# i2kit: A Deployment Tool with the Simplicity of Containers and the Security of Virtual Machines [⋆]

Pablo Chico de Guzmán, Felipe Gorostiaga, and César Sánchez

IMDEA Software Institute, Madrid, Spain

**Abstract.** Container virtualization technologies, like Docker, are becoming increasingly popular. Containers provide exceptional developer experience because containers offer lightweight isolation and ease of software distribution. Containers also solve a fundamental code portability problem.
In contrast, container virtualization is basically insecure when compared to virtualization based on hypervisors. Virtual machines are also better integrated with the rest of the cloud ecosystem. Sum it all, virtual machines are more suitable for production environments. However, virtual machines impose a non-negligible memory footprint and suffer longer boot times, which is impractical for local development. So far, there is no deployment infrastructure that allows both the developer experience of containers and the maturity and isolation capabilities of virtual machines.
We solve this problem in this paper by introducing *i2kit*, an orchestration tool that enjoys the best of both worlds: (1) the development workflow is untouched, containers can be used as usual; (2) at time of deployment, containers are transformed into virtual machines, keeping code portability, but providing better security and better integration with other cloud services. The tool *i2kit* creates virtual machines using Linuxkit. Linuxkit alleviates the drawback in size that using virtual machines would otherwise entail because the footprint of our Linuxkit distributions is only about 60MB. The attack surface of the application is reduced since Linuxkit only installs the minimum set of OS dependencies to run containers. Finally, we report an empirical study using *i2kit* that allows us to conclude that *i2kit* is a promising technology for VM deployment of applications developed using containers.

**Keywords:** virtualization, orchestration; security; resource utilization;

## 1 Introduction

Docker containers [1] have popularized the use of lightweight virtualization technologies such as LXC [2]. Some large companies report running all of their ser-

[⋆] This research has been partially supported by: the EU H2020 project Elastest (num. 731535), by the Spanish MINECO Project "RISCO (TIN2015-71819-P)" and by the EU ICT COST Action IC1402 ARVI (*Runtime Verification beyond Monitoring*).

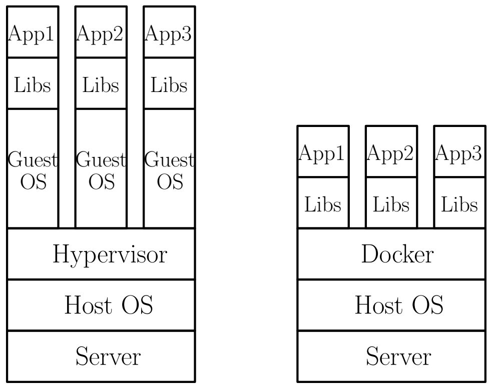

**Fig. 1.** Virtual Machines vs. Containers.

vices in containers (e.g. [3]), and Container as a Service (CaaS) products are available from the main cloud players including Amazon EC2 Container Service, Azure Container Service, and Google Container Engine Service.

There are good reasons for the popularity of containers: containers provide extremely fast instantiation times, small per-instance memory footprints, high density on a single host and ease of software distribution. Fig. 1 illustrates the differences between virtual machines and containers. Containers are lightweight because the operating system layer is not replicated for every application running on the same server. Developers are able to run third-party dependencies such as databases, message brokers, proxies,. . . each in its own container. Additionally, everything is easily integrated with the application under development with enough isolation and density of containers to run many small services in the developer's local machine. In fact, containers have popularized the so-called micro-service architectures [4, 5].

At deployment time, containers solve a fundamental code portability problem. Containers are packaged with all the dependencies and libraries they need to run, making them portable between distributions. However, although the high density of containers is of great value in a *local environment* or for continuous integration (CI) jobs, it introduces new challenges in *production environments*. First, containers are poorly integrated with the rest of the cloud offering, such as auto-scalability, fault tolerance, load balancing, service discovery or networking. Container cluster management tools—like Kubernetes [6], Docker Swarm [7] and Mesos [8]—provide similar services at the cost of adding a new control plane layer, which requires additional setup steps, adds redundancy and might become hard to debug. [9] extends on the complexity of Kubernetes.

But the main challenge introduced by containers is security [10]. Container isolation is based on concepts like namespaces, cgroups, seccomp technologies, the user core Linux permission model or root user capabilities. These mechanisms provide an additional defense on top of application security, but it only takes a single kernel bug to bypass all these mechanisms and escape the container isolation model (see [11] for some vulnerabilities). Some use cases require a higher level of isolation, like sandboxes for running vulnerable or untrusted code, or multi-tenant environments in the case of hosted services. Note that trusted but vulnerable applications running on the same server might become an entry point for malicious agents.

