i2kit: A Tool for Immutable Infrastructure Deployments based on Lightweight Virtual Machines specialized to run Containers

Pablo Chico de Guzmán
IMDEA Software Institute
Madrid, Spain
pablo.chico@imdea.org

Felipe Gorostiaga
IMDEA Software Institute
Madrid, Spain
felipe.gorostiaga@imdea.org

César Sánchez
IMDEA Software Institute
Madrid, Spain
cesar.sanchez@imdea.org

***Abstract*—Container technologies, like Docker, are becoming increasingly popular. Containers provide exceptional developer experience because containers offer lightweight isolation and ease of software distribution. Containers are also widely used in production environments, where a different set of challenges arise such as security, networking, service discovery and load balancing. Container cluster management tools, such as Kubernetes, attempt to solve these problems by introducing a new control layer with the container as the unit of deployment. However, adding a new control layer is an extra configuration step and an additional potential source of runtime errors. The virtual machine technology offered by cloud providers is more mature and proven in terms of security, networking, service discovery and load balancing. However, virtual machines are heavier than containers for local development, are less flexible for resource allocation, and suffer longer boot times.**

This paper presents an alternative to containers that enjoy the best features of both approaches: (1) the use of mature, proven cloud vendor technology; (2) no need for a new control layer; and (3) as lightweight as containers. Our solution is *i2kit*, a deployment tool based on the immutable infrastructure pattern, where the virtual machine is the unit of deployment. The *i2kit* tool accepts a simplified format of Kubernetes Deployment Manifests in order to reuse Kubernetes' most successful principles, but it creates a lightweight virtual machine for each Pod using Linuxkit. Linuxkit alleviates the drawback in size that using virtual machines would otherwise entail, because the footprint of Linuxkit is approximately 60MB. Finally, the attack surface of the system is reduced since Linuxkit only installs the minimum set of OS dependencies to run containers, and different Pods are isolated by hypervisor technology.

***Keywords*-service composition, deployment; immutable infrastructure; resource allocation;**

I. Introduction

Docker containers [1] have popularized the use of lightweight virtualization technologies such as LXC [2]. Some large companies report running all of their services 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. These features allow containers to provide an exceptional developer experience. 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].

However, containers also suffer some drawbacks. Although the high density is of great value in a *local environment* or for continuous integration (CI) jobs, containers introduce new challenges in *production environments* including security, networking, load balancing and service discovery. These challenges have already been addressed by traditional cloud vendor technology using virtual machines (VMs) as the unit of deployment, but these solutions are not immediately applicable when the unit of deployment is the container. Container cluster management tools—like Kubernetes [6], Docker Swarm [7] and Mesos [8]—attempt to address these issues directly. Kubernetes combines the goodness of Docker with Google best practices for massive deployments. For example, applications are defined using a declarative model based on Manifest Files, which are becoming the *de facto* standard to define application behavior at deployment time. Kubernetes also introduces the notion of a Pod, a group of strongly related containers that must be deployed on a single host. Finally, Kubernetes provides service discovery and load balancing in the context of containers, at the price of a full additional control plane layer, which requires an additional setup step and a sensitive runtime dependency that runs in the user's infrastructure. Kubernetes also imposes a non-negligible learning curve, and debugging its control plane when things go wrong can become extremely hard.

We argue in this paper that in production environments there is a better alternative than high density of containers. The first reason is that container isolation is weaker than the isolation provided by hypervisors, which has been reported as a security concern [9]. Second, although containers are very flexible in order to optimize the use of infrastructure resources, modern VM technologies offer machines with

increasingly small sizes (even a few hundred Mb) and that can be booted by seconds. The reduced size of Linux distributions specialized for running containers (Linuxkit is able to create Linux distributions with a 60MB footprint) reduces this difference in size and booting time, which diminishes the advantages of using high density of containers in production environments.

