ClusterSlice: Slicing Resources for Zero-touch Kubernetes-based Experimentation

Lefteris Mamatas (Member, IEEE), Sotiris Skaperas (Member, IEEE), Ilias Sakellariou (Member, IEEE)
Department of Applied Informatics, University of Macedonia, GR-54636 Thessaloniki, Greece
{emamatas, sotskap, iliass}@uom.edu.gr

***Abstract*—ClusterSlice is an open-source solution for automated kubernetes-centered experimentation. It introduces well-designed abstractions that reduce experimentation complexity with improved reliability and reproducibility. Its main capabilities are: (i) automated declarative operation, e.g., through declarative specifications of experimentation slices; (ii) infrastructure-as-a-service (i.e., the utilization of heterogeneous physical and virtual resources), platform-as-a-service (e.g., multiple composable Kubernetes flavors and network plugins), and application-as-a-service (e.g., plug-and-play application features) capabilities; (iii) multi-cluster and multi-domain support, i.e., inter-cluster operation over multiple open testbeds, virtualization systems and domains; and (iv) experimentation automation, e.g., support of automated experiments of various research topics, including network plugin performance assessments and anomaly detection workflows. Here, we provide the basic architectural attributes of ClusterSlice and proof-of-concept results highlighting its above capabilities.**

***Index Terms*—Testbed based experimentation, Kubernetes, Edge Computing, Zero-touch Network Management, Network Slicing.**

I. Introduction

New research endeavours in cloud and networks benefit from open testbed infrastructures that lower the barrier of experimentation complexity, improve reproducibility of experiments, while contribute in the technical accuracy of documented results. An experimenter typically logins in a testbed system (i.e., via a web-site, testbed control application or an API), requests resources with particular configurations (e.g., physical servers, virtual machines or network equipment) and after a while is able to execute experiments. Such infrastructures may also be federated, i.e., supporting allocation of scalable multi-testbed resources over the globe, as well as commonly provide experimentation automation facilities.

However, both testbed hardware and software demand heavy maintenance duties that are sometimes difficult to achieve, especially during periods with research funding shortage. Thus, opening research infrastructures to the wider scientific community is not always feasible, and some testbeds end-up commercializing their offerings, or their software becomes obsolete or their hardware starts to face frequent failures.

On the other hand, Kubernetes is a widely used container orchestration facility with impressive automation, fault-tolerance, scalability, resource optimization capabilities and a highly modular architecture. Kubernetes appears ideal for experimentation automation. Firstly, there is an emerging need to experiment with kubernetes-based systems and containerized workloads, since it is a common way nowadays to deploy real software systems. Secondly, its disruptive management paradigm can enrich experimentation automation through its capabilities and abstractions. Lastly, it is well-maintained, supported and documented from the open-source community for both its basic features and extensions.

Along this direction, we have built ClusterSlice, an open-source solution for automated kubernetes-centered experimentation. ClusterSlice is a powerful solution which is able to convert testbed resources from bare-metal to fully-operatable experimentation slices. It introduces well-designed abstractions that reduce experimentation complexity with improved reliability and reproducibility, as for instance each experimentation node is represented by a container, which deploys and configures the corresponding systems in a parallel fashion and zero-touch manner.

ClusterSlice is cloud-native and fully integrated to Kubernetes, i.e., via custom resource definitions (CRDs) and the operator pattern, and can be fully managed via the `kubectl` tool. Due to this tight integration, it benefits from novel Kubernetes reliability features. It improves reproducibility of experiments, since they can be executed with an one-liner, while supporting a heavy usage of versioning in all components. Finally, it produces debug information and supports a role-based security model, e.g., considers both administrator and experimenter roles.

In summary, ClusterSlice supports:

- *Declarative automated operation*: Experimenters define characteristics of experimentation slices to be deployed at a higher level, in the form of YAML files. This declarative definition includes slice's specification in terms of the cloud resources to be utilized, the Kubernetes configuration, and the application modules to be installed.
- *Infrastructure-as-a-service capabilities*: ClusterSlice supports the utilization of heterogeneous configurable physical and virtual resources, including those utilized from open testbed infrastructures (e.g., CloudLab), as well as VMs allocated in XCP-NG and VirtualBox virtualization systems. It can employ VM snapshots for more rapid replications of experiments.
- *Platform-as-a-service features*: It supports multiple Kubernetes flavors, such as vanilla Kubernetes (k8s), k0s, k3s, and microk8s, as well as network plugins for both

This research was partially funded by the Greek Ministry of Education and Religious Affairs for the project ”Enhancing Research and optimizing University of Macedonia's administrative operation.

Corresponding Author: Sotiris Skaperas (email: *sotskap@uom.edu.gr*).

intra-cluster (e.g., Flannel, Calico, Cilium, Kuberouter, Kube-ovn, etc.) and inter-cluster communication (e.g., Submariner).

- *Application-as-a-service attributes*: An experimentation slice supports the definition of applications to deploy, k8s extensions and modular OS configurations, all in the form of configurable application modules. It is free from any dependencies on external libraries or APIs, since it harvests on the power of *bash* scripting, SSH and Ansible [1], ensuring maximum compatibility with heterogeneous systems.
- *Multi-cluster and multi-domain capabilities*: ClusterSlice can operate across multiple heterogeneous deployment environments through technology-specific infrastructure managers, establishing multi-cluster operation and communication, such as Liqo, OCM, or Submariner. It supports multiple physical and virtual infrastructure controllers (i.e., infrastructure managers in ClusterSlice terminology).
- *Experimentation automation*: Experimentation automation capabilities are not only supported but also actively developed, making the deployment and management of experiments straightforward. For example, we support automated comparable evaluations of network plugins and AD workflows.

The rest of the paper is organised as follows. Section II contrasts ClusterSlice to the related works. Section III provides an overview of the ClusterSlice platform, i.e., details the declarative definition of an experimentation slice and its architecture along with an explanation of its core components. Subsequently, the same section gives an example of a slice deployment workflow. Finally, Sections IV and V present our proof-of-concept results and conclude the paper, respectively.

II. Related Work

Open testbeds are pivotal in enabling real experiments by automating complex implementation, deployment, and configuration tasks. These environments also grant researchers access to valuable and often expensive infrastructures that are otherwise challenging to obtain.

The European SLICES testbed [2] (formerly known as Fed4FIRE+) federates 17 testbeds equipped with diverse technologies, including 5G/6G, SDN, Edge Computing, AI, GPUs, cognitive radio, optical, and more. Similarly, FABRIC [3] (formerly known as GENI), a USA equivalent initiative, enables multiple federations to interoperate, including Emulab, CloudLab, Chameleon, Orbit, and others. PlanetLab [4] adopted a crowd-sourcing model in which individual users or institutes could contribute resources. At its peak, it controlled 1353 nodes across 717 sites in 48 countries. Such solutions focus on the easy provisioning, sharing and federation of the provided infrastructure based on novel testbed control features.

For example, the allocation of physical testbed resources may utilize software capable of deploying or saving hard-disk images, thereby enabling users to switch their experiments on or off, so a time-sharing model for different experimenters can be employed. Similar tools are the OMF [5] or Frisbee. The latter is part of the Emulab software suite [6]. Another approach is to access cloud facilities based on their proprietary APIs, such as VM resources. Network isolation is usually performed based on VLANs or relevant technologies.

GENI has developed the Aggregate Manager (AM) API [7] for testbed federation, which is also utilized by the SLICES testbed. AM operates based on the RSPEC format, which is an XML-based representation of testbed resources. RSPEC is employed by several testbed control tools and APIs, including SLICES' jFed and jFed-cli [8], GENI OMNI [7], and others. The Emulab [6] and CloudLab [9] testbeds have evolved to incorporate a Python API for experimentation control, alongside a web-based GUI (e.g., [10]).