Hypervisors, such as KVM [12], VMware ESXi [13], or Microsoft Hyper-V [14] are proven and mature technology that solved this problem years ago. Following the Linux philosophy of *Do one thing and do it well*, we propose a separation of concerns to provide the security of virtual machines, but the portability and simplicity of containers. In our approach, each container[1] is deployed in its own virtual machine. This container virtual machine (CM) is only meant to provide isolation between containers, and can be reduced to a minimum footprint. The current implementation of *i2kit* is based on Linuxkit [15], which is able to generate Linux distributions specialized to run containers with a memory footprint of approximately 60MB. Smaller distributions also entail:

(a) the reduced attack surface of the system by having less software pre-installed;
(b) faster booting times since booting times are roughly linear in the size of the distribution.

Note that since CMs run containers, code portability and ease of software distribution is maintained.

CMs is the main concept behind *i2kit*, a container orchestrator introduced in this paper which uses the CM as the unit of deployment. The name *i2kit* stands for *immutable infrastructure kit.* Immutable infrastructure [16], also known as *i2*, is an approach to managing software deployments wherein the servers (where components run) are replaced rather than changed or modified in every software update. The tool *i2kit* recreates every CM on every deployment, following a pure *immutable infrastructure* approach. The *i2kit* orchestrator is inspired by Kubernetes, where applications are defined in a declarative way using a YAML Manifest File. The tool *i2kit* provides out of the box solutions for auto scalability, fault tolerance, load balancers, service discovery, rolling upgrades or networking, but instead of reimplementing these services (as Kubernetes does), *i2kit* reuses proven, mature and efficient cloud technology like auto scalability groups, load balancers or DNS services. In a nutshell, compared to containers, CMs provide better security and integration with the rest of the cloud offering.

The rest of the paper is organized as follows: Section 2 introduces the *i2kit* principles and explains how to integrate CMs with the rest of the cloud offering to provide common features available in other orchestrators like Kubernetes. Section 3 describes how CMs are generated from containers. Section 4 describes the implementation of the *i2kit* orchestrator. Section 5 measures the impact of *i2kit* on different metrics such as booting times, networking and cluster memory consumption. Finally, Section 6 concludes and describes some research lines for future work.

[1] In this paper, we refer to containers or pods indistinctly. A pod is a group of strongly related containers that get deployed as a unit.

## 2 The Design of *i2kit*

The *i2kit* tool is open source, and it is actively under development at the IMDEA Software Institute[2]. The *i2kit* tool is a container orchestrator whose unit of deployment is the container virtual machine (CM). CMs are built using Linuxkit (see Section 3 for a detailed description). *i2kit* is inspired by Kubernetes and follows the best principles in the container ecosystem. Kubernetes is an evolution of the Borg [17] and Omega [18] cluster manager tools, adapted for containers, where applications are defined using a declarative model.

For example, Fig. 2 shows a simplified Kubernetes Deployment Manifest input (which we borrow as the input format for *i2kit*), and Fig. 3 represents the deployment of this Deployment Manifest in Kubernetes. Pods are the unit of deployment in Kubernetes, which are essentially a sandbox that allows running containers inside. Informally, a Pod fences an area of the host OS, builds a network stack, creates the necessary kernel name-spaces, and runs one or more containers. A Replica Set builds on top of a set of Pods. A Replica Set takes a Deployment Manifest and instantiates the desired number of replicas of the Pod. Replica Sets also instantiate a background reconciliation loop that ensures that the right number of replicas are always running, forcing the reconciliation between the desired state and the current state. The K-Proxy is responsible for forwarding traffic between Pods, providing load balancing capabilities, based on the ports defined in the Deployment Manifest. Finally, the Service creates a reliable endpoint based on the Deployment Manifest `name` field resolving to the running Pods.

```
name: myapp
replicas: 3
containers:
  nginx:
    image: nginx:1.7.9
    ports:
    - 80
  api:
    image: myapp:1.0
```

**Fig. 2.** *i2kit* Manifest File.

[2] *i2kit* is available at `www.github.com/pchico83/i2kit`.

