Arktos: A Hyperscale Cloud Infrastructure for Building Distributed Cloud

Ying Huang
Yunwen Bai
Sonya Li
Peng Du
yhuang1@futurewei.com
ybai1@futurewei.com
fli2@futurewei.com
pdu@futurewei.com
Futurewei Technologies, Inc.
Santa Clara, California, USA

Xiaoning Ding
Qian Chen
xiaoning.ding@bytedance.com
qian.chen@bytedance.com
Bytedance, Inc.
Seattle, Washington, USA

Deepak Vij
Hongwei Chen
Vinay Kulkarni
Ying Xiong
dvij@futurewei.com
hong.chen@futurewei.com
vkulkarn@futurewei.com
yxiong@futurewei.com
Futurewei Technologies, Inc.
Santa Clara, California, USA

ABSTRACT

Cloud Computing has become the new norm in today's digitized world. It has allowed us to do business in a faster and more scalable way with less cost and more efficiency [15, 37, 44]. As a result, cloud infrastructure scalability and unification of cloud resource orchestration systems are few of the top key challenges for the cloud providers. In this paper, we present Arktos, a cloud infrastructure platform for managing a large-scale compute cluster and running millions of application workload instances as containers and/or virtual machines (VMs). Arktos is an open-source Linux Foundation project [8] envisioned as a stepping-stone from current "single-region" focused cloud infrastructure towards next generation "region-less" distributed infrastructure in the public and/or private cloud environments.

We present details related to the Arktos system architecture and features, important design decisions, and the results and analysis of the performance benchmark testing. Arktos achieves high scalability by partitioning system components into multiple resource partitions (RPs) and tenant workload partitions (TPs), with each partition scaling independently. Our performance testing using Kubemark test tool [38] demonstrates that Arktos with just two RPs and two TPs system setup can already manage a large scale cluster of 50K compute nodes and is able to run 1.5 million application containers with 5 times system throughput (QPS) compared with an existing container management system.

Three key characteristics differentiate Arktos from other open-source cloud platforms. Firstly, Arktos architecture is a truly scalable architecture that supports a very large cluster by scaling more TPs and RPs in the system, Secondly, it unifies the runtime infrastructure to run and manage both VM and container applications natively, therefore eliminating the cost of managing separate technology stacks for VMs and containers. Lastly, Arktos implements a unique "virtual cluster" style multi-tenancy design that provides both strong tenant data isolation and network communication isolation. We present in-depth design details, performance benchmark test results as well as future work planned for Arktos."

SoCC'22, November, 2022, San Francisco, CA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

Cloud Infrastructure, cluster management, distributed system, Container, Virtual machine, Kubernetes, IaaS.

ACM Reference Format:
Ying Huang, Yunwen Bai, Sonya Li, Peng Du, Xiaoning Ding, Qian Chen, Deepak Vij, Hongwei Chen, Vinay Kulkarni, and Ying Xiong. 2022. Arktos: A Hyperscale Cloud Infrastructure for Building Distributed Cloud. In *Proceedings of ACM Symposium on Cloud Computing (SoCC'22).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Cloud Computing has become the new norm in today's digitized world. It has allowed us to do business in a faster and more scalable way with less cost and more efficiency [15, 37, 44]. In the last decade or so, major public cloud providers like Amazon Web Services [21], Microsoft Azure [22] and the Google Cloud Platform [13] have built their own large-scale cloud computing infrastructures to meet ever evolving business digitization needs [20]. Large enterpise companies, such as Meta, Salesforce, and Vodafone, NTT etc. are inspired by these hyper-scaler public cloud providers and are emulating the hyper-scale cloud infrastructure designs as part of their own respective enterprise cloud architecture [52].

As companies move more and more business-critical applications to the cloud, and with the emergence of new types of compute intensive applications such as AI workloads, the scalability and performance of cloud infrastructure continue to be challenged [14, 61]. One of the major factors affecting the scalability of a cloud infrastructure platform is the number of compute nodes a typical cloud platform can manage. The larger the number of compute resources a cloud platform can manage, the more scalable the platform becomes by way of adding more resources to the overall system in order to deploy more customer application instances. This, in turn, results

in higher elasticity, higher availability and more resilience to the user applications deployed within the cloud cluster environment.

Of course, managing a large number of compute nodes (cluster size) incurs certain cost as the platform needs to connect, aggregate and coordinate more compute resources to work together, monitor more nodes for failures and recovery, and spend more time in resource scheduling for allocating nodes to specific applications or jobs based on their resource needs while trying to maintain higher resource utilization for the cluster [16]. In a large-scale cloud environment, however, studies [57, 60] have shown that the cost of managing a large compute cluster is much lower than that of managing multiple independent smaller clusters. A typical hyper-scaler cloud data center, on an average, can host about 100,000 compute servers [46].

Openstack [3] is one of the popular open-source cloud infrastructure platforms. Assuming one OpenStack cluster supports 1,000 compute nodes in it, for a cloud provider to manage 100,000 nodes in a data center, one may need to deploy and manage 100 or so independent OpenStack clusters. The largest OpenStack deployment we are aware of at the time of this writing is [47], which manages around 2,500 compute nodes in a single OpenStack cluster. In this case, we would still need to have 40 independent OpenStack clusters for a typical cloud data center environment. For large scale cloud providers, managing 40 to 100 independent compute clusters per data center is an unbearable cost. Additionally, scaling OpenStack cluster continues to be a major challenge for cloud providers who employ Openstack system as the underlying cloud infrastructure platform [48, 49, 50, 53].

For hyper-scale public cloud providers, such as Amazon AWS, Microsoft Azure and Google Cloud, the scalability and high infrastructure management cost are among the top challenges to them as well [35, 43, 56]. While there are no official data publicly available from these hyper-scaler cloud providers regarding their proprietary infrastructure platforms and the cluster size they can support, it is commonly understood that the number is less than approx. 10,000 nodes or so per cluster.

Kubernetes [39] is yet another very popular open-source cloud platform which has become a de facto standard for orchestrating and managing containerized application workloads in the cloud. Almost all cloud providers offer container services based on Kubernetes, such as GKE [11] by Google, AKS [7] by Azure and EKS [6] by AWS. The state-of-art system manages a cluster of (physical or virtual) compute nodes and automates the deployment, management, and scaling of application containers [39]. Officially, the open-source release of Kubernetes version v1.24 supports a cluster with up to 5000 nodes [41], while Google version of Kubernetes, Google Kubernetes Service (GKE), supports a cluster of 15,000 compute nodes [12, 10]. Unfortunately, while Kubernetes platform supports larger cluster deployment than OpenStack and exhibits much better extensibility in architecture, it still does not meet the scalability challenge faced by cloud providers as business grows rapidly. Running more than 10 independent Kubernetes clusters in a data center still incur much higher management cost as compared to having less than 5 clusters per data center. This is mainly because each independent cluster requires its own control plane nodes and high availability (HA) setup etc.

Additionally, Kubernetes, as good as it is, it is not designed for a cloud infrastructure environment that requires traditional bare metal server and native virtual machine (VM) support as well as multi-tenancy support for strong isolation among tenants. Unfortunately, Kubernetes platform does not support any of these requirements. For example, Kubernetes network design allows every pod (container) talking to every other pod in the same cluster as all pod IP addresses are in the same address space [40]. As a result, majority of the usages of Kubernetes are strictly for a single tenant cluster only [6, 7, 11], meaning that the entire cluster is dedicated to a single tenant.

Furthermore, both public cloud and enterprise private cloud aren't just about containerized workloads only as containers are not appropriate for some enterprise workloads and use cases. Today, most enterprises require cloud infrastructure providers to support and manage a mix of workloads running on bare-metal servers, virtual machines, containers and serverless platform. Unfortunately, current cloud infrastructure systems (open-source or proprietary) don't have the capability to provide support for managing and orchestrating the heterogeneous workload types via a "unified" resource management and orchestration platform as a single pane of glass. In summary, there is no contemporary open source solution that addresses the needs of a very large scale cloud infrastructure and at the same time manages compute and network resources across all workload types in a unified manner. As cloud scale grows, this challenge becomes only more crucial and important to cloud providers and enterprises.

