repository API, which allows Capability Providers to register with it, as illustrated at the bottom of Fig. 1. Registered Capability Providers are periodically queried for any NodeCapabilities and NodeApplications they provide, representing partial content of the “Capabilities” and “Applications” fields of the NodeSummary. The Repository service itself is responsible for merging all provider capabilities and applications; as indicated, providers may include a container engine i.e. Feather, which provides hardware resources, Flocky-deployed Applications, and supported WorkloadTypes based on detected runtimes. Other providers may be container network-based, e.g. Warrens [30] or a VPN, enabling Traits such as NetworkEncryption. Other Traits may be similarly provided, for example through vendor-specific adapters which detect the presence of green energy, or an attestation agent.

Apart from a NodeSummary, remote nodes are also queried for any ComponentDefinitions in their repository, which are then merged with the local store, allowing new ComponentDefinitions to propagate throughout an entire topology from introduction to just a single node.

Finally, the Repository API also provides a number of functions to subscribe to Metadata Repository updates, and to query the repository directly for ComponentDefinitions fitting specific Traits.

D. Swirly & Deployment Services

The orchestration aspect of Flocky is split into two services: the Swirly and Deployment services. The Swirly service subscribes to both the Discovery and Repository services, receiving updates on newly discovered nodes and metadata

```
function Discover(N_known, N_new) is
    N_check = N_known ∪ N_new
    N_ignore = ∅
    if N_check ≠ ∅ then
        d_min = call Ping(N_check[0])
    inReach = false
    while N_check ≠ ∅ do
        n_c = N_check[0]
        Remove(N_check, n_c)
        d[n_c] = call Ping(n_c)
        d_max = AdjustedDistance(Length(N_known))
        if d[n_c] ≤ d_max) then
            add(N_known, n_c)
            N_new = call GetKnownNodes(n_c)
            MergeNodes(N_check, N_new, N_ignore)
            inReach = true
            d_min = min(d_min, d[n_c])
        else
            Add(N_ignore, n_c)
            if d_min ≥ d_max then
                N_new = call GetKnownNodes(n_c)
                MergeNodes(N_check, N_new, N_ignore)
            if Length(N_known) ≤ 1 or d[c_n] ≤ d_min
             then
                d_min = d[c_n]
            else
                Remove(N_known, c_n)
            end
        end
    end
    if inReach then
        N_remove = ∅
        for n ∈ N_known do
            if
             d[n] >AdjustedDistance(Length(N_known))
             then
                Add(N_remove,n)
        end
        N_known = N_known ∩ N_remove
end
function AdjustedDistance(n) is
    ρ_local = ρ_assumed/n
    return d_max·max(1, √(ρ_local))
end
function MergeNodes(N_check, N_new, N_ignore) is
    N_check = N_check ∪ (N_new ∩ N_ignore ∩ N_known)
end
```

Algorithm 1: Node discovery algorithm used in the Flocky Discovery service. N_{new} is the list of nodes discovered through other means since the last update, e.g. being pinged by an unknown node.

changes. When software on the local nodes requires the deployment of an OAM Application through the Swirly API, the application is split into its Components, and ComponentDefinitions - fulfilling the required Traits for each Component - are requested from the Repository API.

For each Component, the orchestrator matches known nodes with the WorkloadTypes and resource requirements of suitable ComponentDefinitions, along with Component Traits, resulting in a list of candidate nodes. Note that some Traits may be declared as "required", while others are not, in which case optional Traits are potentially ignored depending on the QoE Evaluator. The first step of the matching algorithm considers nodes on which it already has a suitable ComponentDefinition (and thus a service instance) deployed; if found, these are contacted through the Deployment API to determine whether they can support another client. If no existing deployment (or available node) is found, the algorithm iterates the remaining known nodes to determine which is currently capable of deploying a suitable ComponentDefinition based on resources and metadata.

In any case, all elegible nodes for a specific Component are subsequently ranked by a QoE Evaluator based on hardware resources, Traits, WorkloadTypes, latency, and other relevant metadata. Current implementations include the "Legacy" evaluator, which simply ranks by latency, and the "Scored" evaluator, which makes extensive use of metadata using a static calculation. Other implementations may include online learning evaluators, matching Components and nodes based on elusive user preferences which are difficult to capture in a static model. Domain-specific implementations can also be created; ML workloads, especially if (online) learning is required rather than inference only, have fundamentally different priorities than, for example, 5G/6G Network Function Virtualization (NFV) workloads. These may be processed by different QoE Evaluators, prioritizing relevant Traits for each use case. By combining the flexibility of Trait (i.e. capability) providers, Trait implementation logic, and the freedom of custom Evaluators to customize Traits effects, Flocky provides potential support for a host of functional and non-functional intents, user-driven or otherwise.

After ranking, each node is contacted in turn through its Deployment API, i.e. the most desirable ones first, to determine whether it can currently deploy the required Component(Definition). If the list is exhausted before the ComponentDefinition is deployed, the entire Application deployment fails and returns an error.

Importantly, due to the dependency of Components on Traits to ensure specific behavior or properties, a single node (i.e. Swirly service) may actually be requested to deploy several instances of the same Component with different non-compatible Traits e.g. a database sidecar for a standard application, and a highly secured version of that same sidecar for a critical application which requires node attestation. In all cases, the algorithm will first attempt to reuse suitable deployments, even if they operate under Traits which are not strictly required. Only if no existing Component instance with the required Traits is found will another one be deployed on a remote node. Conversely, when removing an Application deployment,

its Application ID is removed from all ComponentDefinitions it requires; the Swirly service only unregisters itself from a specific ComponentDefinition on a remote node when no Component of a local Application still uses it. Similarly, the Deployment API keeps a counter of clients for ComponentDefinitions, and only removes them once the client counter hits 0 (or a specifically defined minimum).

Finally, any deployed applications are constantly monitored, and their Components may be migrated to other devices if any Traits or other requirements are violated after initial deployment. Depending on available Traits, this may include resource/hardware availability, specific security features (through CapabilityProviders), or more complex features (e.g. data availability).

Note that technically, Scopes and SubApplications allow the grouped deployment of several interdependent ComponentDefinitions on the same remote node, however this is not currently implemented.

Considering the comparable process of service composition, the Flocky services map to the various stages as described below:

- **Definition** consists of OAM manifests (Applications), received through the Swirly service API.
- **Scheduling** & **Construction** are intertwined, mainly through the Metadata repository and Swirly services. Due to the combination of Traits, decentralization and multiple component implementations, some aspects of Construction are performed first by selecting ComponentDefinitions that match the requirements. Scheduling is then performed, searching for the optimal nodes to deploy these ComponentDefinitions (and ideally, these are already deployed on an eligible node). As this happens iteratively per Component in the Application, there is no clear distinction between the Scheduling and Construction phases, although a rollback operation is possible.
- **Execution** happens when the Swirly service has confirmed all Components can be suitably deployed; the constructed service composition is confirmed for Execution through the Deployment service, otherwise the previous operation is rolled back to remove any prepared ComponentDefinitions from remote nodes.
- **Evolution** is mainly performed by the Swirly service, keeping track of Component requirements for all deployed Applications; whenever a ComponentImplementation strays from the requirements defined by a Component, the Scheduling and Construction stages are (partially) re-executed to find a new optimal composition.

In conclusion, the Flocky architecture shows how OAM can be integrated into a flexible, modular, decentralized orchestrator, answering **RQ2**.

IV. Theoretical Performance

This section explores the computational complexity for the basic components of Flocky, as well as application deployment. Table I explains the symbols used, while Table II summarizes performance, which is discussed in detail in the rest of this section.