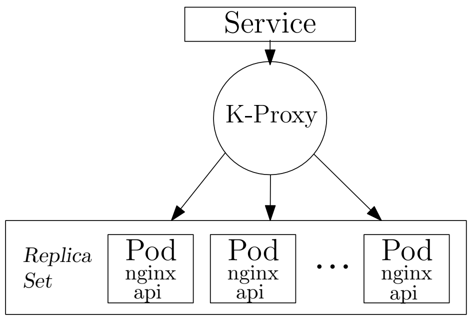

**Fig. 3.** A high-Level view of a Kubernetes Deployment.

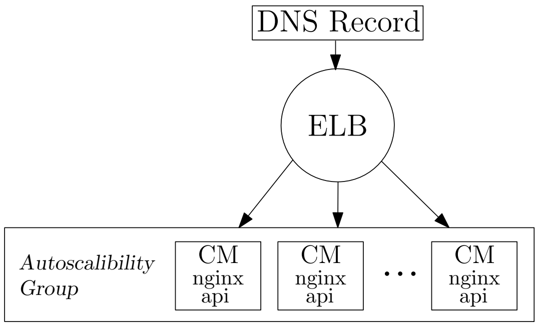

**Fig. 4.** A high-Level view of a *i2kit* Deployment.

The architecture of *i2kit* is similar than Kubernetes, but *i2kit* replaces the Pod by the CM as the unit of deployment. The current implementation of *i2kit* is integrated with Amazon Web Services, although support for other cloud vendors is under development, and provides similar concepts than replica sets, k-proxy, and service by integrating the deployment of CMs directly with the rest of the cloud vendor offering. Fig. 4 shows how the Deployment Manifest presented in Fig. 2 looks like in *i2kit*. The Autoscalabity Group [19] is the equivalent to Replica Sets. It ensures that three CMs are always running, recreating CMs if they become unreachable. The Elastic Load Balancer (ELB) [20] is responsible for forwarding traffic between CMs, providing load balancing capabilities. The DNS Record creates a reliable endpoint based on the Deployment Manifest `name` field resolving to the running CMs for service discovery capabilities. In a nutshell, *i2kit* mimics the Kubernetes architecture but using proven cloud vendor technology thanks to the better integration that virtual machines have compared to containers. We explain these concepts in the next subsections.

**Declarative Model** A declarative model of *i2kit* works as follows:

– The user declares the desired state of his deployment in, for example, a YAML Deployment Manifest that includes a description of which container images to run, the ports exposed by each container, the number of replicas, the commands to execute, or how to instantiate environment variables.
– The orchestrator issues workloads to run the deployment in the cloud, in a manner that is completely transparent to the user.
– The orchestrator also watches the deployment state in order to restore the desired state in the event of a failure. For example, if a CM becomes unreachable for whatever reason, it is replaced by a new CM running the same containers.

There is a significant difference between the declarative approach described above and an imperative language to describe control planes. In an imperative model, the user issues a procedure with specific commands to reach the desired state. A declarative description is usually much shorter and simpler than a long sequence of imperative commands and describes the details of how to create and coordinate the different resources.

Fig. 2 shows an example of an *i2kit* Deployment Manifest. Note that the *i2kit* declarative model supports the execution of several containers on the same CM, as Pods do. There are advanced use-cases that justify the run of multiple containers inside a single CM, for example:

- a log scraper tailing the output of the user container to a centralized logging service.
- a stats collector sending metrics to perform analytics.
- a sidecar container providing features for the user container.

**Failure Tolerance and Autoscalability** In Kubernetes a Replica Set ensures that a fixed number of Pod instances are always running. Replica Sets also replace Pods that get unreachable. Not surprisingly, in the realm of virtual machines, there are solutions that perfectly map this behavior under the assumption of using the CM as the unit of deployment. Amazon Web Services offers Amazon Auto Scalability Groups (similar services exists in the rest of cloud vendors). AWS Auto Scalability Groups help to maintain the health and availability of a fleet of Amazon virtual machines, ensuring that the desired number of CMs is always running. If a CM becomes unhealthy, it gets replaced by a new CM running the same container versions. In addition, Auto Scalability Groups can be used to scale the application up and down under certain situations, such as high CPU or memory usage, or an increment in incoming requests. This allows very flexible deployments only consuming resources on demand, without the need of guessing infrastructure capacity in advance.