This paper presents *i2kit*, an open source tool for immutable infrastructure deployments. The *i2kit* tool reads a simplified version of Kubernetes Deployment Manifest Files, and transforms each Pod into a specialized virtual machine using Linuxkit. The *i2kit* tool applies AWS technology such as Elastic Load Balancers, Auto Scalability Groups and Route 53 Domains to solve the problems of networking, service discovery and load balancing (other cloud vendors technology could be easily supported). The key idea behind *i2kit* is to keep all the good features of Docker for local development and CI, and remove the challenges that container high density introduce in production environments.

The rest of the paper is organized as follows: Section II briefly describes Kubernetes and analyzes some of the challenges when adopting the Kubernetes technology. Section III introduces *i2kit* and explains how to map Kubernetes Deployments into native cloud vendor resources. Section IV describes the *i2kit* implementation by following a Kubernetes Deployment example. Section V measures the impact of *i2kit* on different metrics such as security, booting times, network-performance and cluster memory consumption. Finally, Section VI concludes and describes some research lines for future work.

II. Kubernetes. Advantages and Disadvantages

Kubernetes is an evolution of the Borg [10] and Omega [11] cluster manager tools, adapted for containers. We first introduce the best features of Kubernetes, which we adapt to *i2kit*, and then we analyze some weak aspects of Kubernetes, which *i2kit* is designed to target.

A. Kubernetes Desirable Features

Kubernetes deploys applications using three main entities: Pods, Replica Sets and Services, which are defined in Manifest Files following a declarative model. Fig. 1 shows the relations between these entities for the Kubernetes Deployment Manifest shown in Fig. 2. The next subsections describe in detail these concepts and how they are related.

1) Declarative Model: The declarative model of Deployment Manifest Files works as follows:

- The user declares the desired state of his application in a YAML Manifest File. Manifest Files include a description of which container images to run, the ports exposed by each container, the number of replicas and policies defining how to perform rolling updates.

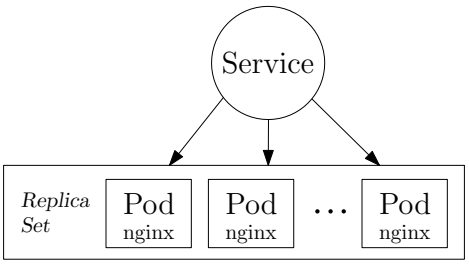

Figure 1. High-Level view of a Kubernetes Deployment.

```
apiVersion: apps/v1beta2
kind: Deployment
metadata: { name: nginx-deployment }
spec:
  selector:
    matchLabels: { app: nginx }
  replicas: 3
  template:
    metadata: { labels: { app: nginx} }
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports: [ containerPort: 80 ]
```

Figure 2. *Nginx* Manifest File.

- The Kubernetes Control Plane receives Manifest Files via its API, which are then recorded as part of the cluster's combined desired state.
- The Kubernetes Control Plane issues workloads to the nodes in the cluster. In our example, the Kubernetes control plane creates three *nginx* Pods, a Replica Set to ensure that three instances of the *nginx* Pod are always running, and a Service to load balance incoming traffic between these Pods.
- The Kubernetes Control Plane watches the Pod state in order to restore the desired state in case of a failure.

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 Kubernetes objects.

Occasionally, the cluster might evolve into an undesired state. Since Kubernetes keeps the desired state, it can then take actions to reconcile the current state and the desired state of the cluster. For example, if a node in the cluster becomes unreachable, the Kubernetes Control Plane will reschedule the Pods running on the node in a healthy node, reconciling the current and the desired state.

2) Pods: The unit of deployment in Kubernetes is not the container, but the Pod of containers. There are advanced use-cases that justify the run of multiple containers inside a single Pod. 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.

The Pod is essentially a sandbox that allows running containers inside. Informally, a Pod ring-fences an area of the host OS, builds a network stack, creates a bunch of kernel name-spaces, and runs one or more containers. Containers running in the same Pod share the same environment. For example, all containers in the same Pod share the same IP address, so if the containers need to communicate they can simply use the Pod localhost interface.

The deployment of a Pod is an all-or-nothing job. A Pod is never declared as up and available until every container is up and running. A Pod can only exist on a single node, even if the Pod runs multiple containers.