TABLE I: Symbols used in computational complexity.

Symbol	Definition
n	Total number of nodes in topology
c	Number of components in application
ρ_n	Density of nodes w.r.t. maximum discovery distance
ρ_f	Density of nodes with suitable properties for component
$\rho(r,\theta)$	Density of nodes at a specific point
r_d	Maximum discovery distance (generic unit)

TABLE II: Summary of computational complexity.

	Best	Average	Worst
Discovery algorithm	-	$O(r_d^2\rho_n)$	$O(n)$
Service repository	-	$O(r_d^2\rho_n)$	$O(r_d^2\rho_n)$
Application deployment	$O(c)$	$O(c/(1-\frac{\rho_f}{\rho_n}))$	$O(cr_d^2\rho_n)$

A. Discovery Service

Discovery service performance is similar to that of SoSwirly [17]. However, instead of edge nodes and fog nodes, Flocky simplifies the model and considers all nodes for exploration; those without a Deployment service are optionally discarded after discovery. As the Discovery service has a maximum discovery range, average complexity depends only on the local node density within that range, i.e. $O(r_d^2\rho_n)$. However, depending on the choice of "range" metric (latency by default) and its setting, the maximum discovery range may be over $1/4$ of the topology size, leading to a worst case performance of $O(n)$ where the discovery service may encounter every node. To elaborate: consider a node perfectly centered within the topology; if the discovery range is around $1/4$ the topology size, it may discover nodes halfway between itself and the topology border, and examine their known nodes for potential exploration, including any nodes up to the topology border. While the number of discovered nodes depends on connectivity and node density ρ within the topology as shown in Eq. 1, for a uniform or locally high node density combined with sufficient connectivity, the Discovery service will discover and process upwards of 78.5% of all nodes in the topology during every round if $r > 1/4$.

$$n = \int_{\theta=0}^{2\pi}\int_{r=0}^{1/2} r\rho(r,\theta)drd\theta \tag{1}$$

B. Metadata Repository

To build the metadata repository, the Repository service periodically contacts nodes within range of the Discovery service. The Discovery service API only provides relevant nodes within range; as such the performance of the Repository service is $O(r_d^2\rho_n)$ in every case, although concrete performance may vary with the amount of metadata (total number of known components, trait system complexity, etc) in the topology.

C. Swirly & Deployment Services

The complexity of the Swirly service depends mainly on the number of nodes within range and the number of components to deploy in any application. Best case, every component can be requested from the first node attempted, resulting in $O(c)$.

Worst case, the algorithm must try requesting each component from every discovered node, leading to the same performance as the Repository service. The average case depends on the quotient of suitable nodes (i.e. free resources, required traits) to discovered nodes, filling the gap from $O(c)$ to $O(cr_d^2\rho_n)$. Note that in each case, the QoE Evaluator may itself have a complexity of $O(cr_d^2\rho_n)$ depending on implementation, however it is assumed that the latencies and operations on remote nodes involved in deployment are the main contributor to complexity and performance.

V. Evaluation

To verify its theoretical performance and examine its baseline requirements, a Golang implementation of Flocky is evaluated using various scenarios. This section details the evaluation hardware setup, the metrics targeted by each evaluation scenario, and methodological details. The code for all services, dummy implementations, and custom evaluation tools is made available on GitHub[7].

A. Evaluation Testbed

Most evaluation scenarios are performed on a five node setup on the IDLab Virtual Wall[8], consisting of consumer grade pcgen6 nodes with an Intel i5 9400 processor, 16GiB of mounted storage, and 32GiB of DDR4 memory. Each node runs a fully updated Ubuntu 20.04 image. The ARM evaluation is performed on an Odroid HC2 device with a Cortex-A7 ARMv7l processor, 32GiB microSD storage, and 2GiB of LPDDR3 memory. For the baseline benchmark, Kubernetes 1.31 is used.

B. Scenarios

To cover all necessary aspects of scalable, decentralized edge orchestration, the following evaluation scenarios are used:

- A **Functional Evaluation** uses the five node setup shown in Fig. 3 to verify the functional aspects OAM in combination with Flocky to enable decentralized intent-based orchestration. Each node has one or two Traits ranging from green energy to security aspects, and is given a number of Component Definitions at boot time. Applications are deployed from nodes 1, 2 and 5 requiring both node Traits and Component Definitions that they must first discover using the Discovery and Repository services (e.g. node 1 requires component A.2 to be deployed on node 2, while node 2 requires A.1 and C.1 to be deployed on either node 4 or node 5).
- The **Baseline Benchmark** sets a baseline for CPU use and memory consumption of each individual service in Flocky, comparing them to the resource requirements of the components of an empty, idle Kubernetes cluster (i.e. single control plane node) using Flannel as a Container Network plugin. Container runtimes are excluded for both Flocky and Kubernetes, while functionality that is not supported in Flocky is also excluded from Kubernetes (e.g. CoreDNS instances). For Flocky, two nodes are set up to ensure all services on the measured node are active and running, while Kubernetes runs only a single control plane node.
- The **ARM Comparison** validates the ability of Flocky to run on an (older) ARM platform, and provides baseline performance comparing x64 to ARM. Similar to the Baseline Benchmark, two nodes are set up to ensure fully active services on the measured nodes.
- A **Scalability Evaluation** simulates the performance of a large number of nodes by running several instances of the Discovery and Repository services on the same machine simultaneously, measuring memory impact, CPU use and network traffic. Node distribution is derived from the evaluation scenario presented in SoSwirly [17], which generates random node topologies based on density maps; in this case, the density map is derived from the population density of Brussels and surrounding towns. Edge topologies are generated ranging from 25 to 150 nodes, and two iterations of each of these topologies are run; specifically for maximum discovery distances of 10ms and 20ms latency, with the maximum end-to-end latency of the topology defined at 73.5ms, showing the effect of both node count and node density on performance. This scenario also measures the impact of metadata corpus size on network traffic; by default, each node starts with 5 component definitions, randomly generated from a pool of 2 implementations for each of 50 components, in order to simulate a sizeable amount of discoverable metadata. This situation represents an "open" computational infrastructure where users can create and deploy applications at will. One step down are topologies with 25 components, each with 2 implementations, representing a focused ecosystem (e.g., a collaborative version of Home Assistant[9]) with a range of verified services and plug-ins. The smallest metadata corpus is 5 components with a single implementation each, representing a vendor with a restricted number of proprietary components for specific use cases (e.g. Smart Home HVAC systems).
- **Deployment Latency** is analyzed from debug timing and logging during the Functional Evaluation, running the node 1 deployment five times in a freshly started cluster. This scenario provides a useful insight into the highest contributing factors for deployment latency, despite the limited number of iterations. Note that while actual container deployment for each Component is stubbed for all evaluations, it is asynchronous and thus would not affect the evaluation and its results.

C. Methodology

For the Baseline benchmark and ARM comparison, CPU and memory metrics are scraped from the "top" command in batch mode, running the command for any relevant processes once per second for 100 consecutive seconds. Specifically, the "Discovery and Repository" (running as a single process under

[7]https://github.com/togoetha/flocky

[8]https://idlab.ugent.be/resources/virtual-wall

[9]Home Assistant - Awaken your home - https://www.home-assistant.io/

"discovery"), "Swirly", and "Deployment" services are evaluated. Flocky does not contain the equivalent of a Container Networking plugin in Kubernetes, instead values from Feather container networking are reused from a previous evaluation of the current version [24]. For Kubernetes, all kube* processes are evaluated, along with etcd and Flannel. Components which have no equivalent implementation in Feather are excluded to ensure a fair comparison.