Note that in the event of a rolling update, new CMs are generated, and the Auto Scalability Group replaces every existing virtual machine by the new ones. The tool *i2kit* does follow a pure immutable infrastructure approach by design. In contrast, Kubernetes reuses the same cluster between deployments. Kubernetes Nodes might leak memory and become unreachable after a number of deployments. For example, it is very common that Kubernetes Nodes become unhealthy due to the lack of storage resources, for example by the garbage accumulated by old docker images from previous deployments. In contrast, following a pure immutable approach like *i2kit* brings important advantages. For example, in the event of a full system failure, applications can be recreated in a different availability zone with a single deploy command on the Deployment Manifest. Guessing the state of a Kubernetes cluster or recreating this state is more difficult and time-consuming.

**Load Balancing** CMs are mortal and, in practice, it is not unusual that a given CM becomes unreachable due to networking issues or other software or hardware failures. On failure, Autoscalability Group replaces the dying CM with a new one, which probably is assigned a different IP. Moreover, when performing rolling updates the new CMs have different IPs than the old ones. Therefore, the application logic cannot rely on CM IPs.

The solution to this problem is the use of load balancers, which provide a reliable networking endpoint for a set of CMs. Amazon Web Services offers Elastic Load Balancers (similar services exists in the rest of cloud vendors). Elastic Load Balancers, not only provides a reliable networking endpoint for a set of CMs but also, as their name suggests, balance the incoming traffic between these CMs. Complex policies can be defined to customize how the traffic load is balanced. The port configuration of the AWS Load Balancer is created based on the information contained in the *i2kit* Deployment Manifest.

**Service Discovery** The reliable networking endpoint provided by Elastic Load Balancers is not configurable, and it is randomly created at deployment time. This is not compatible with the idea of providing a Service Discovery mechanism based on the `name` property of Deployment Manifests.

Our solution is to use Amazon Web Services Route 53, which provides DNS as a service (similar services exists in the rest of cloud vendors). The approach of *i2kit* is to create a Route 53 Domain CNAME entry that resolves the *name* field of the Deployment Manifest File to the AWS Load Balancer endpoint that is proxying incoming traffic between the different CMs. In this manner, *i2kit* can provide the same service discovery mechanism—based on names—as Kubernetes.

## 3 Container Machine Generation

The main drawback of container virtual machines (CM) is a loss of performance because in principle a virtual machine imposes a non-negligible overhead on infrastructure resources compared to a container or Pod. However, there are tools that allow creating minimal Linux distributions specifically crafted to run containers. The current footprint of these distributions can get as small as 60MB, a size comparable to container technology. The tool *i2kit* is built on the assumption that the overhead of running a container per virtual machine is acceptable. Moreover, it is to be expected that this figure will keep improving as leaner Linux distributions are developed (this is discussed in detail in Section 6).

The current implementation of *i2kit* uses Linuxkit [15], a toolkit for building custom minimal, immutable Linux distributions. [3] Linuxkit reads YAML templates that describe how to build a Linux distribution. The container information presented in an *i2kit* Deployment Manifest is transformed by *i2kit* into a Linuxkit template, in order to generate a minimal Linux distribution specialized in running these deployment containers. The result is shown in Fig. 5. From every container in the *i2kit* Deployment Manifest, *i2kit* extracts the container relevant information (such as container image, run command, environment variables) and adds an entry in the `services` section of the Linuxkit template.

In our example, this information is:

[3] We are also exploring how to support alternative technologies to Linuxkit.

```
kernel:
  image: linuxkit/kernel:4.9.63
  cmdline: "console=tty0"
init:
  - linuxkit/init
  - linuxkit/runc
  - linuxkit/containerd
  - linuxkit/ca-certificates
onboot:
  - name: sysctl
    image: linuxkit/sysctl
  - name: rngd1
    image: linuxkit/rngd
    command: ["/sbin/rngd", "-1"]
  - name: dhcpcd
    image: linuxkit/dhcpcd
  - name: metadata
    image: linuxkit/metadata
services:
  - name: getty
    image: linuxkit/getty
    env: [INSECURE=true]
  - name: sshd
    image: linuxkit/sshd
  - name: nginx
    image: nginx:1.7.9
    capabilities: [all]
  - name: api
    image: myapp:1.0
    capabilities: [all]
trust:
  org: [linuxkit, library]
```

**Fig. 5.** Linuxkit template.

```
services:
   - name: nginx
     image: nginx:1.7.9
     capabilities: [all]
   - name: api
     image: myapp:1.0
     capabilities: [all]
```

Note that the value `all` is used for the capabilities of the user containers, which is a limitation of the current *i2kit* implementation. Future work includes equipping *i2kit* with an analysis that limits the capabilities associated with every container.