The above testbeds usually deploy particular software in the form of OS or VM images. Testbed control facilities may also support other software deployment automation capabilities, e.g., jFed prepares Ansible environments based on the configuration of an experimentation slice. It may also enable experimentation automation, e.g., for sharing or collecting measurements among/from the nodes.

These testbeds primarily provide infrastructure-as-a-service capabilities, but they support useful tools and APIs that make them re-usable for higher-level experimentation abstractions or testbeds, e.g., one can access the same resources being available for different context-sensitive testbeds. For example, CloudNativeLab [11] enables the creation of a Kubernetes cluster on SLICES testbeds, using a defined resource configuration for the cluster. It offers access to the cluster via personal VPN and Kubernetes configurations, along with the ability to create customized deployments using tools like kubectl or Helm.

ClusterSlice supports multiple infrastructure managers under a common design abstraction, so all above mentioned testbeds could be jointly utilized and federated. Additionally, it includes in its release a software toolkit for new infrastructure managers, i.e., provisions for the support of new physical or virtual infrastructures. An example of the latter is the CloudLab Infrastructure Manager, that allocates CloudLab nodes for ClusterSlice, based on Geni Tools [7]. Furthermore, ClusterSlice also supports both platform-as-a-service capabilities (e.g., composable Kubernetes clusters or multi-cluster deployments) and application-as-a-service capabilities, i.e, defining the experimentation software or Kubernetes extensions to use: all at the level of experimentation slice definition.

An evolution of PlanetLab is EdgeNet [12], [13], which extends vanilla Kubernetes, based on CRDs and operators, with a number of novel capabilities. For example, EdgeNet defines a role-based security model with untrusting tenants who can operate as vendors (i.e., providing their resources to other tenants) or consumers of resources, with each tenant being allocated a specific quota or a slice (i.e., a number of nodes exclusively assigned to them). It also allows users or institutes to contribute nodes to EdgeNet, similarly to PlanetLab, thereby enriching the scale and heterogeneity of EdgeNet deployments. Its selective deployment feature is manifested through a location-based node selection and deployment, allowing for specific geographical regions or GPS coordinates to be considered as criteria for locating nodes.

Lastly, a new feature of EdgeNet is the support of Kubernetes clusters' federation.

ClusterSlice is an extension of vanilla Kubernetes, architectured around the concept of Custom Resources and Operators, like EdgeNet, as being elaborated in the next Section (Sec. III-B), but also provides full node access to experimenters, as well as the capability to declarative define one or multiple composable Kubernetes clusters and the experimentation applications to use.

Other platform- or application-oriented solutions, include Service Layer At The Edge (SLATE) [14] and the Kubernetes-native testbed environment [15]. SLATE facilitates the federated operation of science platforms, allowing sites to delegate service deployment and configuration to designated application administrators. Sites interested in participating in the service establish a SLATE container/virtualization platform, currently based on Kubernetes. The Kubernetes-native testbed environment integrates various microservices-based applications, CI/CD environments, and monitoring tools, having a specific focus on the application aspect.

Additionally, Kubespray [16] is an Ansible-based deployment tool for creating production-ready Kubernetes clusters. It allows for the deployment of customizable clusters, such as selecting the network plugin, across various infrastructures, including AWS, GCE, Azure, OpenStack, vSphere, bare metal, and more. Additionally, it is compatible with multiple Linux distributions. ClusterSlice supports the declarative definition of applications, experimentation software or Kubernetes extensions to deploy, based on Ansible as well.

Crossplane [17] is a cloud-native control plane framework providing the necessary abstractions to provision, compose and consume any kind of infrastructure, e.g, cloud resources. These abstractions are: (i) Packages: these are extensions of Crossplane that introduce new functionality, essentially bundles of Custom Resource Definitions (CRDs) and controllers designed to represent and manage external infrastructure; (ii) Providers: packages that empower Crossplane to provision infrastructure on external services; (iii) Managed Resources: custom resources that serve as representations of infrastructure primitives; and (iv) Composite Resources: these resources combine managed resources into higher-level infrastructure units.

5G-CDN [18] and NECOS [19], [20] are slicing infrastructures capable of allocating resources across various open testbeds by utilizing the required experimentation control and monitoring abstractions. The above solutions inspired ClusterSlice, especially in the design and implementation of its multi-infrastructure operation. Possible ClusterSlice extensions could be to utilize Kubespray for cluster deployment and benefit from the novel Crossplane abstractions, e.g., adopt compatible provider packages.

Concluding, ClusterSlice aims at providing an overarching approach, that is flexible enough to accommodate both a diverse range of resource providing platforms and easy, fast and reproducible deployment of experiments via a common declarative specification, harvesting on the Kubernetes concepts of CRDs and operators. To our knowledge, no other approach offers all these characteristics simultaneously.

III. The ClusterSlice Experimentation Framework

In transforming testbed resources from bare-metal or hypervisor setups into fully-operational Kubernetes slices in an automated manner, ClusterSlice embraces a unique philosophy characterized by:

- Design empowered from the Kubernetes paradigm, incorporating heavy utilization of Custom Resource Definitions and Kubernetes Operators. This approach inherits the reliability, scalability, and resource optimization capabilities of Kubernetes.
- Introduction of innovative automation abstractions, encompassing both node-level and cluster-level automation, i.e., Resource Managers and Slice Operators respectively.
- Avoidance of API or technology-specific software, aside from tools like *bash*, command-line utilities, and SSH through Ansible. This approach preserves the simplicity, portability, and composability inherent to the Unix philosophy.
- Abstractions of infrastructure managers that interface with diverse testbed facilities (e.g., CloudLab) and cloud/virtualization systems (e.g., VirtualBox or XCP-NG). This maximizes compatibility across heterogeneous systems and ensures seamless integration of future systems.

One of the strong features of ClusterSlice is its capability to deploy a diverse range of characteristics for the experimental slice through a declarative definition. This is further discussed in the section that follows.

A. Declarative Definition

ClusterSlice involves three dimensions of X-as-a-service capabilities, i.e., *Infrastructure*, *Platform (or Kubernetes)* and *Application*. The declarative definition consists of parts, each addressing a specific dimension, located in the respective fields of the YAML file, as well as an overarching part on global experimentation slice properties, as defined below:

- *Global properties*, such as the *name* of the slice to be deployed, the *user namespace*, but also more complex aspects such as the VM *deployment strategy* to be used over multiple servers, as well as security details, i.e., *slice admin username* and *password*.
- *Infrastructure configuration* that includes specifications for all three types of nodes supported, i.e., regular non-Kubernetes nodes, master nodes, and worker nodes. These specifications define for example the number of nodes to be deployed (e.g., *workers count*), the OS image (e.g., *workers osimage*), etc.
- *Platform (i.e., Kubernetes) configuration*, that details the respective deployment, such as the Kubernetes type, supporting alternative flavors (e.g., vanilla, k3s, k0s, microk8s), along with parameters such as the *network fabric*, the *versions of Kubernetes components* (e.g., kubectl, kubeadm, containerd and configuration details, e.g., service and network Classless Inter-Domain Routing (CIDR) used in the experimentation slice. Such a declarative specification leads to a platform that offers effortless

```
apiVersion: "swn.uom.gr/v1"
kind: SliceRequest
metadata:
  name: plainslice
  namespace: swn
spec:
  name: plainslice
  usernamespace: swn
  deploymentstrategy: balanced
  deploymentdomain: swntestbed
  credentials:
    username: clusterslice
    password: <sha-512-encoded-password>
  infrastructure:
    masters:
      count: 1
      osimage: "ubuntu-22-clean"
      mastertype: "vm"
    workers:
      count: 2
      osimage: "ubuntu-22-clean"
      workertype: "vm"
  kubernetes:
    kubernetestype: "vanilla"
    networkfabric: "flannel"
  applications:
    - name: argo
      version: "v3.4.4"
      parameters: "{'workflow':
        'nginx.yaml'}"
      scope: cluster
    - name: docker
      scope: all
```

Fig. 1: A Simple ClusterSlice YAML File

experimentation with a substantial set of alternative deployments, as for example those described in Section IV.