In this paper, we present Arktos, a cloud infrastructure platform for managing a large-scale compute cluster and running millions of application instances as containers and/or virtual machines (VMs). Arktos is an open-source Linux Foundation project envisioned as a stepping-stone from current "single-region" focused cloud infrastructure towards next generation "region-less" distributed infrastructure in the public and/or private cloud environments. We present details related to the Arktos system architecture and features, important design decisions, and the results and analysis of the performance benchmark testing. Arktos achieves high scalability by partitioning system components into multiple resource partitions (RPs) and tenant workload partitions (TPs), with each partition scaling independently. Our performance testing using Kubemark benchmarking tool [38] demonstrates that Arktos with just two RPs and two TPs system setup can already manage a large scale cluster of 50K compute nodes in it and be able to run 1.5 million app containers with 5 times system throughput (QPS) compared with an existing container management system. Arktos makes the following contributions:

- Arktos designs and implements a novel and a truly extensible platform, inspired by Kubernetes architecture, that supports a very large cluster by scaling resource management and tenant workload management separately in the system.
- Arktos abstracts application workload (pod concept) further to separate the workload itself from runtime environment. Arktos then implementes runtime as a unified runtime infrastructure to run and manage workloads as VM or container or any other type of runtime environment such as WebAssembly.

- Arktos introduces and implements "Space" and "Tenant" concepts as part of the multi-tenancy design that provides strong tenancy isolation, relatively to Kubernetes single tenant platform.
- Arktos introduces "VPC" and "Subnet" objects for strong network isolation, relative to Kubernetes flat network design. Each VPC or subnet has its own IP address range and each workload instance is deployed within a subnet or VPC.
- Lastly, we successfully achieved 50,000-nodes cluster benchmark tests using Kubemark [38] simulation testing tool and analyzed test results of 1.5 million workload instances for performance and throughput.

Paper outline: Section 2 firstly enumerates all the current challenges and subsequently discusses how it aims to be a modern cloud infrastructure platform for addressing such challenges. Section 3 describes the Arktos scalability design explaining how it provides support for public cloud infrastructure grade highly scalable and a highly available regional control plane capability. Section 4 describes Arktos unified runtime infrastructure for seamlessly managing containers and virtual machines. Section 5 describes multi-tenancy related in-depth details. Section 6 captures performance tests details related to how Arktos scale-out design provides support for large number of nodes in a cluster. This section also covers how Arktos scale-out cluster performs under high throughput environment. Section 7 compares Arktos with similar cloud infrastructure platforms, either open-source or proprietary. Finally, we present concluding remarks and avenues for future work in section 8.

2 BACKGROUND

We started Arktos project with the goal of replacing 3 distinct existing systems with one infrastructure platform for our public cloud services [36]. These three distinct systems are based on different technology stacks – one for provisioning and managing virtual machines and bare-metal servers, the second for deploying and orchestrating containerized applications, and the third system for scheduling and running Serverless related workloads. In addition to high overall management costs, all the three systems have scalability and performance issues to meet ever growing business needs. The following is a brief list of notable key requirements from our customers and partners, hence the challenges for current technology stacks.

- Large customers want to be able to spin up and run up to 10K virtual machines (VMs) in their peak scenarios within minutes. This poses a major scalability challenge to the small clusters as, at any given time, one cluster may not have enough resources to provision the number of VMs requested. This also poses the throughput and performance challenges to the system technology stack as the system can't provision thousands of VMs within the time requested.
- Many customers have requirements to deploy and manage their applications in the cloud with a mix of VMs and containers as part of a single deployment. For example, some customers have designed their applications with micro-service architecture where some micro-services are designed to run directly in dedicated VMs while others are designed to run as Docker containers in bare-metal servers. Unfortunately, this poses many challenges to current three independent cloud platforms. There is a paramount need to unify the platforms with the same set of API interfaces and overall user experience.
- With businesses containerizing their applications and moving them to run in the cloud, many customers have asked for an ability, for example, to use same Kubernetes APIs to deploy and run their containerized applications without having to worry about building and managing dedicated Kubernetes clusters. We call this Serverless container service where a container cluster is shared with multiple customers (tenants) as long as a strong isolation is provided across tenants. As a result, this poses a scalability challenge to the container platform where we need to run millions of containers with hard multi-tenancy support.
- Some of our partners who help manage the cloud environment have asked for the capability to support and manage more than 100,000 compute nodes in a region in order to reduce the control plane overheads and overall operational costs.

Instead of solving these challenges separately for each system, obviously, building one platform or unifying existing systems into one is a much better solution. Inspired by Kubernetes's flexible architecture design (CRI [29], CNI [31] and CSI [27] interfaces) and its rich workload orchestration patterns (replicaSet, statefulSet, DaemonSet, etc.), as well as its prosperous ecosystem and community (CNCF [23]), we decided to tackle the scalability, performance, and API experience challenges based on Kubernetes extensible architecture. The result is the Arktos project. Arktos further abstracts Kubernetes concepts for seamless API experience, modifies and re-implements part of its architecture for scalability and performance, and fills the infrastructure capability gaps by adding new CRD (Custom Resource Definition) objects.

Specifically, Arktos 1) expands Kubernetes pod concept to include VM and other runtime environments such as WebAssembly [18]. Now a pod can represent a VM application or WebAssembly application, in addition to regular container app, according to user specification; 2) adds tenant and network objects to "virtualize" Kubernetes so that Arktos becomes a truly multi-tenancy platform. Each pod is deployed within the tenant scope and within a tenant network in order to provide strong isolation among tenants; 3) modifies and partitions Kubernetes API server, etcd [25] and controller manager architecture to scale the platform and performance, and finally 4) Arktos extends CRI (Container Runtime Interface) and implements Arktos runtime to manage lifecycle of not only containerized workloads but also VMs, and WebAsembly workloads in the future.

In summary, Arktos transforms Kubernetes from a single container only orchestration system towards managing unified cloud infrastructure resources across all workload types through one technology stack. In the subsequent sections, we will present Arktos system design and optimizations in detail for addressing the challenges described in this section.

3 SCALABILITY

This section describes the unique scale-out design of Arktos which can scale up to hundreds of thousands of compute nodes in a region and at the same time provide support for the cross-zone high-availability model typically required for building out a large scale cloud infrastructure.

As mentioned in previous section that Arktos is striving towards addressing the scalability and performance challenges posed by the underlying Kubernetes architecture [42]. Kubernetes architecture is mostly still a centric-style cluster management system and has scalability limits at various components level. For example, the API server in Kubernetes is the core component in its architecture and it becomes the main bottleneck for scaling the cluster size. Even if there is an option for multiple active-active API servers, these servers are still designed to be identical, each replica holds a full copy of all API objects for the entire cluster. This, in turn, greatly limits the entire system scalability. In addition, all the cluster data is stored in a single etcd [25] cluster. Its throughput and capacity are limited to a single machine due to the nature of underlying RAFT [32] protocol. The largest number of supported compute nodes per cluster as published by Kubernetes community is 15K [33], which is a far cry from the 100, 000 nodes scale required for a typical hyper-scaler cloud data center.

3.1 Scale Out Architecture

Scale out architecture of Arktos re-envisions the way Kubernetes architectural components are deployed with the goal of scaling much beyond what is currently possible with its current design. The key idea is to split the entire architecture into multiple partitions by tenants and by compute nodes. There is no single point of failure such as "system partition" or "root partition" in the scale-out architecture. Any partition failure would not bring down the entire cluster. Arktos scale-out architecture is as shown in Figure 1.

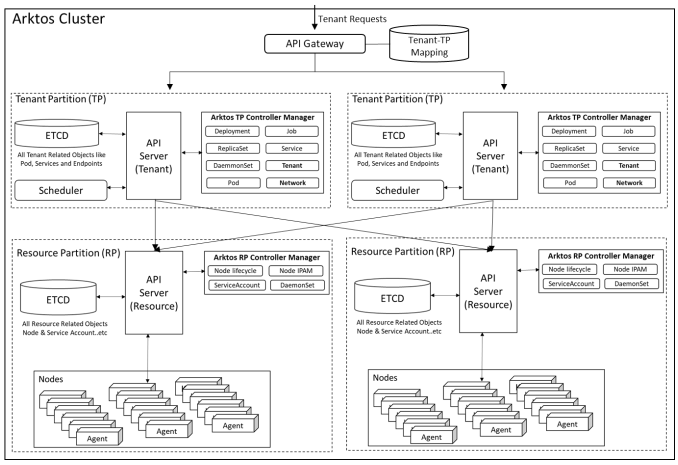

Figure 1: Arktos Scale-Out Architecture

As indicated in the architecture diagram, there are two types of partitions in Arktos: tenant partition (TP) and resource partition (RP). Each manages its own set of objects (workload deployment objects for tenant partition, and node resource objects for resource partition). Tenant partitions and resource partitions can scale independently from each other: 1) If more nodes are needed to host more applications, one can deploy and add more resource partitions; 2) If tenant workloads are high or new tenants are added, one can deploy and add additional tenant partitions to meet the needs; 3) The number of TPs and the number of RPs don't need to match exactly. For example, Arktos can be deployed with a configuration of five tenant partitions (5 TPs) and three resource partitions (3 RPs).