The remaining fields in the Linuxkit template are pre-generated and are identical for every deployment. The filesystem of every custom distribution is currently initialized from the docker image *linuxkit/kernel:4.9.63*. Also, every custom distribution installs the *init* process, *runc*, and *containerd* to be able to run containers, and *ca-certificates* to be able to manage certificates. At boot-time, the following containers are executed in sequence order: *sysctl*, *rngb*, *dhcpcd* and *metadata*. These are basic services required by any software application. Note that *metadata* is installed to be able to manage Amazon Metadata from the CM itself (in this case, supporting other cloud vendors would require to changes). Then, the containers in the `services` section run as daemons in parallel, in particular *getty*, *sshd* and the containers defined in the *i2kit* Deployment Manifest. Finally, *i2kit* uses content-trust-delivery for images coming from the *linuxkit* and the *library* organizations.

Once the Linuxkit template has been generated, *i2kit* builds the minimal Linux distribution and uploads it as an Amazon Machine Image, which is then available to be consumed by the *i2kit* orchestrator. The next section explains the implementation of the *i2kit* orchestrator.

## 4 The *i2kit* Orchestrator

The current *i2kit* implementation supports deployments in Amazon Web Services, but support for other cloud vendor is work in progress. The Amazon Web Services driver makes use of the AWS Cloud Formation Service [21] to create the different resources. Cloud Formation receives JSON manifest files to create and manage a collection of related AWS resources, provisioning and updating them in an ordered and predictable fashion. Cloud Formation templates can specify rolling updates policies to be applied when the template is modified, allowing the simulation of Kubernetes rolling updates.

The *i2kit* orchestrator transforms *i2kit* Deployment Manifest into Cloud Formation templates once the Container Machine Image has been generated using LinuxKit. For example, assume the Amazon Image *ami-XXXXX* has been generated from the Deployment Manifest in Fig. 2 following the process explained in Section 3. Then, a simplified version of the Cloud Formation generated by *i2kit* is shown in Fig. 6.

The Cloud Formation template defines four different resources: `LaunchConfig`, `ASG`, `ELB`, and `DNSRecord`. The resource `LaunchConfig` defines how virtual machines will be created. In our case, each CM will run the AMI created by the process explained in Section 3. The next resource is `ASG`, an Auto Scalability Group which use the `LaunchConfigurationName` created above in order to create CMs. The minimum and the maximum number of CMs matches the number of replicas in the *i2kit* Deployment Manifest. Every CM generated by the Auto Scalability Group is associated with an Elastic Load Balancer defined also in the Cloud Formation template. `ELB` stands for the Elastic Load Balancer that takes the name from the Deployment Manifest `name` field. The `Listeners` information matches the `ports` section of the *i2kit* Deployment Manifest, where the

```
AWSTemplateFormatVersion: 2010-09-09
Resources:
  LaunchConfig:
    Type: AWS::AutoScaling::LaunchConfiguration
    Properties: { ImageId: ami-XXXXX }
  ASG:
    Type: AWS::AutoScaling::AutoScalingGroup
    Properties:
      LaunchConfigurationName:
        Ref: LaunchConfig
      MaxSize: 3
      MinSize: 3
      LoadBalancerNames: { Ref: ELB }
  ELB:
    Type: AWS::ElasticLoadBalancing::LoadBalancer
    Properties:
      LoadBalancerName: myapp
      Listeners:
        LoadBalancerPort: 80
        InstancePort: 80
        Protocol: HTTP
  DNSRecord:
    Type: AWS::Route53::RecordSet
    Properties:
      HostedZoneName: i2kit.com
      Name: myapp.i2kit.com
      ResourceRecords:
      -  Fn::GetAtt:("ELB", "DNSName")
      Type: CNAME
```

**Fig. 6.** Cloud Formation template for the *i2kit* Manifest File.

`Protocol` is inferred from the port number. Finally, the `DNSRecord` resource is a CNAME entry for the Route 53 Domain *i2kit.com*. This domain is received as a parameter of the *i2kit* tool. The CNAME entry is created based on the *i2kit* Deployment Manifest `name` field. It resolves to the `ELB` endpoint, providing service discovery for other deployments. Note that every CM deployed by *i2kit* adds a `DNS_SEARCH` entry in every container pointing to *i2kit.com*. This way, containers will resolve the name of any *i2kit* deployment without the need of specifying the full domain name (ending in *i2kit.com*).