- *Application configuration*. The term "application" refers to experimentation-specific modules, i.e., user applications, Kubernetes extension deployment, or OS configuration tasks. For instance the latter include Argo Workflows, Kubernetes Dashboard GUI, Docker Engine, etc., as well as experimenter applications, defined by the *name*, *version*, *parameters* to pass in the application in JSON format, their deployment *scope* (e.g., to specific or all servers, at cluster level), etc.

Although almost self-explanatory, the YAML file depicted in Fig. 1 specifies a simple ClusterSlice deployment, under the name `plainslice`, with one server and two workers all running Ubuntu, where the network fabric is set to flannel. Multi-cluster deployments can be specified with equal ease. Further details regarding the definition of the declarative specification, as well as examples of more complex deployments are part of the framework documentation located in the repository[1].

[1]https://github.com/SWNRG/clusterslice

B. ClusterSlice Architecture

ClusterSlice architecture is composed of six main types of components: the *Kubernetes API*, the *Slice Request Operator*, the *Slice Operator*, the *Testbed Infrastructure Managers*, the *Resource Managers* and the *Cloud Infrastructure Managers* all with clearly defined roles (Fig. 2). ClusterSlice framework harvests on the extendibility of Kubernetes, hence the *Kubernetes API* is indicated as a component of the architecture, employing custom resource (CR) definitions and their respective controllers to deliver the described functionality.

In brief, the *Slice Request Operator* provides a refinement of the initial declarative description (i.e., the experimentation slice input), via an allocation process that might involve *Testbed Infrastructure Managers*, while the *Slice Operator* oversees the slice deployment and operation via the *Resource Managers*. The latter are responsible for managing participating nodes and rely on the *Cloud Infrastructure Managers* for deploying the necessary VMs. A high-level overview of the ClusterSlice architecture and the respective CR objects is presented in Fig. 2 and their role is further detailed next.

Slice Request Objects, i.e., the instances of the respective slice request CRD, are generated by the *Kubernetes API* and embody the declarative specification of an experimentation slice, outlining the intended configuration and operating state of the latter. The component handling events related to these objects is the *Slice Request Operator*, that implements the objectives specified in the former by initiating a dynamic *Resource Discovery* process. This process encompasses the specified *Compute Resources* that represent entities that host ClusterSlice nodes (such as Kubernetes nodes) and node controllers. They can be configured either by administrators or through dynamic population, as described later in this section and are used both in the *Resource Discovery* process and managed by the *Slice Operator* during operation.

The discovery process might also involve a resource embedding phase, i.e., when multiple server options exist, it selects the most appropriate ones based on particular embedding schemas, as declared at the experimentation slice specification. In the cases of deployments involving testbed resources, a dynamic allocation of physical nodes via abstracted technology-specific testbed control features, i.e., *Testbed Infrastructure Managers*, takes place, and the newly discovered resources are mapped to *Compute Resources*. The outcome of the discovery process in all cases is the generation of *Slice Objects*.

The *Slice Object* represents an active experimentation slice that enhances the targeted configuration and state with the currently active configuration and state. It is the task of the *Slice Operator* to constantly align the active state and configuration with the intended state, at the level of the experimentation slice. This task is accomplished by deploying and managing *Resource Managers* and overseeing abstracted control of clusters and cluster-level applications.

Each *Resource Manager* mirrors to the *Slice Operator* the active configuration and state of a ClusterSlice node, i.e., they embody a concept akin to a digital twin of a node. These managers employ custom Ansible playbook templates to achieve the desired configuration and state, and thus are

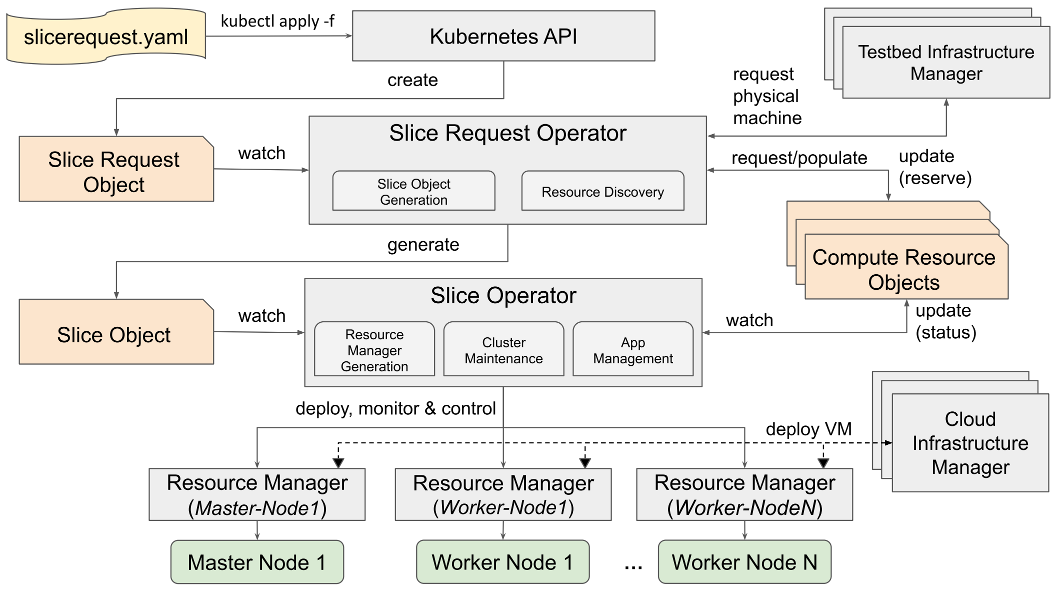


Fig. 2: ClusterSlice Architecture

tailored to specific technologies, yet they interact with the upper part of the ClusterSlice architecture in an abstracted manner. Thus, they provide *a resource abstraction layer* that offers significant versatility in on-boarding a diverse range of nodes and technologies. In the case of deployments over cloud resources, as for instance in the case of deployments over XCP-ng or even VirtualBox, *Resource Managers* rely on the *Cloud Infrastructure Managers* for the deployment of the respective VMs.

Cloud Infrastructure Managers receive a request for a VM, deploy the latter and answer back with the details necessary for the *Resource Manager* to complete the slice deployment. Each such manager is again an abstraction layer over heterogeneous virtualization technologies, such VirtualBox, XCP-ng, etc. The current implementation supports single-server or multiple-server virtualization systems, as well as the utilization of VM snapshots. Our code base provides a software toolkit to implement new infrastructure managers.

Finally, the role of the *Testbed Infrastructure Managers* is to establish communication with technology-specific cloud managers or testbed controllers within specific domains. Each such manager targets a specific testbed, and the current implementation supports CloudLab. However, we are in the process to also support Emulab and SLICES testbeds. These managers function as drivers for the heterogeneous resource management systems, handling both physical and virtual resources, forming an abstraction layer that provides extendable, seamless access to testbed resources.

C. Slice Deployment Workflow

Slice deployment is initiated by an experimenter through defining a *Slice Request Object* manifest in the form of a YAML file (Section III-A). The latter is transformed into a Kubernetes CR object, via the usual Kubernetes command (`kubectl apply -f`, Fig. 2).

The *Slice Request Operator (SRO)* is then notified about the new *Slice Request Object* and proceeds to parse the object descriptor. As mentioned, the SRO implements a *Resource Discovery* process that matches *Compute Resource Objects* with the specified slice requirements, as for instance the number and type of workers needed (i.e., *Resource Discovery* phase). The selected *Compute Resource Objects* are marked as reserved, while their details are stored in a fresh *Slice Object* manifest (i.e., *Slice Object Generation* process, Fig. 2).