Pods is in some way an abstraction for having a dedicated machine for the containers in the Pod, without dealing with the high density of containers running on the same host.

3) Replica Sets: A Replica Set builds on top of a set of Pods. A Replica Set takes a Pod template and instantiates the desired number of replicas of the Pod. Replica Sets 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. In this sense, Kubernetes applies the immutable infrastructure principle [12] at the Pod level instead of at the host level.

4) Services: Pods are mortal and, in practice it is not unusual that a given Pod dies. On failure, the dying Pod is replaced by a new Pod, which probably runs on a different node with a different IP. Moreover, when performing rolling updates it is common that the new Pods have different IPs than the old Pods. Therefore, the application logic cannot rely on Pod IPs.

The solution for this problem is the use of services which provide a reliable networking endpoint for a set of Pods. Services can also load balance the traffic between a set of Pods.

5) Deployments: A very common use case of Kubernetes is an application that needs to run a set of Pods, ensuring a number of running replicas via a Replica Set, and preforming load balance of the incoming traffic to these replicas via a Service object. To facilitate this scenario, Kubernetes provides a higher-level object called Deployment. A Deployment is built using Pods, Replica Sets, and Services, making it transparent the use of these three objects. Deployments also provides versioning of Manifest Files, rolling updates strategies and rollback policies.

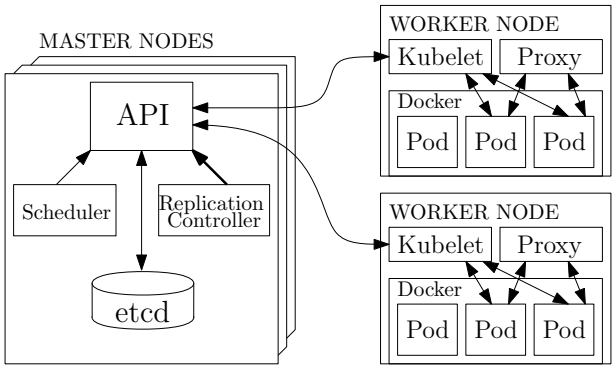

Figure 3. Kubernetes cluster high-level view.

B. The Drawbacks of Kubernetes

Fig. 1 shows a simple Kubernetes Deployment, built using the concepts described above. In order to support all these concepts, Kubernetes needs to implement a complex system. Fig. 3 gives a more precise view of the components which form the Kubernetes Control Plane.

First, Kubernetes requires a Distributed and Reliable Store Cluster. The most common solution to this end is *etcd* [13], a Key-Value Store based on the Raft [14] 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 of service discovery and load balancing in a high density container environment). This complex setup enables powerful deployment scenarios, and it is arguably a great fit for companies running their own infrastructure because these companies already need to handle cluster complexity. However, for the majority of companies and users running on the cloud, managing such a complex setup is complicated and error prone. 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.

All this complexity has been already solved in the context of virtual machines and thoroughly tested in practice. In some sense, Kubernetes is re-implementing similar pre-existing solutions but for environments with high density of containers per host. The consequences of supporting high density of containers are:

- an additional Control Plane layer which consumes resources and adds complexity.
- it requires virtual networking to provide a unique Pod IP, routable from any of the Worker Nodes.
- it imposes a semi-immutable infrastructure approach.

Immutable infrastructure [12] (*i2*) is an approach to

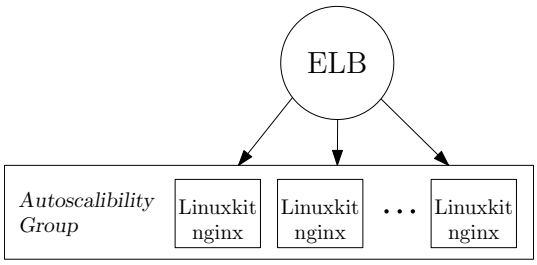

Figure 4. I2kit representation of a Kubernetes Deployment.

managing software deployments wherein the servers (where components run) are replaced rather than changed or modified in every software update. Kubernetes reuses Worker Nodes between Pods replacement so it is not considered to follow the immutable infrastructure principle. Kubernetes only applies immutable infrastructure principles at the Pod level, which is sometimes considered insufficient. For example, Worker Nodes might leak memory and become unreachable after a number of Pods re-deployments. Also, it is very common that Worker Nodes become unhealthy due to the lack of storage resources, for example by the garbage accumulated by old docker images from previous deployments.