Tenant Partition (TP): A tenant partition handles workload deployment requests from tenant users, such as CRUD pods, deployments, or jobs. It stores all the API resources belonging to the tenants who are assigned to the partition and serves all API requests from these tenants. Tenant assignment process is described in the following sub section. A tenant partition is a sub-cluster itself. As a cluster, it is highly available, in other words, there could be multiple API server instances or controller instances within a tenant partition, based on the capacity and availability requirements. Typically, a tenant partition should be able to support tens or hundreds of thousands of tenants. As shown in the Arktos architecture diagram, each tenant partition contains its own 1) etcd instance; 2) API Server instances; 3) controller manager component that runs most of workload related controllers except node controller; 4) scheduler component that schedules workloads on to nodes. There is a system space in each TP that only stores "tenants" objects. Schedulers on each tenant partition connect to the API Servers of each resource partition and get a full view of all the nodes across all resource partitions.

Resource Partition (RP): A resource partition manages a group of physical or virtual compute nodes. It stores API resources related to these nodes, like node info and node status as well as node leases on these nodes. These resources don't belong to any particular tenant. A resource partition also serves the heartbeat traffic and status update traffic from all these nodes. Similar to a tenant partition, a resource partition also contains its own copies of API server and controller manager that run only node related controllers such as node lifecycle controller, node IPAM controller, and service accounts. A resource partition serves resource queries and updates from all tenant partitions and manages plugins that are deployed per-node. For now, only the node, node-lease, and their dependent resources under system space are supported.

Tenancy Assignment: Tenants and tenant partitions are m:1 model. One tenant belongs to one tenant partition and only one tenant partition. One tenant partition may hold multiple tenants, typical tens of thousands or hundreds of thousands of tenants, depending on the planned partition capacity. For large tenants or so-called VIP tenants, they may have their dedicated tenant partitions. A tenant is assigned to a certain tenant partition during the tenant registration time. The assignment rule is arbitrary and can be based on any policy (naming-based, round robin, load balanced, random, etc). For simplicity purpose we don't support cross-partition tenants or tenant movements in the current version of Arktos. Because a single tenant partition can scale to quite a high level (holds more than 750,000 deployment objects in our test environment), it's not an urgent feature request for now and will be addressed as part of our future work.

API Gateway: API Gateway is a stateless service. Multiple identical gateway instances can be deployed to distribute traffic. There is no limit on how many instances can be deployed. API gateway is employed here only for serving tenant requests and doesn't serve data plane traffic when workloads are running. Internal clients (such as system admins and SREs) can connect to tenant partitions and resource partitions directly without through API gateway. Like most existing API gateways, Arktos API gateway perform functions such as authentication, authorizing, request dispatching, auditing, rate limiting, etc.

Tenant-to-TP Mapping DB: On the API gateway side, a mapping of tenant name and tenant partition ID is stored within an etcd cluster. Assuming the average tenant name length is 64 bytes and tenant ID is represented as a 4-byte int. For 1 million tenants and 1000 partitions, the required storage size is about 1M * (64+4) = 68M. If we use tenant ID (GUID) in the request URL, the map size will be reduced to around 20M (1M * (16+4)). The size of map can easily be fully loaded into the memory for each API gateway instance and maintained as a hash map in the memory. This allows us to do an O(1) hash lookup for each incoming request. Any map entry changes will be synchronized to all the API gateway instances via list/watch mechanism.

3.2 Key Performance Optimizations

As the scalability goal of Arktos is to manage hundreds of thousands of compute nodes in a cluster, our performance tests have shown that the goal cannot be achieved by the scale-out architectural changes alone. This section describes the key performance optimizations in Arktos for boosting a single large cluster that manages from 15K up to 50K nodes.

- *Eliminate Unnecessary Reads* - In a standard Kubernetes architecture, node agent (Kubelet) on each node connects to the API Server to watch any changes made to the objects related to the node. When a node status changes and needs to be updated in API server and etcd data store, Kubernetes reads the node object first from API server and then applies the changes to the object. This causes significant performance and scalability issues when the cluster has hundreds of thousands of nodes. Arktos removes the leading reads (to save time and traffic) and updates a node status directly on the existing object. Only when a conflict is detected, Arktos reads the object from the API server and updates the object accordingly. We observed that this small optimization improves the cluster performance and scalability significantly. In the current architecture, it is very rare that other processes would update the node object, and the chances of conflicts are minimal. The update conflict rarely happened during our performance tests.
  Our tests have shown that with this optimization, Arktos reduces pod startup latency (99 percentile – P99) by 13.5 percent, from average 7.22 seconds to 6.24 seconds per pod in a simulated cluster of 25K nodes with 750K pods deployed. Please refer to Section 6 for more detail of performance results and evaluation. Please also note that a pod startup latency of X seconds at P99 means 99 percent of pod startup latency is less than X seconds.
- *Regulate Client List Operation* - List operation is a common operation from all client components, including node agent, to the API server to get a list of objects desired. We observed that a large number of list operations to API server have significant negative performance impact on the API server and overall system performance and scalability. Arktos regulates the list operation and optimizes many list operations in the system to prevent listing objects one pod at a time. With this optimization, in a simulated test cluster of 25K nodes, we were able to reduce P99 pod startup latency by 17.2 percent from 6.94 seconds to 5 seconds.
- *Watch Mechanism Improvement* - Kubernetes uses watch capability to get latest data changes of an object, instead of frequent and unnecessary query to the system. To prevent from reading too old version of data change events, each watch connection starts from a given object version and listens for any changes after the version. Unfortunately, in current Kubernetes design, watch connections are periodically rebuilt after each latest changes applied. This is a very expensive operation as all nodes in the cluster connect to the API server for watching object changes. Especially, when a node agent does not receive any new event for a relative long period of time, each watch connection from the node will be rebuilt starting from a very old version of object data. This will cause a scanning large number of cached events in API Server repeatedly, which results in a performance and scalability issue to the system. The issue is compounded when there are more than 10K nodes in a cluster.
  Arktos introduces a bookmark event after each large event scanning. The Bookmark event will be sent to clients and trigger clients to update its latest watched object version. Subsequent watch connection then starts from the version of the bookmark event. This optimization significantly reduces the number of events to be scanned upon each watch rebuilt. Our performance test results have shown that with this optimization, the P99 pod startup latency improves by 69.8 percent from 9.89s to 2.98s in a simulated test cluster of 50K nodes with 1.5 million of pods deployed.

Arktos scale out architecture, together with the three key performance optimizations, allows a single Arktos cluster (with two TPs and two RPs setup) to support 50K nodes with high throughput. Again, we will present detailed performance test results and evaluation in section 6.

3.3 Arktos High Availability

With Arktos scale-out architecture, it can be deployed for cloud infrastructure with high availability for customers across multiple availability zones (AZs). Figure 2 below demonstrates one potential Arktos deployment cross three AZs. In this deployment model, each tenant partition is deployed in an active-active manner across three AZs. The model allows each tenant partition to survive one or more AZ failures. On the other hand, there is one resource partition deployed at each of three AZs, and each resource partition manages its own compute nodes in the corresponding AZ. If an AZ goes down, the corresponding resource partition will not serve all the compute nodes in that AZ to three tenant partitions, as it will be

down as well due to the AZ failure. In a summary, Arktos resource partitions are recommended to be deployed at the AZ level, which reduces the communication latency between one resource partition and the nodes the RP manages, and therefore improves the overall heartbeat robustness. It also reduces the cross-AZ traffic cost.