It should be noted that the overall discovery and allocation process is designed to handle a range of cases with respect to the type of *Compute Resources* available. If there is a demand for open testbed nodes, these nodes are allocated through the appropriate *Testbed Infrastructure Manager*, and their details populate both new *Compute Resources* and the generated *Slice Object* manifest. In scenarios where there is a demand for new Virtual Machines (VMs), a VM embedding process might also occur in the case that two or more cloud servers are assigned to the reserved *Compute Resource* objects. This task leads to VMs allocated to specific cloud servers, via the *Cloud Infrastructure Managers*. The respective VM placement algorithm employed can be configured within the *Slice Request* manifest, contributing towards the flexibility of the approach.

Once the *Slice Object* is generated, the responsibility shifts to the *Slice Operator*, which extracts the configuration parameters of cloud servers and nodes (such as hostnames and IP addresses) from the former. It proceeds to create a dedicated *Resource Manager* for each node, incorporating the appropriate high-level configuration details, like hostnames, IPs, designated cloud servers, intended application modules for deployment, and more. In essence, *Resource Managers* are in charge of tasks such as VM allocation (*OS installation and configuration phase*), Kubernetes deployment and configuration (*K8s deployment phase*), and installing the predefined application modules (*Application installation phase*). The *Slice Operator* manages horizontal node control processes, like the establishment of the Kubernetes cluster via the communication with the appropriate *Resource Managers*.

The *Slice Object* maintains a status state along with corresponding text descriptions that convey its condition, including the ability to report failure messages. *Resource Managers* are responsible for maintaining comprehensive log information regarding the deployment process, including maintaining the appropriate resource status states for the *Compute Resources* they are responsible for, as for instance "`deploy_VM`", "`booting`", "`install_os`" and more. In the unlikely event that a *Resource Manager* fails or it is terminated, it will automatically restart and continue configuring its assigned node from its current state and onwards.

The completion of the above described workflow leads to the new experimentation slice being up and running. Finally, user access to the slice is achieved via the admin user credentials specified in the *Slice Request* object.

D. Multi-Clustering and Multi-Domain Capabilities

We have recently extended the above single cluster architecture to support multi-cluster and multi-domain functionalities in a rather "recursive" manner, i.e., via introducing a new Multi-Cluster SliceRequest CR and a corresponding Operator. This higher-level abstraction is responsible for creating and overseeing multiple *Slice Request* objects, which in turn allocate the clusters that constitute the multi-cluster slice. It also manages the deployment of multi-cluster management software, such as Liqo or OCM, currently in the form of application modules.

Furthermore, experimenters can specify a deployment domain within a *Slice Request* definition. Multi-Cluster *Slice Request* manifests offer the flexibility to define multiple domains, with one domain per cluster. Enhanced multi-domain capabilities can be enabled by deploying ClusterSlice across multiple clusters, where *Infrastructure* and *Resource Managers* can be distributed at the various domains to speed-up deployments and improve security. However, this aspect is complex enough to deserve a further study.

IV. Experimentation Analysis

In this Section, we detail: i) the methodological aspects of our experimental analysis, including the considered scenarios and metrics, as well as the configuration of the testbed we used; and, ii) our proof-of-concept results, which showcase the key features of ClusterSlice, as elaborated below.

A. Methodology

Our analysis is based on the following distinct experimental scenarios:

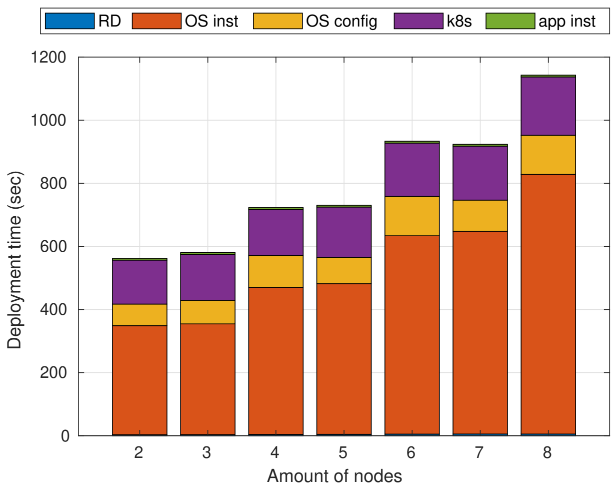
- *Scenario 1 – Single-cluster deployment*. The first scenario focuses on the declarative, automated deployment of clusters with increasing sizes. It also assesses a main infrastructure-as-a-service attribute of ClusterSlice, i.e., through the incorporation of experiments with and without VM snapshots.
- *Scenario 2 – Multi-clustering*. This scenario highlights ClusterSlice's multi-clustering capabilities, through the deployment of multiple clusters and the automated installation and configuration of Open Cluster Manager (OCM).
- *Scenario 3 – Multiple Kubernetes flavors and network plugins*. It demonstrates the deployment of multiple Kubernetes flavors and network plugins, while quantifying the associated deployment times. These results underline the platform-as-a-service capabilities of ClusterSlice.
- *Scenario 4 – ClusterSlice as an experimentation platform*. The last scenario showcases the experimentation automation features of ClusterSlice, performing a comparative analysis between different plugins, as well as, between AD approaches. This scenario focuses on the application-as-a-service capabilities of ClusterSlice.

Concerning the performance metrics considered, in the first three scenarios we focus on the completion times of all phases involved in the deployment process: i) resource discovery (RD); ii) operating system (OS) installation (OS inst); iii) OS configuration (OS config); iv) Kubernetes deployment (k8s); and, v) application installation (app inst). The resource discovery phase is performed by the *Slice Request Operator*, while the rest are handled by the Resource Managers. Such measurements are collected by the *Slice Operator*, which watches the state changes of *Compute Resources*.

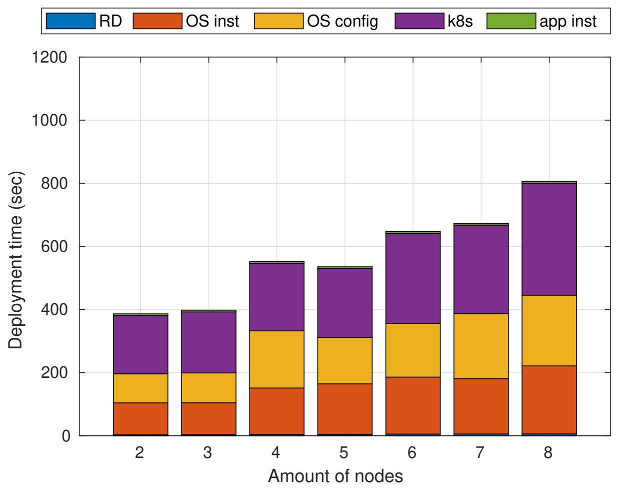
In the fourth scenario, the resource utilization is measured in terms of CPU utilization (%) and memory consumption (MB). For evaluating network plugins, we take into account the average throughput (Mbps). Regarding anomaly detection mechanisms, we measure (i) detection time, referring to the time duration between the occurrence and the estimation of the anomaly, in milliseconds (ms); and, (ii) response time, representing the average time, in ms, for a data point to be processed by the anomaly detection mechanism, including the client-server communication cost. To conduct these experiments, we employ the k8s-bench-suite[2], a widely-used benchmarking tool designed to measure the network performance of Kubernetes clusters. The anomaly detection workflows are implemented using Kubernetes Argo. The corresponding mechanisms are included within a MATLAB container that features a REST API, enabling the selection and configuration of a mechanism, as well as the communication of measurements.

Next, we will provide details about the specifications of the experimental testbed we used. The University of Macedonia (UoM) testbed consists of 2 Dell PowerEdge R360 servers, each equipped with a dual Intel(R) Xeon(R) ES-2620 v4 CPU running at 2.10GHz, 1.5TB of SSD storage, and 64GB and 34GB of RAM, respectively. These servers host the XCP-ng virtualization platform[3]. The VM nodes within the testbed are operating Ubuntu 22.04.2 LTS and are utilizing kernel version 5.15.0-71-generic. Notably, the baseline inter-communication plugin used among the nodes is the Flannel CNI plugin[4].