III. I2kit Design

The *i2kit* tool aims to enjoy the good features of Docker and Kubernetes, while removing the security concerns of weaker container isolation and the extra complexity of self managing a Kubernetes cluster. In a nutshell:

- *i2kit* uses VMs as the unit of deployment, for security and infrastructure maturity reasons;
- every VM will host a Pod instance;
- the rest of the Kubernetes objects (Replica Sets, Services, Deployments) are replaced by the corresponding native cloud vendor technology.

Note that *i2kit* runs Pods, formed by the same Docker images that developers run in their local environments. In this manner, *i2kit* preserves the developer workflow untouched.

Fig. 4 shows a high-level view of the Amazon Web Services created by *i2kit* from the Kubernetes Deployment in Fig. 1. The next subsections explain how this transformation is performed.

1) Mapping Declarative Model: The declarative nature of Kubernetes Manifest Files is a simple and clean way of defining a deployment behaviour. The *i2kit* tool uses a simplification of Kubernetes Manifest Files in order to keep its nice declarative features.

Fig. 5 shows the *i2kit* Manifest file equivalent to the Kubernetes Deployment Manifest shown in Fig. 2, which is just a more concise representation of the same Kubernetes concepts.

```
name: nginx-deployment
replicas: 3
containers:
    nginx:
        { image: nginx:1.7.9 , ports: [ 80 ] }
```

Figure 5. *Nginx i2kit* Manifest File.

The *i2kit* implementation is greatly simplified by relying on the AWS Cloud Formation Service [15]. 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 also follows a declarative description approach. Cloud Formation templates can specify rolling updates policies to be applied when the template is modified, allowing the simulation of Kubernetes rolling updates. Finally, Cloud Formation keeps a record of the different template versions, making this information available to *i2kit*. Section IV shows a Cloud Formation template example generated for the *nginx* Deployment.

2) Mapping Pods: Pods are a very useful abstraction of a virtual machine in a node with high density of containers. There is a trivial way to map Pods into *i2kit*, because every Pod instance will run in its own VM. The drawback of this simple mapping is a loss of performance, because a virtual machine imposes a non-negligible overhead on infrastructure resources. However, there are tools to create minimal Linux distributions specifically crafted to run containers. The footprint of these distributions can get as small as 60MB nowadays, a size comparable to container technology. The tool *i2kit* is built on the assumption that the overhead of running a Pod per virtual machine is acceptable, and this comparison will keep improving as leaner Linux distributions are developed. Section V-1 discusses this issue. For example, Unikernels [16] have the promise of reducing this overhead to the minimum by specializing a Linux kernel for the software that will be running on top. Note also that a small overhead is even more acceptable in the context of production environments, where IT operators tend to be generous allocating resources and where over-provisioning is common.

Additionally, security is an important concern [9] when running Pods from different applications in the same node. The *i2kit* approach alleviates these concerns using directly proven cloud technology built on virtual machines.

The *i2kit* tool uses Linuxkit [17], a toolkit for building custom minimal, immutable Linux distributions. Linuxkit reads YAML templates that describe how to build a Linux distribution. Linuxkit templates support the ability to define *services*, which are a set of containers to be run when the VM boots. Therefore, every attribute of a Pod has a counterpart in a Linuxkit template. Section IV explains this

equivalence using an example.

Linuxkit templates also include sections for the *kernel* filesystem and the *init* processes, making it possible to install the minimum set of OS dependencies to allow the execution of containers. This minimalist approach also improves the security of the system by reducing its attacking surface, since unnecessary dependencies are removed.

As a consequence of running every Pod instance in its own specialized VM, *i2kit* has also immediately solved the problem of virtual networking, since the Pod unique IP is mapped to the node unique IP.