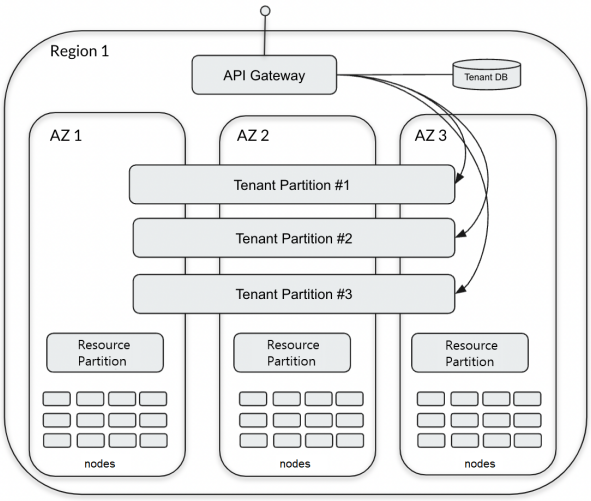

Figure 2: High Availability Architecture

4 UNIFIED RUNTIME INFRASTRUCTURE

While cloud native applications are more suitable for containerized environments, enterprises still have and will continue to have business-critical applications that run on bare metal and traditional VMs for performance and/or complexity reasons [45]. Consequently, cloud providers, as they do today, will have to continue to support various workload types, like container, virtual machine and Serverless, to their customers.

As described in [45, 9], OpenStack and Kubernetes communities are envisioning a unified OpenStack and Kubernetes like environment for managing the heterogeneous application workloads, especially for Telecom service providers. Unfortunately, the effort in OpenStack [3] community has been integrating two platforms together, meaning that you still have two orchestration engines and two set of APIs for managing VM and container workloads. On the other hand, Kubernetes community provides VM support with KubeVirt [1] or Virtlet [5] add-ons, which carries major performance impacts and limits rich VM actions that are offered by OpenStack. Arktos proposes an innovated way to build the unified orchestration infrastructure with two goals in mind:

- Natively support VM and container workloads with same API semantics and same orchestration workflows, and the same runtime agent
- Flexible abstraction for future workload types such as WebAsseembly [18], Uni-kernels [4].

Arktos leverages Kubernetes APIs and orchestration workflows, such as replicaSet, deployment and daemonSet etc., for both VM and Container workloads by further abstracting pod concept to include VM and other types of workloads (in the future). In Arktos, VMs are with their own runtime endpoint. This exposes possibility for VMs unbound from the container networking solutions. Just like VM workloads orchestrated by OpenStack platform, VMs orchestrated using Arktos can leverage new networking capabilities such as direct NIC access etc.

The following subsections present the design and implementation details of the unified orchestration subsystem, including object model, API design, unified scheduling, and multiple runtimes of node agent.

4.1 API Model

Arktos is built around a hierarchical object model, which is at the core of Arktos type subsystem. One design decision is to determine the relationship of container objects and virtual machine objects in this hierarchy. There are two approaches for achieving this, both native approach and add-on approaches were evaluated.

With a native approach, virtual machines and containers are distinct core elements in the system. They are both first-class citizens in the object model and they don't rely on each other at all.

On the contrary, with add-on approach one workload is defined as a variation of another workload, or a plugin. For example, Virtlet [5] defines VM workloads using container pods with a few VM-specific annotations. KubeVirt [1] introduces a new extension object to represent a VM type, side by side with the existing pod object.

In Arktos, the native approach is adopted. The pod specification is extended to represent either containers or virtual machines. Listings 1 and 2 show examples of a virtual machine pod and a container pod.

Listing 1: A VM Pod YAML definition.

```
apiVersion: v1
kind: Pod
metadata:
  name: vm1
spec:
  virtualMachine:
    keyPairName: default.key
    name: vm
    image: ubuntu-18-04.img
    imagePullPolicy: IfNotPresent
    resources:
      limits:
        cpu: "4"
        memory: "8192Mi"
```

Listing 2: A container pod YAML definition

```
apiVersion: v1
kind: Pod
metadata:
  name: pod2
  labels:
    app: vanilla
spec:
  containers:
  - name: vanilla-container
    image: ubuntu
    command: ['sh', '-c', 'echo Hello && sleep 60']
```

The virtual machine specification, just like the container specification, is set at the same level of a pod specification. As a result, all the container workload orchestration objects, like ReplicaSet, Deployment, Job, StatefulSet can be reused for virtual machine

workload seamlessly. User experience with virtual machine workload orchestration is quite similar with what they have with the container workload orchestration. With this approach, Arktos offers orchestration of virtual machine workloads using cloud-native workload composition patterns, as shown in Figure 3 .

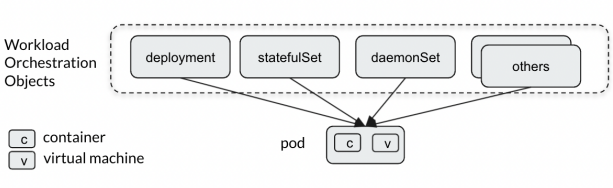

Figure 3: Unified Pod Composition

4.2 Imperative APIs for Richer VM Actions

Like Kubernetes, Arktos uses declarative APIs to interact with users, where users specify the target state and Arktos system keeps reconciling the target state and current state for every pod object. This style has proven to be very effective and resilient for container workloads. However, it doesn't work well for some specific VM life-cycle events, such as reboot, snapshot etc. which are all intrinsically command style operations, which are not suited to be described by state transitions. Even technically doable, the declarative design will require very complex VM action controllers and will end up with a very unmanageable VM pod object in order to support the large number of VM actions Openstack platform offers.

Arktos solves this issue by introducing a new imperative style of APIs called "action APIs". Each action API object includes a target pod and expected actions on the target. Additionally, it also has status field to indicate the result status of the actions. This Action object and API, along with action controllers, handles rich VM action efficiently and flexibly in Arktos.

Listing 3 shows the definition for the action API object which supports the reboot VM life-cycle event as mentioned earlier:

Listing 3: Sample yaml file for rebooting a VM in Arktos

```
apiVersion: v1
kind: Action
metadata:
  name: reboot
spec:
  podAction:
    podName: vm-pod-1
    rebootAction:
      delayInSeconds: 30
```

4.3 Scheduling with Abstraction of Workload Resources

Arktos uses a single scheduler implementation(flow and algorithms) for both container and VM workload(pod). Container and virtual machine specifications differ in many aspects. For example, WorkingDir, Command etc are applicable to containers, but not to VMs. Likewise, fields such as key-pair, initScirpts are applicable to VMs and not to containers. However, from scheduling perspective, it cares about the resource allocation requests, including cpu, memory, volumes, extended resources (such as GPU, FPGA) or other aspects (images etc.) that affect scheduling algorithms. These are all common for both container and virtual machine workloads.

Arktos abstract those scheduling related aspects by implementing a data structure called "CommonInfo". It essentially contains a shadow copy to the resource requests defined in virtual machine or container specifications inside a pod. When containers or a virtual machine is defined in a pod, their resource requirements are copied into this common abstraction. Arktos scheduler reads this common data structure instead of directly reading container objects or virtual machine object in a pod. It does not even need to know whether container or virtual machine workloads are being scheduled. With this approach, most of Kubernetes scheduler code remains unchanged in Arktos.

Listing 4 shows part of the definition of this CommonInfo data structure:

Listing 4: Common fields for VM/container scheduling

```
type CommonInfo struct {
        Name string
        Image string
        Resources ResourceRequirements
        VolumeMounts []VolumeMount
        ImagePullPolicy PullPolicy
        ...
}
```

4.4 Node Agent with Unified Runtime

Together with the Arktos controllers at the control plane, Arktos node agent starts and manages both container and VM workloads at the host in a unified flow.

As shown in Figure 4, Arktos node agent has two layers: unified control layer and interface layer. The former handles the lifecycle of containers and virtual machines in a unified workflow, while the latter adapts to different runtime services, network services and storage services.