[2]https://github.com/InfraBuilder/k8s-bench-suite
[3]https://xcp-ng.org/
[4]https://github.com/flannel-io/flannel



(a) Regular VM deployment.



(b) VM deployment with snapshots.

Fig. 3: ClusterSlice deployment times across two physical servers for the two VM deployment approaches with respect to a varying number of cluster sizes.

To enhance the statistical accuracy of our results, each scenario has been conducted 10 times and we document the average values of the replications.

B. Experimental Results

1) Scenario 1: Single-cluster deployment: In the initial scenario, we start with a single cluster setup where the number of nodes varies, ranging from 2 to 8. In this setup, one node takes on the role of the master, while the rest function as workers. Additionally, we implement two alternatives for VM deployment: (i) regular VM deployment; and (ii) VM deployment with snapshots. This approach allows us to measure the impact of using snapshots on the overall installation times.

In Fig. 3, we illustrate ClusterSlice's completion times for each individual deployment phase across different cluster sizes. When comparing VM deployment without snapshots, as shown in Fig. 3(a), to VM deployment with snapshots, depicted in Fig. 3(b), it becomes evident that the former approach results in a significant increase in the duration of ClusterSlice deployment, particularly as the number of in-cluster nodes increases. The extended duration can be primarily attributed to the OS installation phase, which is a computationally and network-intensive operation. This phase also makes the most significant contribution to the overall deployment time.

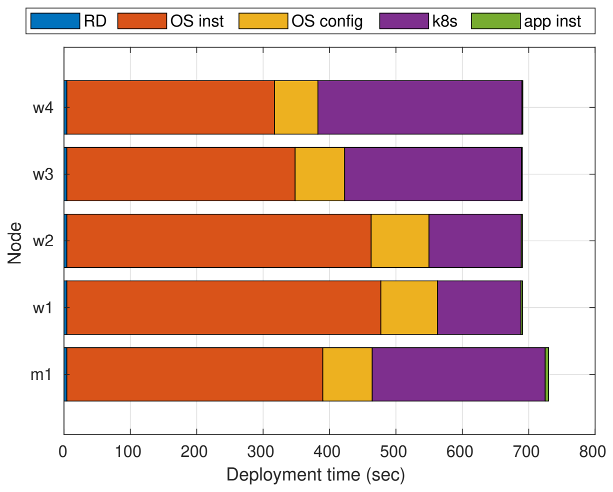
Specifically, VM deployment with snapshots exhibits a remarkable decrease of up to 30% when compared to the scenario without VM snapshots across all variations of in-cluster nodes being examined. In contrast, the number of cluster nodes does not seem to have a noticeable impact on the overall deployment time for other installation phases. For instance, the Kubernetes installation phase, for both options, remains consistent as the number of nodes increases.

Next, we aim to gain a better understanding on the parallel behavior of the deployments and the dependencies between the respective phases. Fig. 4 depicts the duration of each installation phase on each individual node in the experiment. In this case, a cluster with a total of five nodes (m1 being the master node and w<x> representing the worker nodes) is considered, both with and without the option of VM snapshots. Each horizontal bar in the figure represents the time interval of the corresponding phase (PD, OS installation, etc.) on each node. It indicates the start and end times of each phase. As shown, the deployment duration for the OS installation and configuration phases varies among the nodes due to the placement decisions of the VM nodes on the two available servers in the testbed. The latter follows a simple "round-robin" approach, allocating the master in the first physical machine, and then alternating allocation of workers among the two hosts (i.e., w1 on the first server, w2 in the second, etc.). As a result, in this specific instance, m1, w1 and w3 nodes reside in the first physical machine, while w2 and w4 in the second.

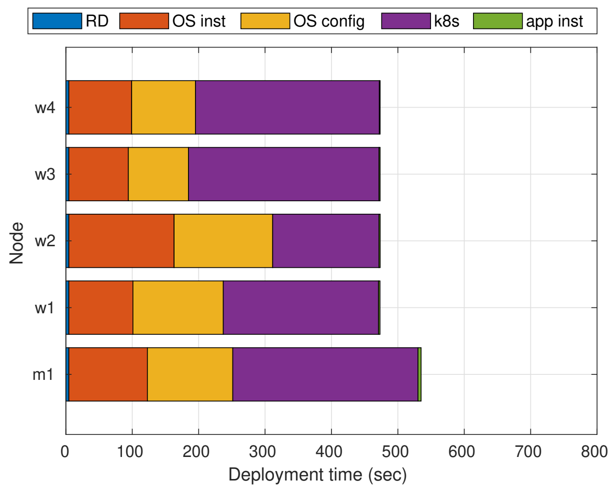
Finally, regarding the k8s deployment, the master node incurs the longest deployment time, primarily because of synchronisation issues in cluster creation. In practice, the master has to create the cluster, generate the necessary token for worker nodes to join, and then to wait until all worker nodes have successfully joined. This can be observed in the end times in Fig. 4, where worker nodes complete the k8s deployment simultaneously, once they receive the cluster token. Note that if a worker node concludes the OS configuration installation (e.g., w3 and w4) before the master node (e.g., m1), the former node waits until the latter proceeds to the k8s deployment phase.

2) Scenario 2: Multi-clustering: In this experiment, we demonstrate the operation of ClusterSlice over multi-cluster Kubernetes deployments, through the incorporation of Open Cluster Management (OCM), an open-source Kubernetes multi-cluster orchestration platform.

Following the OCM mindset, we specify one cluster as the hub, while the remaining clusters serve as the managed clusters (ranging from 1 to 3). Each cluster contains three



(a) Regular VM deployment.



(b) VM deployment with snapshots.

Fig. 4: ClusterSlice deployment time analysis (breakdown) for a cluster of five virtual nodes across two physical servers.

nodes, i.e., a master and two worker nodes. This scenario considers exclusively deployments using VMs with snapshots, due to the deployment time advantages they provide.

Fig. 5 depicts the OCM deployment time along with the different phases of the deployment process, for a varying number of participating clusters, aiming also to provide insights into the scalability aspects of this approach.

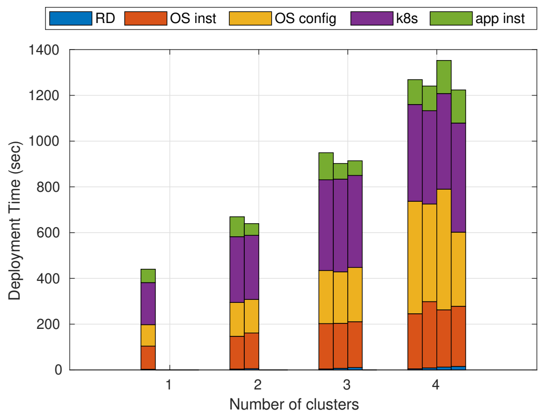


Fig. 5: Multi-cluster OCM deployment time for different number of clusters.

As expected, the deployment time increases with the number of clusters, while the deployment time for each individual cluster, across the different multi-cluster deployments, is distributed fairly evenly. Next, focusing on the deployment phases, it appears that the OS configuration and k8s installation phases have the most significant impact on the overall deployment time. The impact of OS installation, on the other hand, remains relatively low due to the utilization of VM snapshots. Finally, in contrast to the single cluster approach, we observe an increased impact on the application installation time, since the latter also includes the OCM installation.

3) Scenario 3: Multiple Kubernetes flavors and network plugins: In the third scenario, we assess ClusterSlice's ability to support multiple Kubernetes flavors and intra-cluster network plugins, highlighting its potential in providing composable Kubernetes deployments. With respect to the different Kubernetes flavors, we consider the commonly used vanilla Kubernetes that offers a fully featured k8s distribution and two lightweight k8s distributions, i.e., k3s and k0s, which target edge computing and IoT environments.