In the scalability evaluation, process IDs for all simulated instances are tracked and used to gather CPU and memory metrics from **/proc/%PID%/stat** directly. Network throughput is not tracked per process; instead, average throughput is calculated from **/proc/net/dev**. Due to the low CPU load of Flocky services, the gathered cumulative CPU statistics increase relatively slowly. In order to maximize accuracy, this scenario is run for 20 samplings, with 10 seconds between each sampling. Additional metadata is gathered from the simulated nodes themselves, allowing the calculation of discovery "Accuracy", i.e. the number of available links within maximum latency which nodes have discovered between themselves, compared to the theoretical maximum. The use of a single physical machine to simulate all nodes and services limits this scenario to around 150 nodes, at which point significant timeouts and latencies start to occur. Despite extensive debugging, the cause was not found but likely involves threading or socket issues due to OS resource exhaustion. However, future evaluations using Flocky as an orchestration platform using multiple physical nodes are planned, which may reveal more information.

For all evaluations, CPU scaling is disabled to ensure a proper comparison. To accurately measure the impact of the Flocky framework itself, especially in the Scalability scenario, Feather and Warrens are replaced with dummy implementations that provide the basic functionality for Flocky to operate. Finally, deployment latency was gathered from the "time" command to estimate REST request latency, combined with debug logging to calculate latency for each step of the orchestration process.

VI. Results

A. Functional Evaluation

The observed outcome of the functional evaluation is shown in Fig. 4, in which all components are deployed according to their requirements. Technically, this outcome depends on the order of deployment. As nodes never attempt to deploy components to themselves, node 5 may throw an error if node 2 is allowed to deploy its application first and chooses node 4 to deploy both components; in that case, the CPU resources of node 4 are 50% utilized for A.1 and C.1 for node 2, leaving too little resources for the deployment of B.1 for node 5. However, this is an extreme case in a very limited evaluation scenario; the only viable option is when both node 4 and node 5 run one of the necessary B.1 components, as these require too many resources to run on the same node.

B. Baseline Benchmark

The results of the Baseline benchmark are shown in Fig. 5 for memory use. All Flocky components use between 13-16MiB of memory, except Feather which uses ≈54MiB idle. Kubernetes components use between 48MiB and 262MiB of memory each, with the exception of Flannel at 37MiB. As it is impossible to directly compare individual components in terms of functionality, totals are included for both Flocky and Kubernetes, showing that Flocky consumes around 108MiB of memory including the multi-runtime functionality of Feather, or ≈84% less memory than Kubernetes.

Fig. 6 shows the respective results for CPU use, indicating that while almost every Kubernetes component is performing some work even in an idle state, only the discovery service of Flocky consistently uses a small amount of processing power to discover new nodes. While the idle CPU load is technically divided between the Discovery and Repository services, the Discovery service is by far the largest load in this state. In total, Flocky requires between 86% and 100% (median) less processing power than Kubernetes in idle state. However, as these numbers are measured in % of a single CPU core, both Flocky and Kubernetes have at worst a minimal effect on total CPU power when idle.

C. ARM Performance

Fig. 7 shows the memory consumption of Flocky components on x64 compared to ARMv7l. While the Discovery (and Repository) service consumes around 2MiB less on ARM, the Swirly and Deployment services consistently consume around 3MiB less, or ≈26.5%. However, significantly more processing power is used on ARM compared to x64, as shown in Fig. 8. On ARM, the Discovery service requires an order of magnitude more processing power compared to x64, and up to 16 times more in the maximum case. Even the Deployment and Swirly services, which respectively have no processing load and metadata processing for a single node, are observed to require some processing power in their maximum cases. However, even the maximum combined 18% of a single CPU core for Flocky is a relatively minor load for the 8 core Odroid HC2 device, and newer ARM devices with hard-float capabilities undoubtedly fare much better [31], as they allow hardware processing of floating point operations used in all Flocky algorithms and service operations.

D. Scalability Evaluation

Memory and CPU scalability for an increasing number of nodes is shown in Fig. 9. Solid lines indicate a maximum discovery distance of 10, while dashed lines indicate a maximum distance of 20. As the topology size is constant for each iteration, only node density is affected within each series, and performance generally adheres to the linear predictions in Table II. Between series, there is no expected quadratic increase of CPU use with a doubling of the discovery range; however, the discovery range of 20 is slightly over 1/4 of the topology size of 73.5, which according to Section IV-A results in an upper bound for discovery performance rather than the average case. Memory use, however, is closer to the expected behavior, as not all visited nodes are effectively kept in the node cache; $O(n)$ is thus an upper limit on CPU use, not

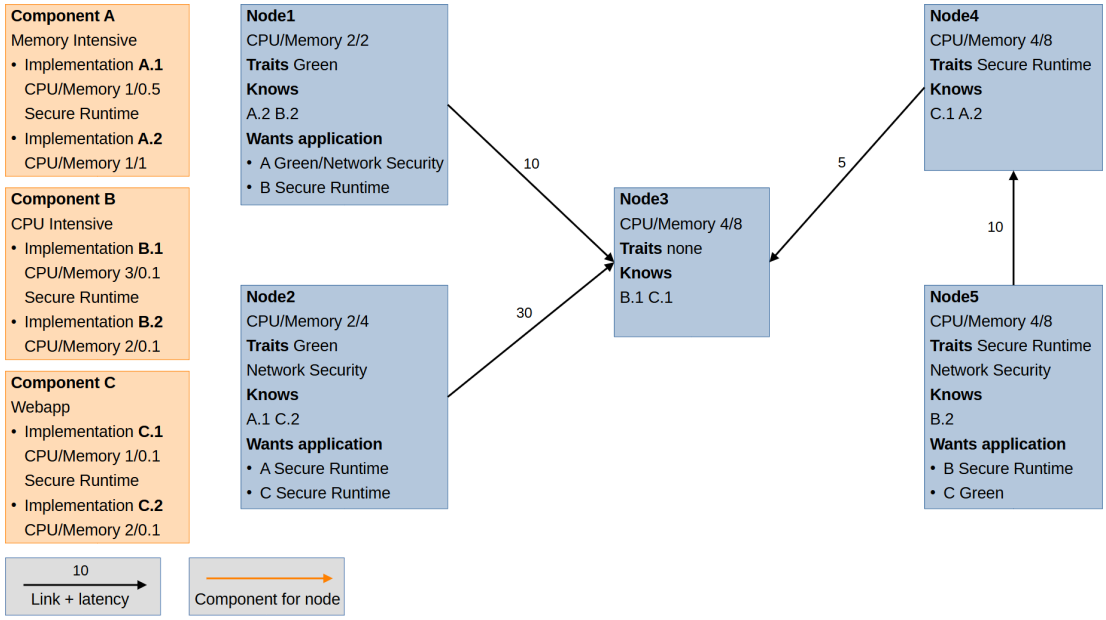


Fig. 3: Testbed setup for the function evaluation of Flocky, including initial node topology (arrows) and latencies. Flocky is deployed on five nodes, with a specific set of known component definitions and traits for each, while nodes 1, 2 and 5 must find suitable candidates for the application components they require.