## 5 Empirical Evaluation

This section compares *i2kit* versus the native Kubernetes implementation based on three different metrics: booting times, memory consumption and network performance.

| Pods | 1 | 10 | 20 | 30 | 40 |
|---|---|---|---|---|---|
| *i2kit* | 78 MB | 0.78 GB | 1.56 GB | 2.34 GB | 3.12 GB |
| K8 | 1.94 GB | 2.09 GB | 2.27 GB | 2.44 GB | 2.62 GB |

**Table 1.** Total memory footprint of *i2kit* vs Kubernetes running the example in Fig 2.

**Booting Times** The creation of a virtual machine running *i2kit* Linuxkit distributions in AWS takes about one minute while creating a Pod in Kubernetes takes only seconds (depending on the size and the local availability of docker images). Even though this booting time difference can be very relevant in local environments, it is less relevant in production environments. For example, it is a common practice to introduce a delay of at least 30 seconds between Pod creations during a rolling update, in order for load balancers to have enough time to be updated. Using this common practice induces a comparable delay to the time required to create *i2kit* CMs. Therefore, even though *i2kit* is slower than Kubernetes in terms of booting times, we argue that difference is not very relevant in production environments.

Note also that though we use AWS for deploying the Virtual Machines, these are not the "usual" virtual machines but much smaller specialized machines generated using Linuxkit.

**Memory Consumption** The overhead that *i2kit* imposes for every Pod creation is a consequence of the overhead of the CM running the Linuxkit distribution. It is linear on the number of Pod instances. In contrast, the Kubernetes overhead for running a Pod is due to the overhead of running the Worker Node components, which are shared by several Pods. It is constant on the number of Pod instances.

Table 1 displays the memory comparison between *i2kit* and Kubernetes for running the Deployment Manifest shown in Fig. 2 using different numbers of replicas. Table 1 shows that for a low number of replicas, *i2kit* requires significantly less memory than Kubernetes. Even though the growth in the memory required is faster for *i2kit* than for Kubernetes, *i2kit* is more memory efficient when running less than (approx.) 30 containers in the same Worker Node. Note that the Kubernetes web page [22] does not recommend running more than 30 Pods per Worker Node. Therefore, we can conclude that the memory consumption of *i2kit* behaves better than that of Kubernetes for standard workloads. In fact, we were not able to create with Kubernetes more than 42 Pods on the same Worker Node running on a *t2.xlarge* AWS EC2 Instance. Moreover, the data reported in Table 1 does not take into account the memory consumption of Master Nodes or the Kubernetes Distribute Storage Layer, which would report an even more favorable comparison to *i2kit*.

Finally, sharing a host between several containers imposes performance side effects on the containers running on the same host. Although some research has been done in this area, [23, 24], it is a more mature approach to have resources affecting performance isolated at the hypervisor level.

| Pods | 1 | 5 | 25 |
|---|---|---|---|
| *i2kit* | 129.86 Mbps | 128.191 Mbps | 128.58 Mbps |
| K8-1 | 129.17 Mbps | 25.92 Mbps | - |
| K8-5 | 108.44 Mbps | 108.36 Mbps | 21.73 Mbps |
| K8-25 | 97.95 Mbps | 98.11 Mbps | 97.84 Mbps |

**Table 2.** Network of *i2kit* vs Kubernetes.

**Network Performance** Table 2 shows the network performance comparison between *i2kit* and different Kubernetes cluster sizes. The experiment uses *iperf2* to measure the average network bandwidth consumed by each replica, where each replica runs an *iperf2* server. On the other hand, the *i2kit* configuration runs every *iperf2* server in its own CM using a *t2.large* AWS EC2 Instance. In the table, *K8-N* stands for a Kubernetes cluster with *N* Worker Nodes, where every Worker Node runs on a *t2.large* AWS EC2 Instance. In order to be able to measure the consumed bandwidth, every experiment runs a large amount of *iperf2* clients, where each client runs on its own VM. These clients first send traffic to warm the load balancers up and then synchronize to sending traffic at the same time for 3 minutes.