3) Mapping Replica Sets: A Replica Set ensures that a fix number of Pod instances are always running. Replica Sets also replace Pods that get unreachable. Non surprisingly, in the realm of virtual machines there are also solutions that perfectly map this behaviour under the assumption of running a single Pod per VM. For example, Amazon Web Services offers Amazon Auto Scalability Groups [18]. Auto Scalability Groups help to maintain the health and availability of a fleet of Amazon virtual machines, ensuring that the desired number of VMs is always running. If a VM becomes unhealthy, it also gets replaced.

Note that in the event of a rolling update, a new Amazon Machine Image (AMI) is generated, and the Auto Scalability Group replaces every existing virtual machine by new VMs running the last AMI built. The tool *i2kit* does follow a pure immutable infrastructure approach by design.

4) Mapping Services: A Kubernetes Service provides two different functions: (1) it proxies incoming requests between a set of Pods, and (2) it provides a custom immutable endpoint which resolves to the associated Pods. Again, there is an immediate solution in Amazon Web Services for proving proxy capabilities, by simply using Amazon Load Balancers [19].

It is possible to automatically attach every virtual machine created by an Auto Scalability Group to an AWS Load Balancer at creation time. The AWS Load Balancer provides a reliable endpoint for our set of Pod instances. The port configuration of the AWS Load Balancer is created based on the information contained in the Deployment Manifest.

There is a drawback with this solution, though: AWS Load Balancer endpoints are not customizable, while the Kubernetes Service endpoints are. To overcome this obstacle, *i2kit* creates a Route 53 Domain CNAME entry resolving to the AWS Load Balancer endpoint using the Deployment Manifest *name* field. In this manner, *i2kit* provides the same service discovery mechanism—based on names—than Kubernetes.

IV. I2KIT IMPLEMENTATION

The *i2kit* tool is open source, and it is actively under development at the IMDEA Software Institute[1]. Currently

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

```
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:alpine,
      capabilities: [ all ] }
trust:
  org: [ linuxkit, library ]
```

Figure 6. *Nginx* Linuxkit template.

i2kit transforms Deployment Manifests into AWS Cloud Formation templates, but support for other cloud vendors is easy to implement. This section describes how *i2kit* processes the Deployment Manifest shown in Fig. 5 and creates a Cloud Formation template. The first step transforms the Pod information contained in a Deployment Manifest into a Linuxkit template, in order to generate a minimal Linux distribution specialized in running the Pod containers. The result is shown in Fig. 6. From every container in the Pod, *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:

```
services:
   { image: nginx:alpine, 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 to every container.

The remaining fields in the Linuxkit template are pre-generated and are identical for all Pods. 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 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 VM itself. Then, 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: nginx-deployment
      Listeners:
        LoadBalancerPort: 80
        InstancePort: 80
        Protocol: HTTP
  DNSRecord:
    Type: AWS::Route53::RecordSet
    Properties:
      HostedZoneName: i2kit.com
      Name: nginx-deployment.i2kit.com
      ResourceRecords:
      - Fn::GetAtt:("ELB", "DNSName")
      Type: CNAME
```

Figure 7. Cloud Formation template for the *nginx* Deployment.

containers in the `services` section run as daemons in parallel, in particular *getty*, *sshd* and the Pod containers. 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. Assume the id of this AMI is *ami-XXXXX*. The next step is the generation of the Cloud Formation template, shown in Fig. 7.

The Cloud Formation template will create four different resources: `LaunchConfig`, `ASG`, `ELB` and `DNSRecord`. The resource `LaunchConfig` defines how virtual machines will be created. Each machine will use the previously created AMI. The next resource is `ASG`, an Auto Scalability Group which use the `LaunchConfigurationName` created above in order to create new VMs. The minimum and the maximum number of virtual machines matches the number of replicas in the Deployment Manifest. Every VM generated by the Auto Scalability Group is associated to 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 Deployment Manifest, where the `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 Deployment Manifest `name` field. It resolves to the `ELB` endpoint, providing service discovery for other deployments.