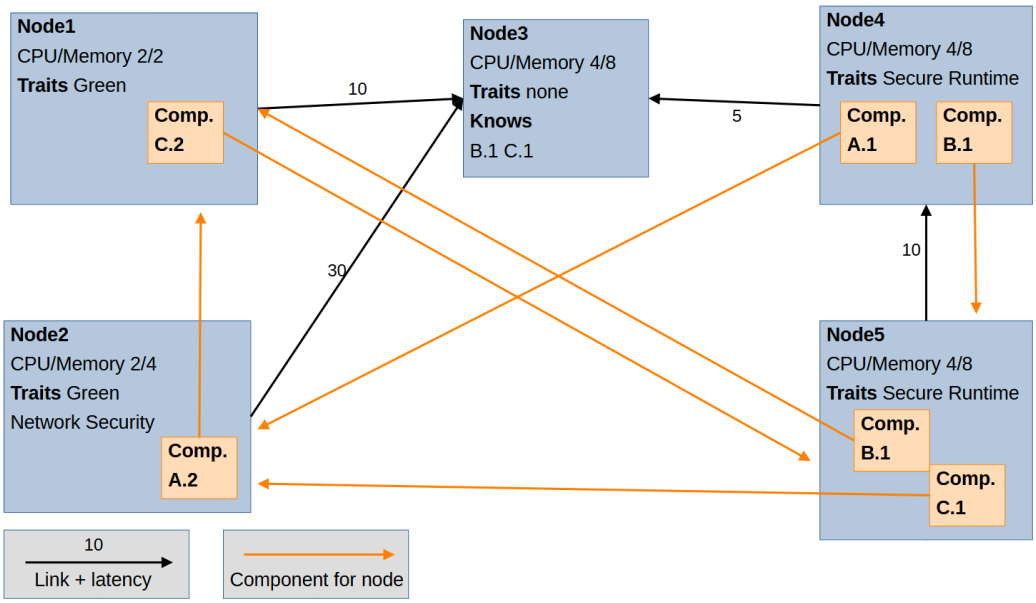


Fig. 4: The observed outcome of the functional evaluation, with all components deployed according to their requirements.

memory consumption, until the discovery range is closer to 50% of the topology size.

The latter is illustrated by Fig. 10, showing a tripling of the number of discovered neighbours when doubling the discovery range. Network traffic, although it roughly follows the same scaling factor of the number of known neighbours w.r.t. discovery distance, rises slightly faster than linear w.r.t. node density. Considering the effect of a large discovery range, there is likely a minute $O(n^2)$ effect from node discovery at play, as nodes will still receive discovery data about the same node multiple times from their neighbours, even if they subsequently ignore it.

In absolute numbers, all metrics are acceptable at higher node densities; nodes can actively communicate with around 60 neighbours on average while using less than 3% of a single CPU core, at the cost of 8MiB of memory compared to the baseline, much of which consists of extensive node metadata. While network traffic is relatively high at 400Kbps for 60 neighbours, the update cycles for the evaluation are set at 3s for discovery and 5s for metadata. Realistically, these intervals could be longer, or an event-driven approach could be implemented which only propagates changes to neighbours instead of the entire node cache/metadata store.

The accuracy of the discovery mechanism is illustrated in

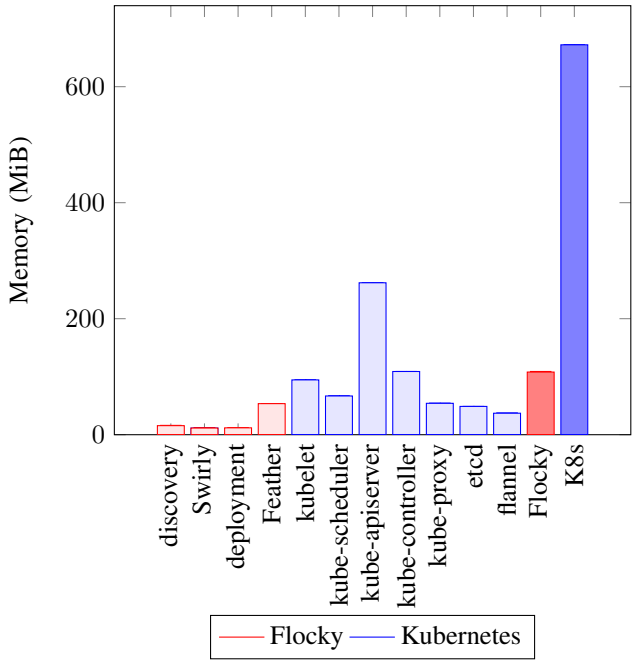


Fig. 5: Memory use of Flocky services compared to relevant Kubernetes components. Totals for both Flocky and Kubernetes (K8s) are included.

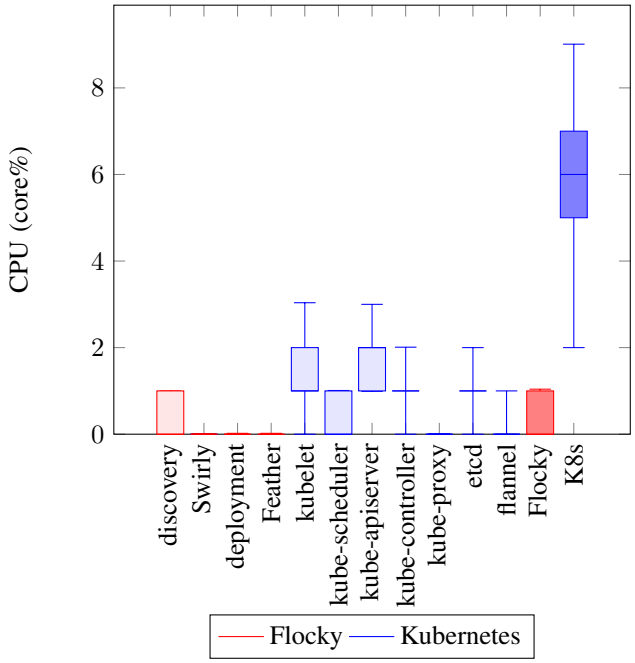


Fig. 6: CPU use of Flocky services compared to relevant Kubernetes components. Totals for both Flocky and Kubernetes (K8s) are included.

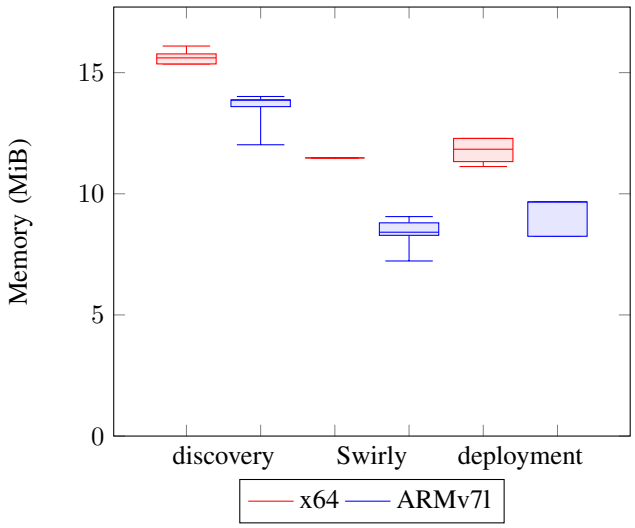


Fig. 7: Memory use of Flocky components on x64 compared to an ARMv7l platform.