Table 2 indicates that *i2kit* scales linearly on the number of replicas, as expected. The network overhead of using an AWS Load Balancer is negligible. Note that the limit of the virtual machine incoming traffic is 130 Mbps. The row *K8-1* in Table 2 shows that the overhead imposed by Kubernetes when running on a single node is not very relevant (less than 2%). Since *K8-1* runs all Pod replicas on the same server, running more than one Pod replica quickly hits the VM incoming bandwidth limit, distorting the experiment for the case of 5 replicas. Moreover, we were not able to successfully run 25 Pods on a single Worker Node. The row *K8-5* shows that Kubernetes imposes an overhead of about 20% when the Kube-Proxy needs to forward traffic between five different Worker Nodes. As expected, the overhead grows with the cluster size, as we can see in the *K8-25* row, which accounts for a 30% network overhead. *K8-5* also shows how the traffic is dramatically affected by the virtual machine incoming bandwidth limit when running 25 Pod replicas.

This experiment exposes some side effects of running the additional control plane of Kubernetes on top of cloud vendor technology. In this case, the functionality provided by the K-Proxy is redundant as it is already provided by the AWS Load Balancers, and consequently this additional control plan imposes unneeded performance overheads. This experiment also illustrates that containers are poorly integrated with the rest of the cloud offering. For example, running more than one Pod per VM hits the VM incoming traffic limit. This issue does not happen in *i2kit*—which runs each replica in its own virtual machine—because virtual machines are better integrated with the networking capabilities of the cloud vendor.

## 6 Conclusions and Future Work

This paper has presented *i2kit*, a deployment tool that pursues the following main goals: (1) to preserve the docker development workflow untouched, as containers are a great fit for local environments; (2) to transform containers into lightweight virtual machines upon deployment for better isolation; (3) to provide fault tolerance, load balancing or service discovery without reimplementing these features in a new layer thanks to the better integration of virtual machines with the rest of the cloud offering.

The tool *i2kit* follows the Linux principle of "*Do one thing and Do it well.*" The responsibility of virtual machines is to provide workload isolation and security. The responsibility of containers is to offer portability and ease of container image distribution. The drawback of using virtual machines for container isolation is higher resource utilization, but *i2kit* is specifically designed to exploit Linxkit to generate virtual machines with low memory footprint. The results in Section 5 suggest that the memory consumption of *i2kit* is better than the one of Kubernetes for standard workloads.

Note also that there is a very active research effort targeting VM optimization which *i2kit* can potentially leverage in terms of memory usage to get even better results. Kata Containers [25] fulfills similar goals than *i2kit* in terms of security, building lightweight virtual machines that feel like containers but provide the isolation level of virtual machines. The main difference is that Kata Containers are conceived to be a container runtime [26] instead of integrating with the rest of the cloud offering. LightVM [27] is a new virtualization solution based on Xen that is optimized to offer fast boot-times regardless of the number of active VMs. LightVM features a complete redesign of Xen's control plane reducing the hypervisor to a minimum. LightVM can boot a VM in 2.3ms, comparable to fork/exec on Linux (1ms), and two orders of magnitude faster than Docker. LightVM can pack thousands of LightVM guests on modest hardware with memory and CPU usage comparable to that of processes. The current *i2kit* implementation uses Linuxkit instead of LightVM because it is easily integrable with the AWS cloud offering. LightVM is based on Unikernels [28], which are also very promising on this area. Exploiting VM optimizations and Unikernels to improve *i2kit* further is a line of current and future work. Finally, another research line is to analyze synergies between *i2kit* and serverless architectures [29] provided by cloud vendors.

Section 5 also shows that the Kubernetes control plane introduces redundancies that can affect network performance (and probably other metrics). As we show with *i2kit* in this paper, containers can be integrated with the rest of the cloud offering without the need of adding a complex control plane for container orchestration. Also, Kubernetes is not a cloud native technology. First, Kubernetes requires a Distributed and Reliable Store Cluster. The most common solution to this end is *etcd* [30], a Key-Value Store based on the Raft [31] protocol. Kubernetes also requires a cluster of Master Nodes. Master Nodes execute three different components: Api, Scheduler, and Replication Controller. Also, every Worker Node requires the Kubelet (responsible of executing the tasks assigned by the Scheduler) and the Kube-Proxy (responsible for service discovery and load balancing in a high-density container environment). Even further, users need to take into account that the components in the Kubernetes Control Plane are a runtime dependency for the applications. An error in the Kubernetes Control Plane is not only difficult to debug, but it also disturbs running applications by affecting, for example, service discovery. Managing a large cluster infrastructure and optimizing the scheduling of containers all backed by a complex distributed state store is counter to the premise of the cloud. Cloud vendors let users utilize resources as they go, without guessing capacity, and providing deep operational control without operational burden. The tool *i2kit* allows developers to write their code and have it run, without having to worry about configuring complex management tools. As a result, *i2kit* turns containers into a secure and cloud native technology.