In the unified control layer, both container and virtual machine pods share a set of common components as described below:

- Pod State and life cycle Management: This component manages the pod states including when initially a new pod is assigned to the host, subsequent pod state changes (reboot a virtual machine pod, etc), pod configuration changes such as resizing a pod, pod deletion, etc.
- Image Management: This component downloads container images or virtual machine images and manages the image caches.
- Resource control, monitoring, and eviction: Those components monitor, control the resource usage of the container or virtual machine process and evict the workload per the eviction policies. Both container and virtual machine workloads are children of the node agent's control group.
- Volume Management: This component manages the volume plugins and works with CSI [27] interface to mount or unmount a volume.
- Network Management: This component manages the CNI [31] plugins and works with CNI interfaces to manage virtual network devices.

For runtime management, the traditional CRI interface cannot meet our requirements as it doesn't include many required methods

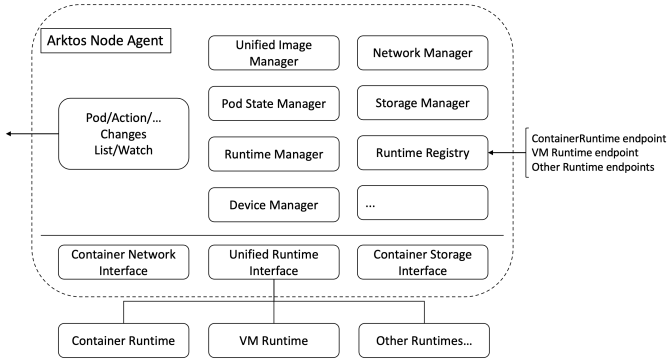

Figure 4: Arktos Node Agent

to support VM life-cycle events, such as rebooting a VM or taking snapshot of a VM.

Kubernetes RuntimeClass [30] matured to beta in Kubernetes release 1.14. It allows cluster admins to provide RuntimeClasses to expose the multiple configurations of the CRI runtime service and the POD (workload) to specify the desired configuration to run the containers specified in the POD. The Kubernetes RuntimeClass has a few limitations: 1) it is limited to the CRI interface itself and cannot support operations such as reboot, snapshots etc. which are required for virtual machines; 2) it is bound to the container creation stack and lacks flexibility to the virtual machine workload improvement, for example, leverage modern NIC technologies for VM performance improvement.

Arktos resolves those limitations with the following implementations, which are illustrated in Figure 4:

- Arktos builds its vm runtime service, called Arktos-vm-runtime which extends CRI interface with rich VM actions, such as reboot, start/stop, snapshots, attach/detach devices.
- Arktos adds a runtime registry in the node agent which allows admin to register new runtime endpoints.
- Arktos node agent registers Arktos-VM-runtime service along with other container runtime endpoint such as Contained.
- Arktos node agent then uses a Unified Runtime Interface (URI) to facilitate both container workloads and virtual machine workloads

With the shared control components and a unified workflow with the URI in Arktos, along with the standard CNI, CSI interfaces to interact with network and storage systems. Arktos users have the same experiences managing their VM based application/services as managing the container based application/services.

5 HARD MULTI-TENANCY

Multi-tenancy is a fundamental requirement for cloud platforms. It enables multiple organizations or teams to safely share a physical resource pool. Multi-tenancy is not only for resource access isolation, but also for resource usage isolation and performance isolation among different tenants.

Unfortunately, there is no inherent multi-tenancy design in Kubernetes and it lacks the capability for strict tenant isolation. Name spaces do provide a certain level of isolation among running pods, but such soft multi-tenancy is not enough for true cloud infrastructure platform for cloud providers. We will describe these challenges in section 5.1.

To meet hard multi-tenancy isolation requirements, Arktos implements a strong multi-tenancy model. The model implementation is based on the idea of "virtual cluster" or "Space", where each tenant thinks they own a cluster in an exclusive way and they are not aware of the existence of other tenants at all. Therefore it's totally transparent to cluster users and users can still use native Kubernetes APIs. This enables users to still use their familiar Kubernetes APIs and tool chains without code changes.

5.1 Challenges

There are several different multi-tenancy models for different scenarios [28]. In general, these models can be categorized as "soft" models or "hard" models on the two ends of a spectrum. Soft models are for scenarios of private cloud or within an organization, where there is a certain level of trust between tenants, and it's appropriate to share some resources among tenants, such as shared storage pools or shared management roles. While hard model is for situations where there is no trust among tenants and hard model provides strong isolation.

Since Arktos aims to support the scenarios of no tenant trust, Arktos multi-tenancy is the strictest hard multi-tenancy model. Under this multi-tenancy mode,

There are a few challenges to implement such a strict multi-tenancy model:

- Isolation: tenants should be strictly isolated from each other. They are not aware of the existence of other tenants at all. Cross-tenant access is allowed only if tenant admins set custom authorization policies.
- Autonomy: tenant admins should be able to perform management duties within their own tenant scope without turning to cluster admins. Such duties includes managing quota, maintaining RBAC roles and so on.
- Manageability: cluster operators should be able to manage tenants and to enforce some common policies across all tenants. For example, deploy a network plugin across all tenants.
- Compatibility: Multi-tenant design should keep API compatibility so that existing tools can continue to work.

5.2 Space, Tenant and API Path

Arktos introduces the following concepts and objects to support hard multi-tenancy.

- Space - a logical concept and each resource object in the cluster resides in one and only one space
- Tenant Object and controller - Tenant object is a CRD object representing the tenant. Tenant Controller is a controller to manage tenants in Arktos. When Arktos admin registers a new tenant, a default space is created for the tenant. All objects created by the tenant using object APIs will be located in the space in etcd store
- System Tenant - Every Arktos cluster has a default tenant created when the cluster is initialized. The default tenant is

called system tenant and all tenant objects themselves are stored in the system tenant.
- Tenant metadata field - All deployment objects, such as pod, now has a new field in their object definition, as shown in Figure 5

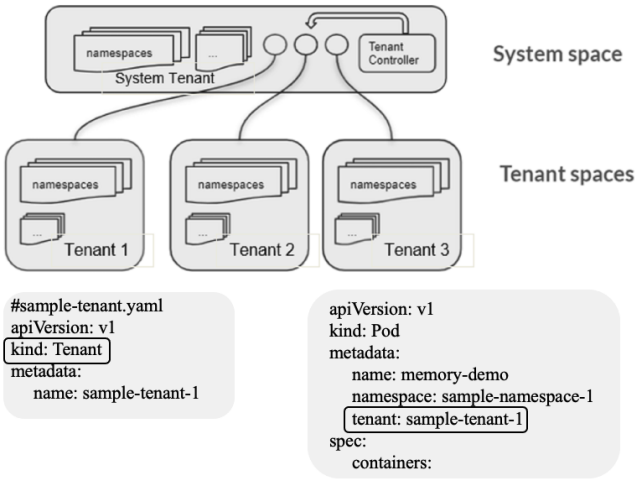

Figure 5: Multi-Tenancy Model - Space and Tenant

As mentioned previously, Arktos leverages rich Kubernetes APIs for seamless user experience. Kubernetes API is a restful API on a collection of API resources. A resource is either namespace scoped (such as a pod) or cluster-scoped (such as a persistent volume). For example, '/api/v1/namespaces/foo/pods/pods/pod1' represents a pod named "pod1" under namespace "foo", since it's a namespace-scoped resource. However, with the introduction of tenant concept, another layer of separation is needed if we want tenants to have their own API resources without worrying about name conflicts. Arktos achieves this by implementing the concept "space" described above. When a tenant is initialized, by default a space with the name of the tenant is created. For example, the following path represents a resources in the space for tenant "tenant" under namespace default: "/api/v1/tenants/tenant1/namespaces/default/pods/pod1".

As we can see, spaces provide an extra layer of resource separation. Within one's space, a tenant can freely create his own resources like namespaces or persistent volumes without worrying about naming conflicts. If an API resource such as node and storage volume is not under any tenant space, then it's in the system space. There is only one system space in the whole cluster, which by default is only accessible to system tenant users. Isolated access to resources in tenant space is governed by various access control mechanisms such as RBAC, which works in a similar way to Kubernetes except that additional tenant credentials are also used for authentication and authorization.

5.3 Network Isolation

Network Model Abstraction: "Network" is the core concept of Arktos multi-tenancy network model, it is a new Custom Resource Definition(CRD) object within Arktos. Resources specific to a particular "network" are IP address, DNS, Pod, Service and Endpoint, Network Policy and Ingress.