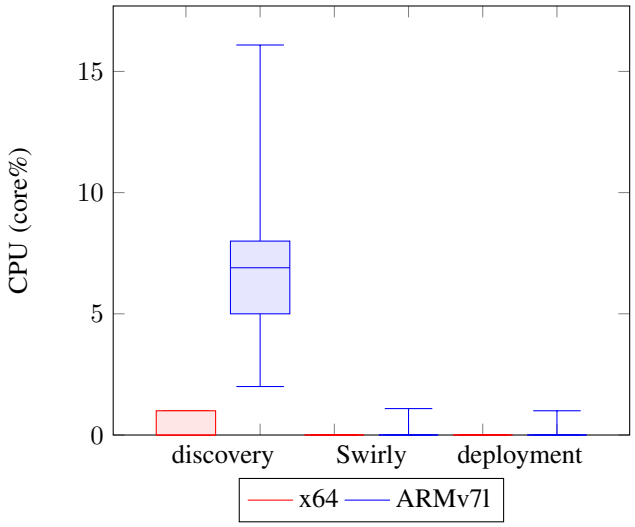


Fig. 8: CPU use of Flocky components on x64 compared to an ARMv7l platform.

Fig. 11 for a maximum discovery distance of 20. Even with only 25 nodes in the entire topology, all nodes manage to discovery almost 96% of the viable links between them. At 75 nodes, this number rises to slightly above 99%, where it stabilizes for higher node densities. The accuracy for a maximum discovery range of 10 is similar, although it only reaches 99% accuracy around 125 nodes. Component discovery, while not charted, is consistently observed to discover all content within a topology for randomly sampled node logs. Importantly, accuracy does not change over time during the evaluation; from the first round of sampling, i.e. 10 seconds or maximum two discovery rounds, accuracy remains at the presented levels, indicating a fast convergence of discovery.

To summarize, Fig. 12 shows the absolute scalability of the results from Figs. 9 and 10, by normalizing to a topology with 25 nodes and a maximum discovery distance of 20. While CPU use scales perfectly with the number of neighbours

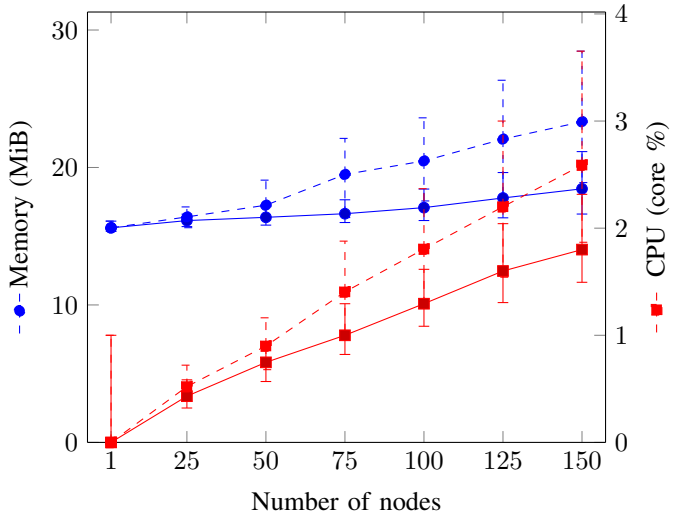


Fig. 9: Scalability of memory consumption and CPU use for an increasing number of nodes in a topology. Solid lines indicate a maximum discovery distance of 10, while dashed lines indicate a maximum distance of 20.

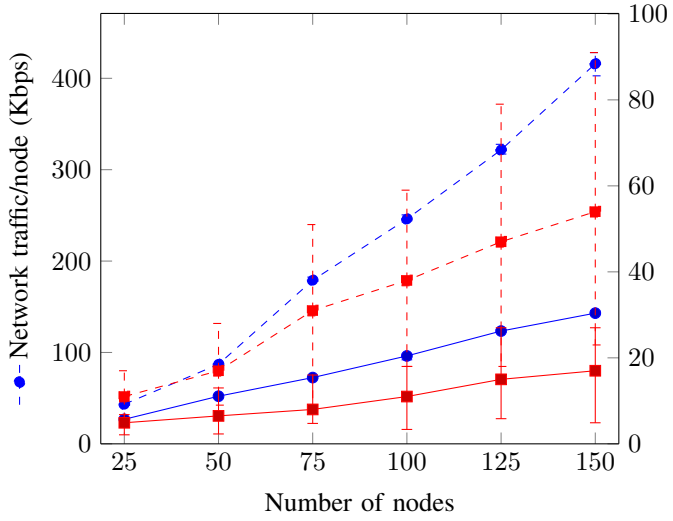


Fig. 10: Scalability of network traffic and number of discovered neighbours with increasing node count in a topology. Solid lines indicate a maximum discovery distance of 10, while dashed lines indicate a maximum distance of 20.

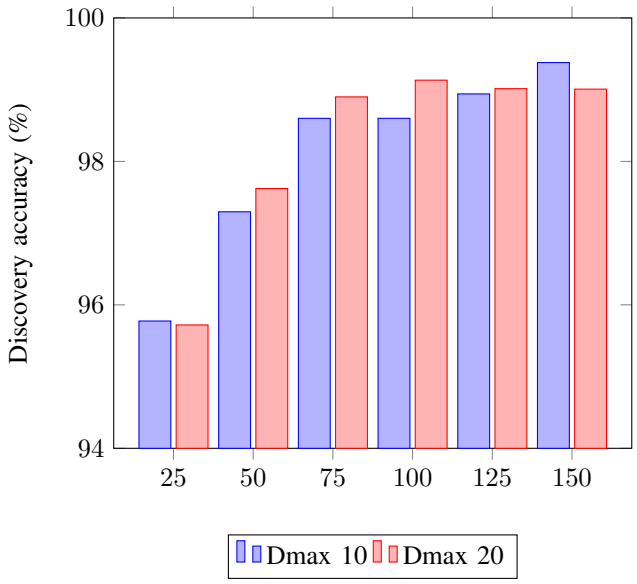


Fig. 11: Discovery algorithm accuracy, as defined by the number of links discovered by all nodes within their maximum range divided by the total of such links in a topology.

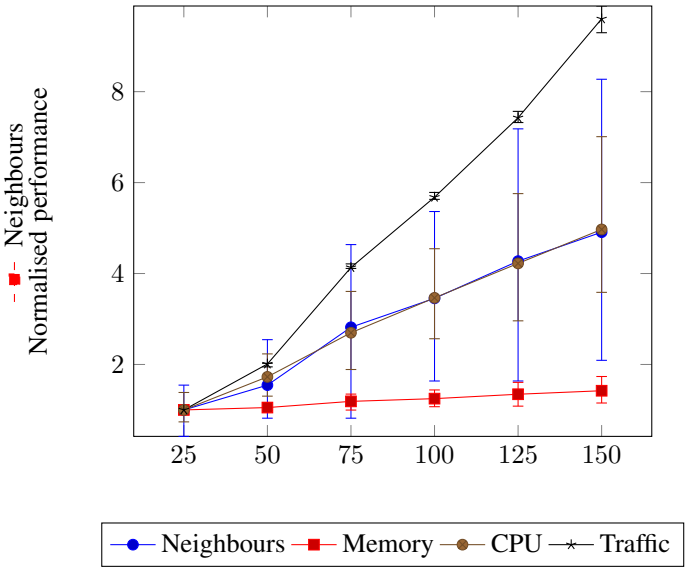


Fig. 12: Relative scalability of all metrics for an increasing number of nodes in a topology, normalized to 25 node topologies.

per node, network throughput suffers a secondary effect as discussed, and rises about twice as fast over the observed range of node densities. Memory scaling, on the other hand, is fairly insignificant compared to the base memory use of the Discovery service.