We also consider the following widely-used network plugins: (i) *Flannel*, a simple and easy-to-use network plugin that provides basic IP networking connectivity; (ii) *Calico*, which is a more advanced and feature-rich plugin that supports advanced security policies and network segmentation; and, (iii) *Kuberouter*, a lightweight and highly efficient Kubernetes network plugin that focuses on simplicity and performance. These plugins are also supported in k3s and k0s flavors, i.e., the latter have a more limited set of options compared to vanilla k8s.

Furthermore, we consider a single-cluster environment consisting of three nodes, with one acting as the master node and the rest as worker nodes. The results of this experiment are illustrated in Fig. 6 and are discussed below.

Initially, we observe that the variation in the overall deployment time, for the alternative Kubernetes flavors and network plugin options, mainly depends on the Kubernetes deployment. In this context, vanilla k8s based on the Calico network plugin results on the higher overall deployment time while the integration of k3s with Flannel to the lowest. In general, the Calico plugin leads to longer Kubernetes deployment times, while Flannel results in shorter deployment times for all the considered Kubernetes flavors. This result is consistent with existing literature, as seen in [21].

When comparing the Kubernetes flavors, the lightweight options, such as k3s and k0s, outperform the vanilla k8s solution in terms of deployment time, regardless of the network plugins under consideration. For instance, with the Calico

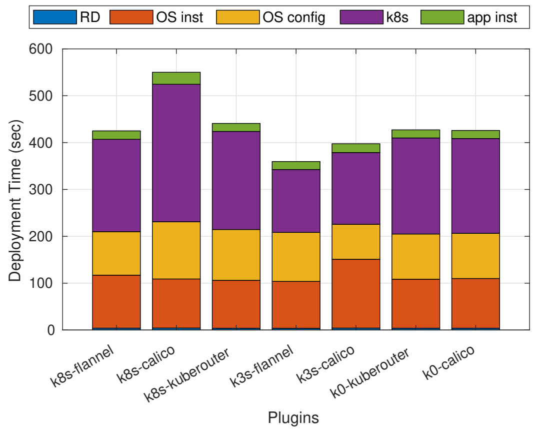


Fig. 6: ClusterSlice deployment time over different combinations of Kubernetes flavors/network plugins.

plugin, vanilla k8s completes the deployment with 37% higher deployment time compared to k3s and 34% higher compared to k0s.

4) Scenario 4: ClusterSlice as an experimentation platform: This last scenario showcases ClusterSlice's ability to function as an experimentation testbed, supporting a diverse range of experimental objectives and requirements. It allows for the effortless definition of complex experimental scenarios involving various deployment alternatives.

More specifically, we demonstrate the execution of two distinct experiments: (i) a networking benchmark across various Kubernetes flavors and network plugins; and (ii) a performance evaluation test between two different machine learning-based anomaly detection techniques, namely the ratio-type CUSUM and a standard CUSUM detector as described in [22].

In both experiments, the cluster comprises a master and a worker node to gain insights into the ideal maximum performance. It is worth noting that both the definition and execution of experiments were easily accomplished using appropriate *Slice Request* manifests.

Regarding the first experiment, Fig. 7 depicts the average throughput (Mbps) obtained for TCP/UDP communication between pods (pod2pod) and communication between pods and services (pod2svc) across different Kubernetes flavors and network plugins. As observed, the Kuberouter plugin outperforms both Calico and Flannel in terms of throughput for all communication types, while the Flannel plugin consistently exhibits the lowest throughput. Additionally, Flannel notably decreases its throughput performance for the TCP protocol, whereas Calico exhibits a decrease for the UDP protocol, within both vanilla Kubernetes (k8s) and k3s flavors. In contrast, the Kuberouter plugin maintains its throughput performance consistently across all communication types and Kubernetes flavors considered.

Concerning resource utilization, Fig. 8 illustrates the CPU utilization and memory consumption, for both client and server pods. These results highlight: (i) the lightweight nature of Flannel, when compared to Kuberouter and Calico, and, (ii) the resource intensive performance of the Calico plugin, in consistency to the outcomes of Scenario 3 (Fig. 6). Additionally, CPU usage is not significantly affected from the choice of Kubernetes flavor or the network plugin, especially concerning the client utilization. On the other hand, memory usage with the Calico plugin is notably greater than with the Kuberouter and Flannel. As expected, resource utilization, in terms of both memory and CPU, is higher for the nodes hosting server pods.

Next, we further demonstrate ClusterSlice's capabilities as an experimentation platform that extends beyond conducting networking experiments, showcasing its ability to enable easy experimentation with more complex workflows. In more detail, we demonstrate its effectiveness in evaluating machine learning approaches to gain insights into their performance for real-world applications. In this context, Fig. 9 highlights the operation of both a ratio-type CUSUM and a typical CUSUM change-point detector, assessing their detection and response times, as well as their CPU and memory consumption in parallel. Validation is performed using synthetic time-series (X_t) of length $N = 200$, following the standard Normal distribution $X_t \sim \mathcal{N}(0, 1)$ and introducing a single change-point (CP), at the time instance $t = 100$, increasing the mean value by 1.

In Figs. 8(a)–(b), we illustrate the CPU and memory usage of both CP procedures, highlighting how the choice of the Kubernetes flavor and network plugin impacts the resource utilization of the ML procedures. Our results are consistent with the findings from our previous experiments. For instance, both procedures consume more resources when the Calico network plugin is employed. Interestingly, despite the fact that the ratio-type CUSUM is more computationally intensive compared to the typical CUSUM, when using the Flannel plugin, it exhibits lower resource consumption compared to the typical CUSUM with the Calico plugin. Furthermore, according to Figs. 8(c)–(d), the detection and response times (processing time for each time step) of both procedures are also affected by the selected plugin.

Last but not least, for the needs of the four experimentation scenarios discussed above, we deployed a total of 410 clusters and 1530 nodes. Remarkably, there was not a single failure, which serves as the first validation of the reliability advantages of ClusterSlice. In the future, we also plan to further assess this aspect using chaos engineering methodologies.

V. Conclusions

ClusterSlice contributes towards an effortless, zero-touch solution for transforming testbed resources from bare-metal to fully-operational experimentation slices, targeting automated kubernetes-centered experimentation. Its well-designed abstractions, harvesting on the CRD and Operator patterns of Kubernetes offer the ease of a declarative specification and allow to act as a infrastructure/platform/application-as-a-service solution, over single and multi-cluster, multi-domain environments, that include local and remote testbed resources. The current paper demonstrates both its potential in offering the above as well as to act towards its aims to reduce experimentation complexity, while maintaining and improving reliability and reproducibility.

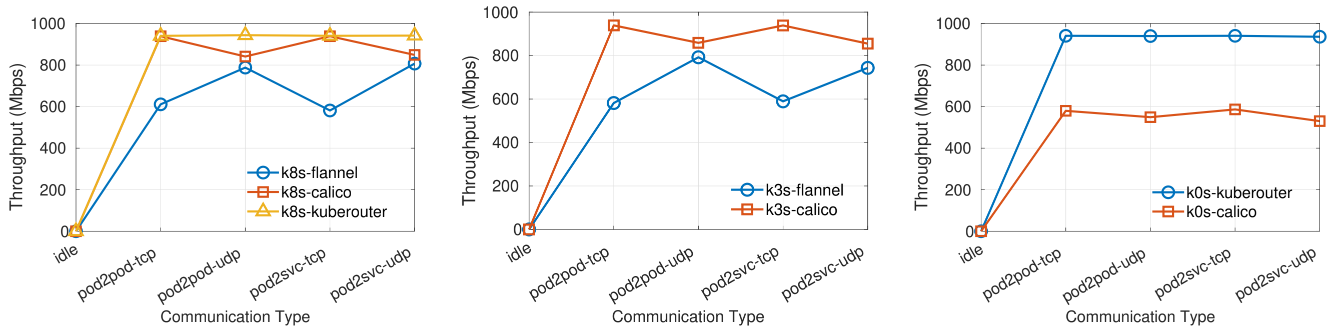


Fig. 7: Throughput of network plugins for different Kubernetes flavors and communication types.