Cloud native containers is also the goal of AWS Fargate. [32] Some differences are: (1) the *i2kit* declarative model is cleaner and allows the execution of several containers per VM, which is very valuable in advanced uses cases like sidecars or log/stats collectors.; (2) *i2kit* future work conceives the option to install OS dependencies on the Linuxkit distribution. For example, tools like Sysdig [33] needs to be installed as a kernel module. (3) *i2kit* is open source and more flexible than AWS Fargate on controlling the runtime technology. Multi cloud vendor support is currently implemented for *i2kit*.

## References

1. D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, Mar. 2014.
2. C. Wang, "LXC and docker explained," http://www.infoworld.com/article/3072929/linux/containers-101-linux-containers-and-docker-explained.html.
3. J. Clark, "EVERYTHING at google runs in a container," http://www.theregister.co.uk/2014/05/23/google_containerization_two_billion/.
4. J. Lewis and M. Fowler, "Microservices: a definition of this new architectural term," http://martinfowler.com/articles/microservices.html.
5. J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 113–116, 2015.
6. B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Commun. ACM*, vol. 59, no. 5, pp. 50–57, Apr. 2016.
7. *Docker Swarm*, https://github.com/docker/swarm.
8. B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. of NSDI'11.* USENIX Assoc., 2011, pp. 295–308.
9. J. Moiron, *IsK8s Too Complicated?*, http://jmoiron.net/blog/is-k8s-too-complicated/.
10. A. Mouat, "Five security concers when using docker," https://www.oreilly.com/ideas/five-security-concerns-when-using-docker.
11. *Linux Kernel Security Vulnerabilities*, https://www.cvedetails.com/vulnerability-list.php.
12. I. Habib, "Virtualization with kvm," *Linux J.*, vol. 2008, no. 166, Feb. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1344209.1344217

13. D. Mishchenko, *VMware ESXi: Planning, Implementation, and Security*, 1st ed. Boston, MA, United States: Course Technology Press, 2010.
14. A. Velte and T. Velte, *Microsoft Virtualization with Hyper-V*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 2010.
15. *LinuxKit*, https://github.com/linuxkit/linuxkit.
16. C. Fowler, *Trash Your Servers and Burn Your Code: Immutable Infrastructure and Disposable Components.* [Online]. Available: http://chadfowler.com/2013/06/23/immutable-deployments.html
17. A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proc. of EuroSys'15*. ACM, 2015.
18. M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *Proc. of EuroSys'13*. ACM, 2013, pp. 351–364.
19. *Auto Scalability Groups*, https://aws.amazon.com/autoscaling/.
20. *Elastic Load Balancing*, https://aws.amazon.com/elasticloadbalancing/.
21. *Cloud Formation*, https://aws.amazon.com/cloudformation/.
22. *Building Large Kubernetes Clusters*, https://kubernetes.io/docs/admin/cluster-large/.
23. C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," *SIGARCH Comput. Archit. News*, vol. 42, no. 1, pp. 127–144, 2014.
24. J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Souffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proc. of MICRO'11*. ACM, 2011.
25. *Kata Containers*, https://katacontainers.io.
26. E. Ernst, "Kata containers doesnt replace kubernetes," 2018, https://katacontainers.io/posts/why-kata-containers-doesnt-replace-kubernetes/.
27. F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My VM is lighter (and safer) than your container," in *Proc. of SOSP '17*. ACM, 2017, pp. 218–233.
28. A. Madhavapeddy and D. J. Scott, "Unikernels: Rise of the virtual library operating system," *Queue*, vol. 11, no. 11, pp. 30:30–30:44, Dec. 2013.
29. *Serverless Architectures*, https://martinfowler.com/articles/serverless.html.
30. *Etcd*, https://github.com/coreos/etcd.
31. D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. of USENIX ATC'14*. USENIX Assoc., 2014, pp. 305–320.
32. *AWS Fargate*, https://aws.amazon.com/fargate/.
33. G. Borello, "System and application monitoring and troubleshooting with sysdig." Washington, D.C.: USENIX Association, 2015.