Fig. 13 shows the effect of the amount of cluster-wide discoverable metadata on network traffic per node. Interestingly, the bulk of traffic appears to consist of node discovery and synchronization calls; there is only an 8% to 18% difference in network traffic between almost no metadata and 100 discoverable components for a maximum discovery distance of 10, and 10% to 20% for a discovery distance of 20. The effect of doubling the discovery distance is an order of magnitude greater than increasing the number of components by 20 times, showing that a good choice for discovery distance far outweighs a growing number of components in a Flocky topology. This is partially achieved by only exchanging component IDs during metadata updates, resulting in around 1300 bytes for 50 components; full OAM manifests, which may be several kilobytes individually, are only requested when a node encounters an unknown component.

Finally, Fig. 14 illustrates the memory use of an entire Flocky topology including ancillary services such as Feather (measured) compared to an idle Kubernetes cluster of the same size, with a single control plane node (at a bare minimum as

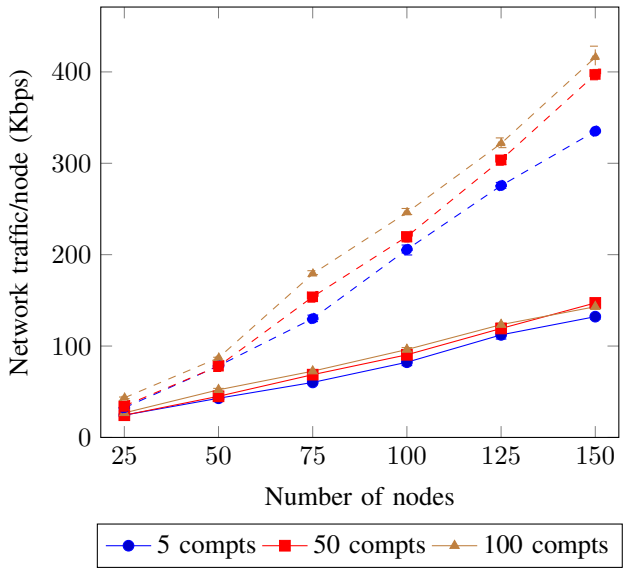


Fig. 13: Scalability of network traffic with increasing metadata corpus in a topology, expressed in discoverable components. Solid lines indicate a maximum discovery distance of 10, while dashed lines indicate a maximum distance of 20.

derived from component memory use in Fig. 5). The results show that, despite a decentralization of orchestration and responsibilities, the edge-designed Flocky results in a lower overall memory use across an entire topology. Furthermore, Flocky discovery distance is shown to have an almost insignificant impact on total memory use compared to the memory consumption of a Kubernetes cluster.

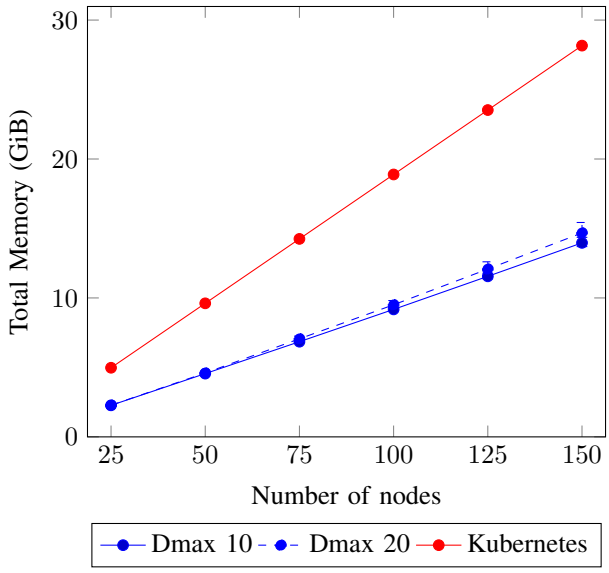


Fig. 14: Cluster-wide memory consumption for an increasing number of nodes in a topology, for different maximum discovery distances, in Flocky compared to an idle Kubernetes cluster.

E. Deployment Latency

A breakdown of all the factors contributing to Application deployment latency is shown in Fig. 15. The entire range of latencies observed is noted within each segment, while timings for the median case are marked below the time axis. Network delays are by far the greatest contributor; of the 21.1ms of the median deployment, 18.7ms are directly attributed to REST calls, i.e. “Request”, “Node availability” and “Async deploy”. The orchestration algorithm itself requires only 0.2ms and 0.4ms, respectively, to find an optimal target for the first and second Components to be deployed. Handling DNS changes after each Component deployment, although not yet fully implemented, takes slightly longer, between 0.8ms and 1ms.

While limited, this scenario indicates that the Metadata Repository and orchestration algorithm are unlikely to form a bottleneck while scaling; rather, resource contention and repeated node availability calls will cause deployment times to balloon, incentivizing both a relatively low maximum discovery range and a balanced topology with at least some available resources on most nodes (i.e. limiting demanding non-Flocky workloads).

VII. Discussion and Future Work

The Baseline benchmark shows that the Flocky services, combined with Feather as a multi-runtime deployment engine, is significantly more memory efficient than Kubernetes; the average memory consumption of around 108MiB should be low enough for many edge devices to utilize Flocky, and depending on node requirements some services may be ignored (e.g. Deployment services for client-only nodes). Evaluation on an ARM platform shows the memory use for each service is significantly lower compared to x64, saving around 8MiB in total. This does not include any performance improvements for Feather on ARM as it could not be properly run; both the Odroid HC2 and HC4 kernel versions (4.9) are too outdated for cgroups v2 and Feather eBPF programs, which require kernel version 5.15. Although the entire stack may be run on a Raspberry Pi 4 or 5, these devices were not available for evaluations.

Concerning scalability, the evaluation results mostly adhere to the computational complexity of Flocky components; CPU use scales linearly with the number of neighbours, and thus local node density. The memory impact of scaling is less significant, as each service already uses around 10MiB for, among others, REST APIs even when idle. There is a small $O(n^2)$ effect at play in network traffic, however, which may be mitigated by limiting the discovery range for nodes as it is still dependent on local node density. Additionally, the discovery timings can be finetuned, as it is unlikely the entire metadata store should be updated every 5 seconds as in the evaluations. Considering that all metadata is eventually passed throughout the entire topology as with Gossip algorithms, the results allow geographically widespread topologies for decentralised orchestration to be formed, containing large numbers of nodes while limiting the performance impact for each individual node. The discovery algorithm is shown to be highly accurate; even with a low number of discoverable nodes in range, all

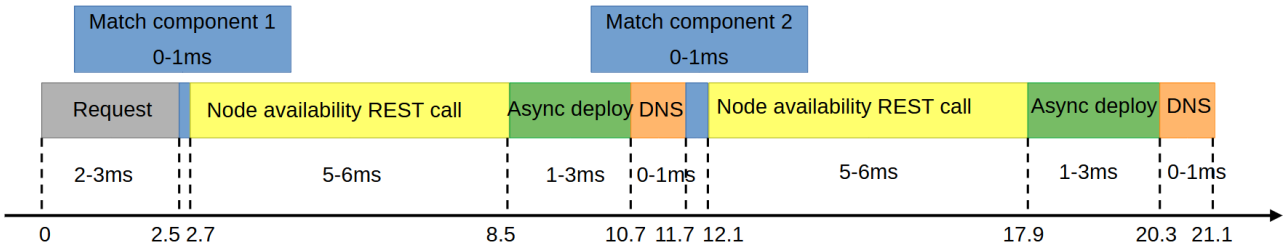


Fig. 15: Overview of contributing factors to Application deployment latency. The full range of latencies observed is noted within each segment, while timings for the median case are marked below the time axis.

nodes combined discover 96% of the valid links between them after just a few discovery rounds, and upwards of 99% of all valid links for medium to high node densities.