The current platform can be extended to a number of directions. The first includes building of an open-source community around the current implementation, in order to test its capabilities in a wider, more diverse setting. The second is to investigate a number of optimization problems that arise in the various phases of the slice deployment, such as the resource discovery and VM placement, problems that have been of a concern to the wider research community. Finally, a number of extensions regard on-boarding more experimentation facilities by including more Cloud and Testbed Infrastructure Managers.

REFERENCES

[1] “Ansible,” https://www.ansible.com/, Accessed, October 2023.

[2] “SLICES Test-Beds,” https://portal.slices-sc.eu/, Accessed, October 2023.

[3] I. Baldin, A. Nikolich, J. Griffioen, I. I. S. Monga, K.-C. Wang, T. Lehman, and P. Ruth, “Fabric: A national-scale programmable experimental network infrastructure,” *IEEE Internet Computing*, vol. 23, no. 6, pp. 38–47, 2019.

[4] “PlanetLab,” https://planetlab.cs.princeton.edu/, Accessed, October 2023.

[5] “OMF,” https://omf.orbit-lab.org/, Accessed, October 2023.

[6] “Emulab,” https://gitlab.flux.utah.edu/emulab, Accessed, October 2023.

[7] “GENI tools,” https://github.com/GENI-NSF/geni-tools, Accessed, October 2023.

[8] “jFed and jFed-cli,” https://jfed.ilabt.imec.be/, Accessed, October 2023.

[9] “CloudLab,” https://www.cloudlab.us/, Accessed, October 2023.

[10] “Portal Tools,” https://gitlab.flux.utah.edu/stoller/portal-tools, Accessed, October 2023.

[11] “CloudNativeLab,” https://practicum.cloudnativelab.ilabt.imec.be/, Accessed, October 2023.

[12] B. C. Şenel, M. Mouchet, J. Cappos, O. Fourmaux, T. Friedman, and R. McGeer, “Edgenet: a multi-tenant and multi-provider edge cloud,” in *Proc. 4th ACM Int. Workshop Edge Syst., Analytics and Netw. (EdgeSys)*, Edinburgh, Scotland, UK, 26 Apr. 2021, pp. 49–54.

[13] “EdgeNet,” https://www.edge-net.org/, Accessed, October 2023.

[14] J. Breen, L. Bryant, G. Carcassi, J. Chen, R. W. Gardner, R. Harden, M. Izdimirski, R. Killen, B. Kulbertis, S. McKee, B. Riedel, J. Stidd, L. Truong, and I. Vukotic, “Building the slate platform,” in *Proc. ACM Pract. Experience Adv. Res. Comput. (PEARC ’18)*, Pittsburgh, PA, USA, Jul. 2018, pp. 1–7.

[15] “Kubernetes-Native-Testbed,” https://github.com/kubernetes-native-testbed/kubernetes-native-testbed, Accessed, October 2023.

[16] “Kubespray,” https://github.com/kubernetes-sigs/kubespray, Accessed, October 2023.

[17] “Crossplane,” https://github.com/crossplane/crossplane, Accessed, October 2023.

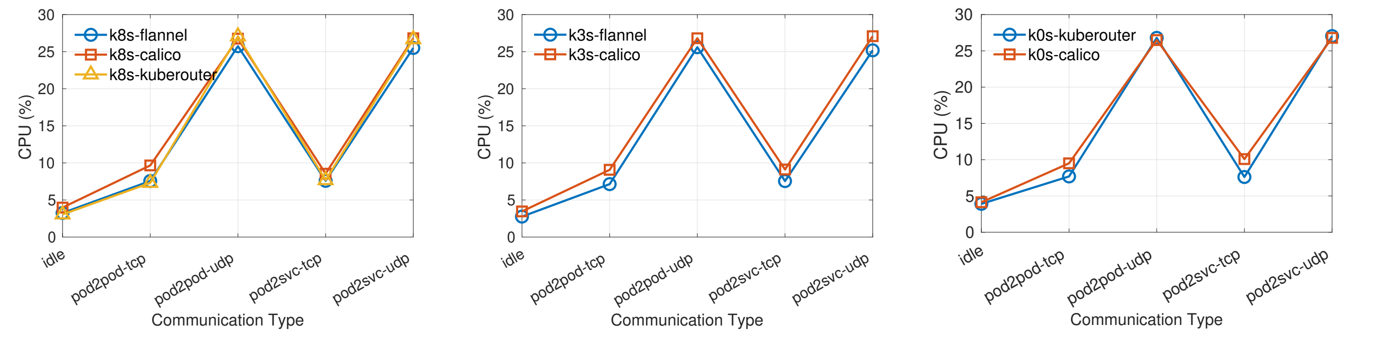
[18] P. Valsamas, I. Sakellariou, S. Petridou, and L. Mamatas, “A multi-domain experimentation environment for 5G media verticals,” in *Proc. IEEE Int. Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, Paris, France, Apr. 2019, pp. 461–466.

[19] P. Valsamas, P. Papadimitriou, I. Sakellariou, S. Petridou, L. Mamatas, S. Clayman, F. Tusa, and A. Galis, “Multi-pop network slice deployment: A feasibility study,” in *Proc. 8th IEEE Int. Conf. Cloud Netw. (CloudNet)*, Coimbra, Portugal, Nov. 2019, pp. 1–6.

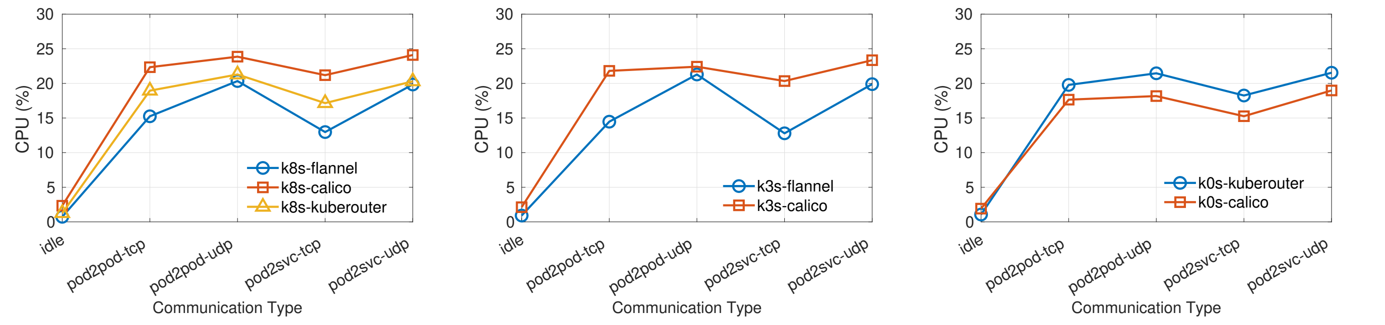
[20] S. Clayman, A. Neto, F. Verdi, S. Correa, S. Sampaio, I. Sakelariou, L. Mamatas, R. Pasquini, K. Cardoso, F. Tusa *et al.*, “The necos approach to end-to-end cloud-network slicing as a service,” *IEEE Commun. Mag.*, vol. 59, no. 3, pp. 91–97, 2021.

[21] S. Qi, S. G. Kulkarni, and K. Ramakrishnan, “Assessing container network interface plugins: Functionality, performance, and scalability,” *IEEE Trans. Netw. Service Manag.*, vol. 18, no. 1, pp. 656–671, 2020.