Arktos "Network Model" provides an abstraction for integrating with the underlying network virtualization solution by using the appropriate CNI [31] plugin. Arktos integrates with Mizar networking [2] as a default out-of-the-box network virtualization solution. Mizar [2] is a high-performance cloud networking solution for hyper-scale cloud environment powered by eXpress Data Path (XDP) [34] and Geneve [24] protocol.

More importantly, Mizar [2] provides support for strong network multi-tenancy using VPC (Virtual Private Cloud) as a first-class citizen. Mizar [2] uses Geneve [24] as the encapsulation and tunneling protocol for providing layer-2 (L2) level network (hard) isolation at the VPC level.

Inside a VPC, network (soft) isolation is provided using network policies for restricting the network traffic between pods. Developers are able to declaratively specify network policies.

6 PERFORMANCE TEST EVALUATION

As part of the Arktos project, we have done extensive functional and performance tests to verify Arktos architecture and designs. This includes testing of various scale-out architecture settings, VM/-Container hybrid workload deployments, and network isolation between two tenants. For scale-out architecture setting, we tested one-TP/one-RP cluster setup, two-TPs/one-RP and one-TP/two-RPs cluster settings, and finally two-TPs/two-RPs cluster setup for a 50K node cluster. This section mainly presents on the performance test results and analysis of various scale-out test cluster settings.

All scale-out performance tests were conducted on Google Cloud Platform (GCP). All tests were simulated tests using the well-known Kubernetes benchmark tool [26] as we are not able to build a real test cluster environment with 50K compute nodes. The perf test tool includes load tests and density tests that are used to analyze and safeguard each Kubernetes release. Load tests perform functional test while density tests evaluate overall system performance. Density tests consist of two phases: saturation test phase and latency test phase. The former focuses on deploying pods to the entire cluster until it is running at full capacity. The latter deploys small number of pods to test the pod start up latency when the cluster is running at full capacity. Pod startup latency in latency test phase is one of the major criteria to determine whether a cluster is healthy.

Test Environment Setting. On GCP. We used container-optimized OS [17] from Google as the OS for all our tests. The hardware configurations used in the test are as follows:

- Control plane server hardware configuration - We used one machine of type n1-highmem-96 (96 vCPU, 624GB memory, 1000GB disk) for each TP, RP, and admin cluster required for performance test tool. Thus, there are total 3 machine instances needed for setting up one-TP and one-RP Arktos cluster and total of 5 machine instances for setting up two-TPs and two-RPs Arktos cluster.
- Node server hardware configuration - We used machines of type n1-highmem-16 (16 vPCU, 104G memory, 1000GB disk) as the compute nodes for perf test tool admin clusters. We figured out through testing that we needed 505 such machine instances on GCP to simulate 50K nodes in a two-TPs and

two-RPs cluster, and 260 the machine instances to simulate 25K nodes in a one-TP and one-RP cluster.

To reduce the cost of running performance tests, we increased the QPS for the saturation test phase from 20 to 100. This reduced total test time from 22.5 hours to 4.5 hours without noticeable change to pod startup latency in latency test phase.

Comparable Tests. To evaluate performance test results for Arktos, we performed two tests for each test setting. The first test is a baseline benchmark test on a comparable vanilla Kubernetes cluster. The second test is on an Arktos scale-out cluster, so that we can compare and evaluate how well Arktos performs in respect to vanilla Kubernetes. Please note that all tests were done using container pods since the existing test tool does not support VM pods at this time.

Vanilla Kubernetes Test Results The following table (Table 1) shows the performance test results for Kubernetes release 1.18.5 and 1.21.

Table 1: Pod Startup Latency Test Results - Vanilla Kubernetes Cluster

Version	Cluster Size	Saturation Test QPS		Saturation Test Pod Startup Latency (s)	Latency Test QPS		Latency Test Pod Startup Latency (s)
1.18.5	15K	100	p50	1.4548	5	p50	1.3690
			p90	2.1734		p90	1.9476
			p99	3.4212		p99	2.5986
1.21	15K	100	p50	4.0636	5	p50	1,543.1601
			p90	605.9949		p90	2,612.0888
			p99	775.8657		p99	5,645.0563
1.21	20K	100	p50	926.3056	5	p50	1,747.1640
			p90	1,397.0717		p90	3,309.9713
			p99	1,525.6756		p99	9,285.5425

For pod startup performance, as you may observe from Table 1, the test shows:

- Kubernetes v1.18 is actually better than v1.21 for a cluster with 15K nodes. We did not investigate why Kubernetes v1.21 is worse than v1.85 as we wanted to focus on Arktos test results. From the numbers, we conclude that v1.18 supports 15K-nodes cluster, while v1.21 does not due to too high pod startup latency.
- The test of v1.18 with 20K-nodes cluster failed (hence no test numbers shown in the table). The test of v1.21 with 20K-nodes cluster did not fail, however, the pod startup latency was too high to consider successful.

Arktos Scale-out Test Results Table 2 below shows the performance test results for Arktos 1-TP and 1-RP scale-out cluster as well as for 2-TPs and 2-RPs scale-out cluster. As previously mentioned, 1-TP and 1-RP scale-out cluster supports 25,000 nodes, and 2-TPs and 2-RPs scale-out cluster supports 50, 000 nodes [1].

Table 2: Pod Startup Latency Test Results - Arktos Scale-out Cluster

Cluster Configuration	Cluster Size	Saturation Test QPS		Saturation Test Pod Startup Latency (s)	Latency Test QPS		Latency Test Pod Startup Latency (s)
1TP/1RP	25K	100	p50	1.8870	25	p50	1.7987
			p90	2.9966		p90	2.6265
			p99	7.3548		p99	4.9631
2TP/2RP	50K	200 (100 QPS per TP)	p50	2.0716	50 (25 QPS per TP)	p50	1.8307
			p90	5.9373		p90	2.7759
			p99	9.7578		p99	7.3256

As exhibited in table 1 and table 2, we observe from the performance test results that:

- For Arktos scale-out test cluster (2-TPs/2-RPs cluster with 50K nodes), the average pod startup latency at 99 percentile is higher than that of Kubernetes v1.18 cluster with 15K nodes (7.3s vs 2.6s), but it is still within 8 seconds. This means that, for the 2-TPs/2-RPs Arktos cluster, all 1.5 million pods got started within 8 seconds on average. This is really encouraging number considering the cluster having 50,000 nodes (35,000 more nodes than v1.18 cluster) and each node running 30 pods, with total of 1.5 million pods running.
- Similarly, For the 1-TP/1-RP scale-out cluster with 25,000 nodes, the average pod startup latency at 99 percentile for 750,000 pods is less than 5 seconds (4.95s), comparing to the average pod startup latency of 2.6s for Kubernetes 1.18 with 15K nodes.
- All tests (with 15K and 20K nodes) with Kubernetes v1.21 are not comparable (latency numbers are too high) with Arktos scale-out tests. Therefore, no conclusion is made for comparing v1.21 and Arktos clusters.
- The default latency test QPS is 5. Arktos 1-TP/1-RP cluster with 25K nodes was tested with QPS of 25 and Arktos 2-TPs/2-RPs cluster with 50k nodes was tested with QPS of 50. Both yielded good performance test results. While we did not test vanilla Kubernetes with increased latency QPS of 25 or 50, we are very confident, as the test numbers shown, that with multiple TPs architecture, Arktos has much higher throughput than vanilla Kubernetes.

Arktos Optimization Test Results. Arktos scale-out performance test is based on the same test tool with minor changes for adapting to Arktos multi-tenancy modifications. We conducted rigorous performance tests on scale-out architecture for each key performance optimization described in section 3.2 are shown in Figure 6.

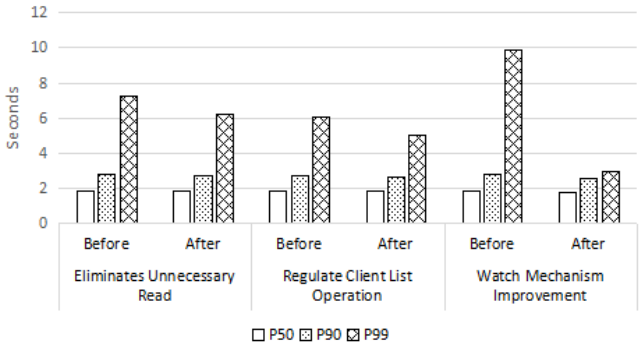

Figure 6: Pod Startup Latency Comparison for Key Performance Optimization

In summary, our extensive performance test results have convinced us that Arktos scale-out architecture is a right design for building large scale cloud infrastructure platform, and we conclude that Arktos 2-TPs/2-RPs cluster with 50,000 nodes passes industrial standard acceptance tests. When the cluster is running at full capacity (1.5 million pods running), we can still create 50K pods with the

[1] 2-TP/2-RP tests were conducted with the reported metrics slightly different between the two TP instances. Table 2 lists the maximal latency from the two TP instances.

speed of 50 pods per second in the cluster without significant pod startup latency increase, until no additional resources are available.