Finally, deployment latency is shown to be mostly dependent on the physical latency between nodes; the processing time required to find suitable deployment targets in a small topology is only 2.8% of the total time taken, excluding container downloading and startup. As such, the main contributors to deployment latency in any topology will be network latency and resource availability; high resource contention will cause a node to contact many others to find suitable candidates for each required component.

As a result, **RQ3** is answered, showing that the resource requirements and performance scalling of Flocky are acceptable for many edge devices and topologies.

However, there are several aspects for potential future work. First of all, the QoE Evaluators implemented are rudimentary, and only take into account a static model for QoE calculation. Other versions may consider additional node Traits, enabling more granular preferences for component deployments, or leverage ML to learn optimal service deployments based on user preferences. Additional Traits can be implemented, most importantly (green) energy based Traits to optimize power use, and security Traits for node features such as attestation, trusted computing modules, encrypted payloads, etc. Furthermore, the framework currently only uses a single QoE Evaluator simultaneously; with some modification, and by extending Application metadata, different QoE Evaluators could be dynamically loaded and applied depending on Application (or even Component) type.

To further optimize processing requirements and network throughput, many discovery operations can be converted to event-based rather than poll-based. For example, nodes could notify subscribers whenever their metadata or list of deployed components changes, rather than having those frequently polled. However, additional measures should be taken to avoid divergence between node metadata stores with such an incremental system. The evaluation results indicate that most traffic consists of node discovery and heartbeat type synchronizations; these types of traffic specifically may benefit by moving from a pull-based mechanism to an event-based mechanism, similar to how Kubernetes Node Leases[10] and Watches have been developed to severely reduce synchronization overhead.

While this manuscript focuses on the core design principles and relevant performance metrics of Flocky itself, there are important design considerations and application domains for future work. In terms of security, a VPN-like solution such as Warrens [30] may be implemented as a capability provider to ensure secure communications between nodes, using encryption features up to the level supported by individual pairs of nodes. At the device and deployment level, several Zero-trust tenets [32] may be implemented by integrating a framework such as TrustEdge [33]: a Kubernetes Operator which allows for up-front attestation of edge devices, generating certificates and keypairs, and deploying a verified Feather image while leveraging Trusted Execution Environments (TEEs) if available. Such a framework could be used to deploy verified versions of Flocky services alongside Feather, and because of its reliance on certificates could be decentralized in tandem with Flocky through hierarchical attestation and deployments. This integration would also provide ongoing verification and validation of devices after reboots or OS modifications (updates). In terms of application domains, a capability provider can be built specifically for the detection of AI training and inference capabilities, along with extra trait definitions to leverage those capabilities in components containing ML payloads. This would enable a generic Gossip Learning service to be built using the metadata store, allowing pluggable ML components to run and learn on edge devices with minimal configuration and optimal allocation. Finally, an additional layer allowing abstract user input to be translated into OAM-based manifests is required to form a complete, closed-loop intent-driven system. Ideally, this would be achieved through NLP to improve user-friendliness.

Furthermore, the DNS system to support multiple components of the same type at different locations with different Traits is not yet fully implemented; the current implementation only supports a single instance. While this scenario is somewhat unlikely to occur (i.e. two users on the same edge device requiring two completely different remote versions of the same service), it is important to develop for increased support.

[10]https://kubernetes.io/docs/reference/node/node-status/#heartbeats

Finally, the current implementation returns an error if it does not find a suitable deployment candidate for a component within its own neighborhood. However, unlike the Discovery and Metadata services, where setting a maximum discovery range is important for performance, deployment should not necessarily be limited to directly known neighbours unless required by the QoE Evaluator or Traits (i.e. SoftDistanceLimit Trait). A third stage in the deployment algorithm can be devised so it first covers any known deployments for a component, attempts to deploy it on directly known nodes if none exist, and only upon failure attempts second-hop nodes for deployment, followed by multiple-hop nodes. However, this will significantly impact deployment latency as the deployment algorithm will have to perform its own discovery beyond the maximum discovery range, up to $O(n^2)$. This result can only be avoided by not being too stringent with Trait requirements for each component, and by not supporting too great a variety of Traits within a single topology.

VIII. Conclusion

The network edge and the range of applications within it are constantly expanding, often driven by increasing user expectations. This article states a number of research questions to determine if it is possible to build a fully decentralized, intent-driven orchestration framework using OAM, and presents Flocky as a solution to these questions.

The architecture of Flocky is presented, showing how OAM is extended for use between its various services to detect and discover metadata. Importantly, the framework is designed to be agnostic of both the endpoint workload engine and workload runtimes; any Component may have multiple implementations suitable for different runtimes (e.g. WASM, containerd, OSv). Traits, on the other hand, are leveraged to discover node properties and use them for orchestration purposes when matched with Application deployments. At the network level, the Discovery service is responsible for finding nearby Flocky nodes within a specified maximum latency. This information is used by the Repository service to build a local metadata repository, tracking the hardware availability, Traits, and deployed Components for each discovered node.

The Swirly and Discovery services leverage this metadata repository to find the optimal remote nodes for Application deployments, splitting an Application into its constituent Components and matching its required Traits to known nodes. Different, pluggable methods of calculating QoE allow for additional Traits and priorities in future implementations.

Flocky is evaluated both functionally and non-functionally; the functional evaluation shows that in a purposefully designed topology, several nodes deploy their Applications as expected. Baseline performance is shown to be significantly less resource intensive than centralized alternatives such as Kubernetes, using only 16% of the memory and a fraction of the processing power, and the total requirements are suitable for many classes of edge devices. Performance scaling is shown to be relative to node density and discovery distance, with no theoretical limit on total topology size, although superfluous traffic in the discovery algorithm and the total amount of Components known to the topology may impose soft limits. Discovery itself is shown to be highly accurate, starting at around 96% in sparse topologies and upwards of 99% in medium to dense topologies.

Finally, some topics for future work are discussed, which may significantly improve both the functionality of Flocky, and improve its network traffic scaling in large, dense topologies.

IX. Acknowledgment

The research in this paper has been funded by Flanders Research Foundation Junior Postdoctoral Researcher grant number 1245725N, and the NATWORK Horizon Europe project.

References

[1] S. Dustdar, V. C. Pujol, and P. K. Donta, "On distributed computing continuum systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 4, pp. 4092–4105, 2022.

[2] G. S. S. Chalapathi, V. Chamola, A. Vaish, and R. Buyya, "Industrial internet of things (iiot) applications of edge and fog computing: A review and future directions," *Fog/edge computing for security, privacy, and applications*, pp. 293–325, 2021.

[3] Q. V. Khanh, V.-H. Nguyen, Q. N. Minh, A. D. Van, N. Le Anh, and A. Chehri, "An efficient edge computing management mechanism for sustainable smart cities," *Sustainable Computing: Informatics and Systems*, vol. 38, p. 100867, 2023.

[4] M. Sebrechts, B. Volckaert, F. De Turck, K. Yang, and M. Al-Naday, "Fog native architecture: Intent-based workflows to take cloud native toward the edge," *IEEE Communications Magazine*, vol. 60, no. 8, pp. 44–50, Aug. 2022.

[5] X. Xu, C. Yang, M. Bilal, W. Li, and H. Wang, "Computation offloading for energy and delay trade-offs with traffic flow prediction in edge computing-enabled iov," *IEEE Transactions on Intelligent Transportation Systems*, vol. 24, no. 12, pp. 15 613–15 623, 2022.

[6] E. Bozkaya, "Digital twin-assisted and mobility-aware service migration in mobile edge computing," *Computer Networks*, vol. 231, p. 109798, 2023.