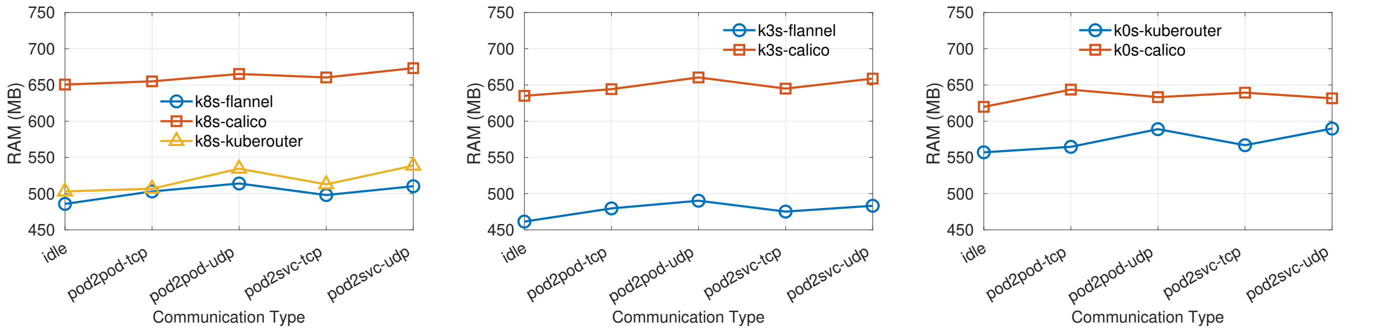
[22] S. Skaperas, L. Mamatas, and A. Chorti, “Real-time video content popularity detection based on mean change point analysis,” *IEEE Access*, vol. 7, pp. 142 246–142 260, 2019.



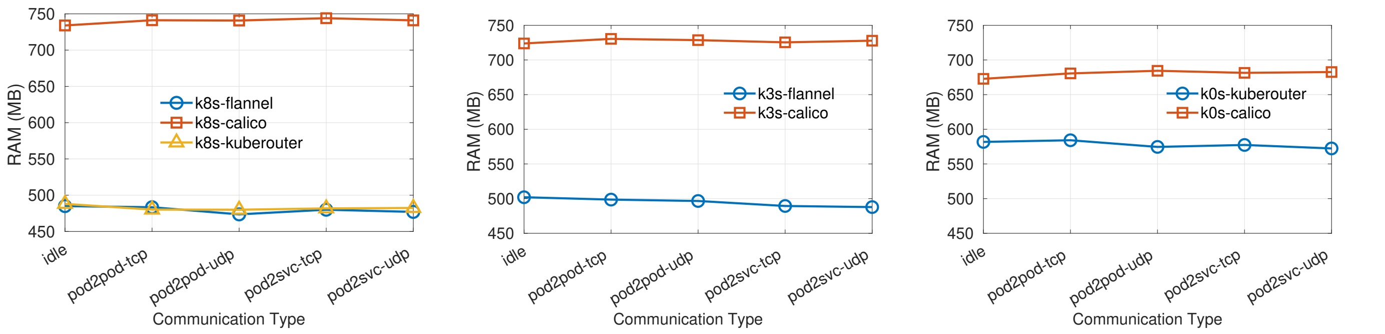
(a) CPU utilization of node hosting client pods.



(b) CPU utilization of node hosting server pods.

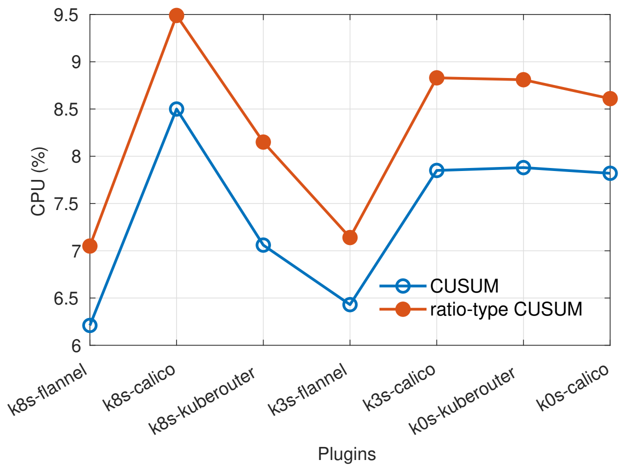


(c) RAM consumption of node hosting client pods.

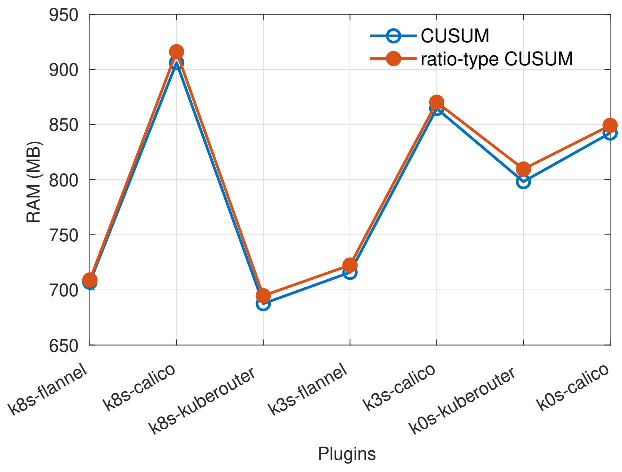


(d) RAM consumption of node hosting server pods.

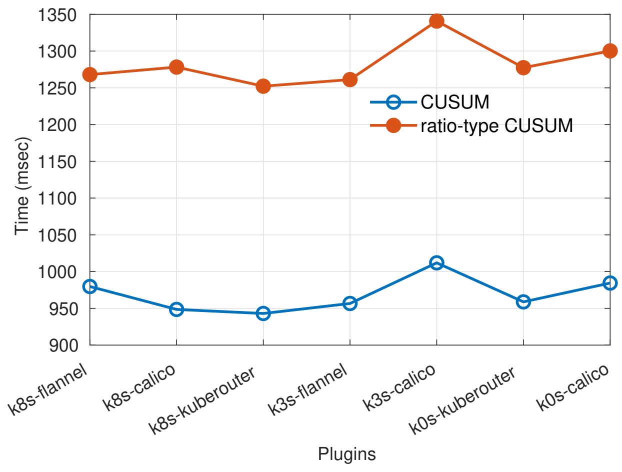
Fig. 8: CPU utilization and RAM consumption for different combinations of Kubernetes flavors/network plugins, for the nodes hosting client (rows (a) and (c)) and server pods (rows (b) and (d)).



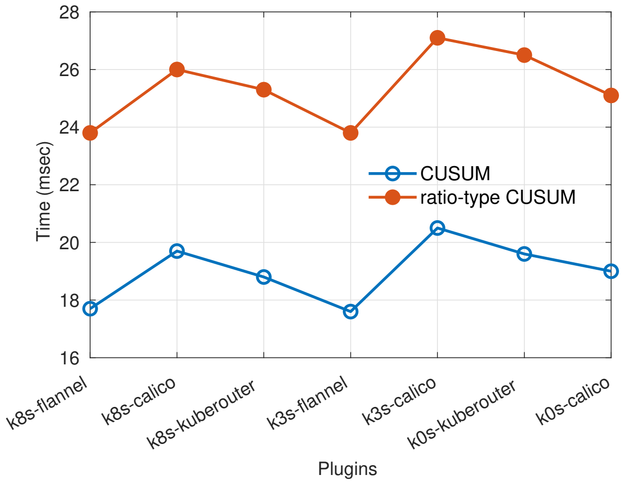
(a) CPU utilization.



(b) RAM consumption.



(c) Detection time.



(d) Response time.

Fig. 9: Comparison of (a) CPU utilization, (b) RAM consumption, (c) detection time, and, (d) response time between two different anomaly detection schemes, over several Kuberneters flavors/network plugins.