7 RELATED WORK

In this section, we compare Arktos with similar cloud infrastructure platforms, either open-source or proprietary. We strictly focus on the most relevant work in the context of cloud compute infrastructure technology landscape.

Kubernetes [39] is an open-source cloud platform to manage containerized workloads and services. Kubernetes is almost a defacto standard container orchestration system. It was initially developed at Google and was modeled similar to the Google internal proprietary system called "Borg" [59]. As mentioned earlier in section 2 that Kubernetes lacks several key desired capabilities typically required by a hyper-scale infrastructure platform..

Borg [59] is the container management system developed internally at Google. It was originally built to manage both long-running services and batch jobs, which had previously been handled by two other separate systems: Babysitter and the Global Work Queue — both predated Linux control groups and container technology landscape, in general. **Omega** [54] was an offspring of Borg. It was driven by a desire to improve the software engineering of the Borg ecosystem. It applied many of the patterns that had proved successful in Borg but was built from the ground up to have a more consistent, principled architecture.

OpenStack Nova [51] is another open source cloud infrastructure management system. It was originally designed for managing VM workloads but recently support for orchestrating containers has been added as well. However, as described in the background section, Openstack Nova Compute [51] suffers scalability and performance issues .

Twine [55] is a large scale cluster management system internally used by Facebook. However, like Kubernetes, Twine is strictly for containers only, without any VM orchestration support. Twine shares some similarities with Arktos in terms of scalability design. They both adopt the sharding approach within a single cluster instead of federating multiple standalone clusters.

Protean [19] describes the large-scale VM scheduling service deployed in Microsoft Azure data centers. Unfortunately it didn't disclose the entire cluster management system.

Hadoop **YARN** [58] provides great scalability but it's focused on data processing applications, lacking generic container orchestration or VM orchestration capabilities.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we present Arktos, a cloud infrastructure platform for managing a large-scale compute cluster and running millions of application instances as containers and/or virtual machines. We enumerated all the challenges faced in our public cloud services [36], and explained how we addressed these challenges. Unlike the traditional VM-centric infrastructure platform, Arktos is inherently both container and VM native with built-in support for VM operations and managements, via a "unified" resource management and workload orchestration layer as one technology stack. We presented the key innovations in Arktos, including scale-out architecture, unified container/VM scheduling and APIs, and a strong multi-tenancy model with tenant and network objects and controllers. All these core designs make Arktos a large-scale modern cloud infrastructure to seamlessly support modern workloads. Arktos scale-out architecture has been tested and evaluated to demonstrate support for large cluster infrastructure consisting of 50K nodes. Our evaluation also demonstrated that a 50K-nodes Arktos cluster can successfully handle higher QPS than vanilla Kubernetes while the cluster is running at full capacity. In summary, with its scale-out architecture and many key performance optimizations, Arktos can be used to build your next generation cloud infrastructure.

Some of future works include a) *Region-less Distributed Cloud Infrastructure*: As part of our journey towards the vision of next generation "region-less" distributed infrastructure, we plan to expand Arktos architecture to provide support for multiple regions as part of the global cloud platform. b) *Scheduling Algorithm Optimizations*: We plan to build advanced scheduling algorithms and resource allocation mechanism to further increase Arktos scalability and performance. c) *Additional Performance Optimizations*: Improve virtual machine networking performance and remove pod sandbox layer for VM workloads. d) *Geo-distributed metadata store*: As part of "regional-less" vision, we intend to build geo-distributed data store to replace etcd database for further scaling Arktos platform. e) *Performance evaluation for Arktos data plane*: We plan to complete performance evaluation on both control plane and data plane for Arktos, with both VM and container workload types and network integration.

REFERENCES

[1] KubeVirt Authors. 2022. KuberVirt. (May 2022). Retrieved May, 2022 from https://kubevirt.io/.

[2] Mizar Authors. 2022. Mizar. (May 2022). Retrieved May, 2022 from https://github.com/CentaurusInfra/mizar.

[3] OpenStack Authors. 2022. OpenStack. (May 2022). Retrieved May, 2022 from https://www.openstack.org/.

[4] Unikernels Authors. 2022. Unikernels. (May 2022). Retrieved May, 2022 from http://unikernel.org/.

[5] Virtlet Authors. 2022. Virtlet. (May 2022). Retrieved May, 2022 from https://docs.virtlet.cloud/.

[6] AWS. 2022. Amazon elastic kubernetes service (eks). (May 2022). Retrieved May, 2022 from https://aws.amazon.com/eks/.

[7] Microsoft Azure. 2022. Azure kubernetes service (aks). (May 2022). Retrieved May, 2022 from https://azure.microsoft.com/en-us/services/kubernetes-service/.

[8] Centaurus LF Project. 2022. Centaurus Distributed Cloud Infrastructure. (June 2022). Retrieved June, 2022 from https://www.centauruscloud.io/.

[9] Dan Kohn Cheryl Hung. 2019. Cncf telecom user group kickoff. (May 2019). https://kccnceu19.sched.com/event/MSzj/intro-deep-dive-bof-telecom-user-group-and-cloud-native-network-functions-cnf-testbed-taylor-carpenter-vulk-coop-cheryl-hung-dan-kohn-cncf.

[10] Google Cloud. 2020. Bayer crop science seeds the future with 15000-node gke clusters. (June 2020). Retrieved May, 2022 from https://cloud.google.com/blog/products/containers-kubernetes/google-kubernetes-engine-clusters-can-have-up-to-15000-nodes.

[11] Google Cloud. 2022. Google kubernetes engine. (May 2022). Retrieved May, 2022 from https://cloud.google.com/kubernetes-engine.

[12] Google Cloud. 2022. Guidelines for creating scalable clusters. (May 2022). Retrieved May, 2022 from https://cloud.google.com/kubernetes-engine/docs/best-practices/scalability.

[13] cloud.google.com. 2022. Google Cloud. (May 2022). Retrieved May, 2022 from https://cloud.google.com/.

[14] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. 2018. Elasticity in cloud computing: state of the art and research challenges. *IEEE Transactions on Services Computing*, 11, 2, 430–447. DOI: 10.1109/TSC.2017.2711009.

[15] FlexeraTM. 2022. 2022 State of the Cloud Report, Flexeratm. (May 2022). Retrieved May, 2022 from https://info.flexera.com/CM-REPORT-State-of-the-Cloud.

[16] Google. 2022. Cloud architecture center - using clusters for large-scale technical computing in the cloud. (May 2022). Retrieved May, 2022 from https://cloud.google.com/architecture/using-clusters-for-large-scale-technical-computing.

[17] Google. 2022. Container-optimized os documentation. (June 2022). Retrieved June, 2022 from https://cloud.google.com.mcas.ms/container-optimized-os/docs.

[18] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with Webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2017). Association for Computing Machinery, 185–200. ISBN: 9781450349888. https://doi.org/10.1145/3062341.3062363.

[19] Ori Hadary and Luke et al. Marshall. 2020. Protean: Vm Allocation Service at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, (Nov. 2020), 845–861. ISBN: 978-1-939133-19-9. https://www.usenix.org/conference/osdi20/presentation/hadary.

[20] Cynthia Harvey. 2021. Aws vs. Azure vs. Google Cloud: 2022 Cloud Platform Comparison. (Aug. 2021). Retrieved May, 2022 from https://www.datamation.com/cloud/aws-vs-azure-vs-google-cloud/.

[21] https://aws.amazon.com/. 2022. Amazon Web Services (AWS). (May 2022). Retrieved May, 2022 from https://aws.amazon.com/.