[7] F. Faticanti, D. Santoro, S. Cretti, and D. Siracusa, "An application of kubernetes cluster federation in fog computing," in *2021 24th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. IEEE, 2021, pp. 89–91.

[8] H. Lin, L. Yang, H. Guo, and J. Cao, "Decentralized task offloading in edge computing: an offline-to-online reinforcement learning approach," *IEEE Transactions on Computers*, 2024.

[9] S. Aggarwal, M. Bastopcu, S. Ulukus, T. Başar *et al.*, "Fully decentralized task offloading in multi-access edge computing systems," *arXiv preprint arXiv:2404.02898*, 2024.

[10] M. B. Bahy, N. R. D. Riyanto, M. Z. F. N. Siswantoro, and B. J. Santoso, "Resource utilization comparison of kubeedge, k3s, and nomad for edge computing," in *2023 10th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*. IEEE, 2023, pp. 321–327.

[11] iofog - bring your own edge. [Online]. Available: https://iofog.org/

[12] P. Arroba, R. Buyya, R. Cárdenas, J. L. Risco-Martín, and J. M. Moya, "Sustainable edge computing: Challenges and future directions," *Software: Practice and Experience*, vol. 54, no. 11, pp. 2272–2296, 2024.

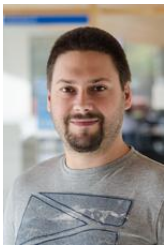
[13] X. Zhang, D. Hou, Z. Xiong, Y. Liu, S. Wang, and Y. Li, "Eallr: Energy-aware low-latency routing data driven model in mobile edge computing," *IEEE Transactions on Consumer Electronics*, 2024.

[14] A. Younis, S. Maheshwari, and D. Pompili, "Energy-latency computation offloading and approximate computing in mobile-edge computing networks," *IEEE Transactions on Network and Service Management*, vol. 21, no. 3, pp. 3401–3415, 2024.

[15] J. Yu, A. Alhilal, T. Zhou, P. Hui, and D. H. Tsang, "Attention-based qoe-aware digital twin empowered edge computing for immersive virtual reality," *IEEE Transactions on Wireless Communications*, 2024.

[16] K. M. Sim, "Agent-based fog computing: Gossiping, reasoning, and bargaining," *IEEE Letters of the Computer Society*, vol. 1, no. 2, pp. 21–24, 2018.

[17] T. Goethals, F. De Turck, and B. Volckaert, "Self-organizing fog support services for responsive edge computing," *Journal of Network and Systems Management*, vol. 29, no. 2, Jan. 2021.
[18] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE cloud computing*, vol. 1, no. 3, pp. 81–84, 2014.
[19] V. Kjorveziroski and S. Filiposka, "Webassembly orchestration in the context of serverless computing," *Journal of Network and Systems Management*, vol. 31, no. 3, p. 62, 2023.
[20] cloudius-systems/osv: Osv, a new operating system for the cloud. [Online]. Available: https://github.com/cloudius-systems/osv
[21] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov, "{OSv—Optimizing} the operating system for virtual machines," in *2014 usenix annual technical conference (usenix atc 14)*, 2014, pp. 61–72.
[22] A. Randazzo and I. Tinnirello, "Kata containers: An emerging architecture for enabling mec services in fast and secure way," in *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. IEEE, 2019, pp. 209–214.
[23] "Hyper-efficient serverless on kubernetes, powered by webassembly." Nov. 2024. [Online]. Available: https://www.spinkube.dev/
[24] T. Goethals, M. De Clercq, M. Sebrechts, F. De Turck, and B. Volckaert, "Feather: Lightweight container alternatives for deploying workloads in the edge," in *Proceedings of the 14th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, 2024, pp. 27–37.
[25] A. Leivadeas and M. Falkner, "A survey on intent-based networking," *IEEE Communications Surveys & Tutorials*, vol. 25, no. 1, pp. 625–655, 2023.
[26] K. Mehmood, K. Kralevska, and D. Palma, "Intent-driven autonomous network and service management in future cellular networks: A structured literature review," *Computer Networks*, vol. 220, p. 109477, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128622005114
[27] A. Clemm, L. Ciavaglia, L. Z. Granville, and J. Tantsura, "Intent-Based Networking - Concepts and Definitions," RFC 9315, Oct. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9315
[28] Open container initiative - an open governance structure for the express purpose of creating open industry standards around container formats and runtimes. [Online]. Available: https://opencontainers.org/
[29] "Open application model - an open model for defining cloud native apps." Feb. 2025. [Online]. Available: https://oam.dev/
[30] T. Goethals, M. Al-Naday, B. Volckaert, and F. De Turck, "Warrens: Decentralized connectionless tunnels for edge container networks," *IEEE Transactions on Network and Service Management*, vol. 21, no. 4, pp. 4282–4296, Aug. 2024.
[31] D. Molloy, *Exploring BeagleBone: Tools and Techniques for Building with Embedded Linux, Chapter 7*. Wiley, Dec. 2018.
[32] V. Stafford, "Zero trust architecture," *NIST special publication*, vol. 800, no. 207, pp. 800–207, 2020.
[33] J. Thijsman, M. Sebrechts, F. De Turck, and B. Volckaert, "Trusting the cloud-native edge: Remotely attested kubernetes workers," in *2024 33rd International Conference on Computer Communications and Networks (ICCCN)*, 2024, pp. 1–6.



Tom Goethals is a senior researcher at Ghent University and imec's IDLab group, teaching distributed data processing, distributed systems management and computer security. His research deals with intelligent, decentralized cloud-edge discovery and orchestration, and optimization of orchestration components for low-resource edge devices. He has worked on several national and international research projects and is author or co-author of 20 peer-reviewed papers published in international journals and conference proceedings, having received four awards.

Merlijn Sebrechts is a senior researcher at imec and teaches at Ghent University in Belgium. He leads a number of research tracks focused on software deployment and trust in the cloud and on devices. He is currently serving on the Ubuntu Community Council and is standardizing WebAssembly System Interfaces for IoT devices as part of the W3C and the Bytecode Alliance. He teaches topics such as Distributed Systems Design, Open Source ecosystems and Computer Security. His work has been published in over 20 scientific publications and has received four awards.

Mays Al-Naday is an associate professor and manager of the Network Convergence Laboratory (NCL) in at the University of Essex. Her research focuses on developing novel solutions for cloud service orchestration and their cybersecurity, including data- and AI-as-a-Service. Mays has participated in a number of international (EU) projects and successfully supported the delivery of TRL-7 system trials. Mays has led a number of national and international projects and has (co-)authored over 40 publications in prestigious journals and conferences.

Filip De Turck leads the network and service management research group at Ghent University, Belgium and imec. He (co-) authored over 750 peer reviewed papers and his research interests include design of efficient softwarized network and cloud systems. He is involved in several research projects with industry and academia, served as chair of the IEEE Technical Committee on Network Operations and Management (CNOM), and serves as a steering committee member of the IM, NOMS, CNSM and NetSoft conferences. Prof. Filip De Turck served as Editor-in-Chief of IEEE Transactions on Network and Service Management (TNSM), was named an IEEE Fellow in 2020, and received the IEEE ComSoc Dan Stokesberry Award in 2021.

Bruno Volckaert is professor of advanced software engineering and secure distributed systems at Ghent University and imec's IDLab group. His research deals with reliable and high-performance distributed software for a.o. scalable data ingestion and processing, secure software architectures, and autonomous optimization of cloud/edge-based applications. He has worked on over 80 national and international research projects and is author or co-author of more than 250 peer-reviewed papers published in international journals and conference proceedings.