[22] https://azure.microsoft.com/. 2022. Microsoft Azure. (May 2022). Retrieved May, 2022 from https://azure.microsoft.com/en-us/.

[23] https://cncf.io. 2022. CNCF. (May 2022). Retrieved May, 2022 from https://cncf.io.

[24] https://datatracker.ietf.org. 2020. Geneve. (Nov. 2020). Retrieved May, 2022 from https://datatracker.ietf.org/doc/html/rfc8926.

[25] https://etcd.io/. 2022. Etcd. (May 2022). Retrieved May, 2022 from https://etcd.io/.

[26] https://github.com/kubernetes/perf-tests. 2022. Kubernetes perf-tests. (June 2022). Retrieved June, 2022 from https://github.com/kubernetes/perf-tests.

[27] https://kubernetes.io/blog. 2019. Container Storage Interface (CSI). (Jan. 2019). Retrieved May, 2022 from https://kubernetes.io/blog/2019/01/15/container-storage-interface-ga/.

[28] https://kubernetes.io/blog. 2021. Kubernetes Multi-tenancy Models. (Apr. 2021). Retrieved May, 2022 from https://kubernetes.io/blog/2021/04/15/three-tenancy-models-for-kubernetes/.

[29] https://kubernetes.io/docs/concepts/architecture. 2022. Container Runtime Interface (CRI). (May 2022). Retrieved May, 2022 from https://kubernetes.io/docs/concepts/architecture/cri/.

[30] https://kubernetes.io/docs/concepts/containers. 2022. Kubernets Runtime Class. (May 2022). Retrieved May, 2022 from https://kubernetes.io/docs/concepts/containers/runtime-class/.

[31] https://kubernetes.io/docs/concepts/extend-kubernetes. 2022. Container Network Interface (CNI). (May 2022). Retrieved May, 2022 from https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/.

[32] https://raft.github.io/. 2022. RAFT. (May 2022). Retrieved May, 2022 from https://raft.github.io/.

[33] https://siliconangle.com. 2020. Google Cloud scales up Kubernetes Engine to 15,000 nodes for Bayer Crop Science. (June 2020). Retrieved May, 2022 from https://siliconangle.com/2020/06/23/google-cloud-scales-kubernetes-engine-15000-nodes-bayer-crop-science/.

[34] https://www.iovisor.org. 2022. XDP. (May 2022). Retrieved May, 2022 from https://www.iovisor.org/technology/xdp.

[35] Lily Hu, Qing Wang, Stephan Hoyer, TJ Lu, and Yi-fan Chen. 2021. Distributed data processing for large-scale simulations on cloud. In *2021 IEEE International Joint EMC/SI/PI and EMC Europe Symposium*. Raleigh, NC, USA. DOI: 10.1109/EMC/SI/PI/EMCEurope52599.2021.9559316.

[36] Huawei. 2022. Huawei Cloud. (May 2022). Retrieved May, 2022 from www.huaweicloud.com.

[37] Inc. IDG Communications. 2020. IDG Cloud Computing Survey. (2020). Retrieved May, 2022 from https://cdn2.hubspot.net/hubfs/1624046/2020%20Cloud%20Computing%20executive%20summary_v2.pdf.

[38] Kubemark LF CNCF Project. 2022. Kubemark performance test tool. (June 2022). Retrieved June, 2022 from https://github.com/kubernetes/community/blob/master/contributors/devel/sig-scalability/kubemark-guide.md.

[39] Kubernetes Authors. 2022. Kubernetes. (May 2022). Retrieved May, 2022 from https://kubernetes.io/.

[40] Kubernetes.io. 2022. Cluster networking. (May 2022). Retrieved May, 2022 from https://kubernetes.io/docs/concepts/cluster-administration/networking/.

[41] Kubernetes.io. 2022. Considerations for large clusters. (May 2022). Retrieved May, 2022 from https://kubernetes.io/docs/setup/best-practices/cluster-large/.

[42] Kubernetes.io. 2022. Kubernetes Components. (May 2022). Retrieved May, 2022 from https://kubernetes.io/docs/concepts/overview/components/.

[43] Ramón Medrano Llamas, Michael Wildpaner, and Sebastian Kirsch. 2019. Designing and operating highly available software systems at scale. (2019). https://research.google/pubs/pub48000/.

[44] Baker Mckenzie. 2021. 2021/2022 Digital Transformation and Cloud Survey: A Wave of Change. (2021). Retrieved May, 2022 from https://www.bakermckenzie.com/-/media/files/insight/publications/2021/12/2021-digital-transformation--cloud-survey--a-wave-of-change.pdf.

[45] Sagar Nangare. 2019. Analysis-of-kubernetes-and-openstack-combination-for-modern-data-centers. (Sept. 2019). https://superuser.openstack.org/articles/analysis-of-kubernetes-and-openstack-combination-for-modern-data-centers/.

[46] U.S. Chamber of Commerce Technology Engagement Center. 2022. Data centers: jobs and opportunities in communities nationwide. (May 2022). Retrieved May, 2022 from https://www.uschamber.com/assets/archived/images/ctec_datacenterrpt_lowres.pdf.

[47] OpenStack. 2021. Deploying the world's largest private openstack cloud with multi-cell supportkanda- tying the room together with akanda. (Jan. 2021). Retrieved May, 2022 from https://www.openstack.org/videos/summits/tokio-2015/deploying-the-worlds-largest-private-openstack-cloud-with-multi-cell-supportkanda-tying-the-room-together-with-akanda.

[48] OpenStack. 2019. Inspur completes the world's largest single-cluster test based on openstack rocky. (Sept. 2019). Retrieved May, 2022 from https://www.openstack.org/news/view/437/inspur-completes-the-worlds-largest-singlecluster-test-based-on-openstack-rocky.

[49] OpenStack. 2022. Lessons learned in deploying large-scale open infrastructure software. (May 2022). Retrieved May, 2022 from https://the-report.cloud/lessons-learned-in-deploying-large-scale-open-infrastructure-software.

[50] OpenStack. 2022. Openstack challenges. (Jan. 2022). Retrieved May, 2022 from https://ubuntu.com/blog/openstack-2022-challenges.

[51] OpenStack Nova Authors. 2022. Openstack Compute (Nova). OpenStack Compute (nova) - nova 24.1.0.dev111 documentation. (May 2022). Retrieved May, 2022 from https://docs.openstack.org/nova/latest/.

[52] Redhat. 2020. Openstack Customer Stories. (2020). Retrieved May, 2022 from https://www.redhat.com/en/about/videos/red-hat-openstack-platform-success-stories.

[53] Redhat. 2020. Scaling red hat openstack platform 16.1 to more than 700 nodes. (Aug. 2020). Retrieved May, 2022 from https://www.redhat.com/en/blog/scaling-red-hat-openstack-platform-161-more-700-nodes.

[54] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, 351–364. http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf.

[55] Chunqiang Tang et al. 2020. Twine: A Unified Cluster Management System for Shared Infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, (Nov. 2020), 787–803. ISBN: 978-1-939133-19-9. https://www.usenix.org/conference/osdi20/presentation/tang.

[56] Toolbox. 2019. Researchers seek to simplify the complex in cloud computing. (Mar. 2019). Retrieved May, 2022 from https://www.microsoft.com/en-us/research/blog/researchers-seek-to-simplify-the-complex-in-cloud-computing/.

[57] Toolbox. 2022. Single cluster vs. multiple clusters: how many should you have in a kubernetes deployment? (Apr. 2022). Retrieved May, 2022 from https://www.toolbox.com/tech/devops/articles/single-or-multiple-clusters-for-kubernetes-deployment/.

[58] Vinod Kumar Vavilapalli et al. 2013. Apache Hadoop Yarn: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (SOCC '13) Article 5. Association for Computing Machinery, 16 pages. ISBN: 9781450324281. https://doi.org/10.1145/2523616.2523633.

[59] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France.

[60] CNCF Webinars. 2020. One large cluster or lots of small ones? pros, cons and when to apply each approach. (July 2020). Retrieved May, 2022 from https://www.cncf.io/online-programs/one-large-cluster-or-lots-of-small-ones-pros-cons-and-when-to-apply-each-approach/.

[61] Renyu Yang and Jie Xu. 2016. Computing at massive scale: scalability and dependability challenges. In *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, 386–397. DOI: 10.1109/SOSE.2016.73.