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

(a) Memory comparison

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

(b) Network comparison

Table I
COMPARISON OF *i2kit* VS KUBERNETES.

V. EMPIRICAL EVALUATION

This section compares *i2kit* versus the native Kubernetes implementation based on three different metrics: memory consumption, network performance, resource fragmentation and booting times. Additionally, we compare both approaches qualitatively in terms of security.

1) Memory Consumption: The overhead that *i2kit* imposes for every Pod creation is a consequence of the overhead of the VM running the Linuxkit distribution. In contrast, the Kubernetes overhead for running a Pod is the overhead of running a Worker Node, which can be shared by several Pods. Table I(a) shows the memory consumption for running the *nginx* deployment example using different numbers of Pods replicas. Table I(a) displays the memory footprint of the total amount of virtual machines that are created (for *i2kit*) and the memory footprint of a single Worker Node running all the Pod replicas (for Kubernetes). The Worker Node memory overhead per Pod gets better when more Pods run on the same Worker Node.

Table I(a) shows that *i2kit* is more memory efficient when running less than (approx.) 30 Pods per Worker Node. Note that the Kubernetes web page [20] does not recommend running more than 30 Pods per Worker Node. Therefore, we can conclude that the memory consumption of *i2kit* behaves very well compared to Kubernetes. 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 Machine. Moreover, the data reported in Table I(a) does not take into account the memory consumption of Master Nodes, which would report a more favorable comparison to *i2kit*.

Finally, there is a very active research effort targeting VM optimization [21], [22] which *i2kit* can leverage in terms of memory usage. Unikernels [16], for example, are very promising on this area. Also, other virtual machine optimizations have been studied in the context of programing languages [23].

2) Network Performance: Table I(b) shows the network performance comparison between *i2kit* and different Kubernetes configurations. The experiment uses *iperf2* to measure the average network bandwidth consumed by each Pod, where each Pod runs an *iperf2* server. On the other hand, the *i2kit* configuration runs every *iperf2* server in its own *t2.large* AWS Machine. In the table, *K8-N* stands for a Kubernetes cluster with *N* Worker Nodes, where every Worker Node runs on a *t2.large* AWS Machine. 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 I(b) indicates that *i2kit* scales linearly on the number of Pod 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 I(b) shows that the overhead imposed by Kubernetes when running on a single node is not very relevant (approximately 1-2%). Since *K8-1* runs all Pod replicas on the same machine, running more than one Pod replica quickly hits the VM incoming bandwidth limit. 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. Also, *K8-5* shows how the traffic is dramatically affected by the virtual machine incoming bandwidth limit when running 25 Pod replicas.

3) Resource Fragmentation: The resource fragmentation suffered by *i2kit* is implicitly higher than Kubernetes, since the size of the smallest VM in AWS is 512MB. However, this figure is competitive in production environments where services tend to consume on the order of Gigabytes. Also, container-based serverless architectures [24] are a promising option for alleviating *i2kit* resource fragmentation.

On the other hand, sharing a host between different Pods imposes performance side effects on the rest of Pods running on the same host. Although some research has been done in this area, [25], [26], in practice IT operators reserve fix memory and CPU resources for every Pod, introducing a similar resource fragmentation than *i2kit*.

As a final note, virtual machines running Worker Nodes tend to waste additional resources because Worker Nodes do not run at full capacity all the time, which introduces another level of resource fragmentation. The tool *i2kit* creates VMs on demand, not spawning more virtual machines than needed, and avoiding the idle Worker Node problem altogether.

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

5) Security: There is an intrinsic security concern about running Pods from different applications on the same Worker node [9]. Some use cases require higher level of isolation, like sandboxes for running vulnerable or untrusted code, or multi-tenant environments in the case of hosted services. Container isolation uses 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 (and they have helped to mitigate some kernel vulnerabilities [27]), but it only takes a single kernel bug to bypass all these mechanisms and escape the container isolation model (see [28] for some vulnerabilities).

This is a strong point in favor of the isolation that *i2kit* provides compared to Kubernetes (see also [21]). The approach of *i2kit* is to isolate Pods using secure virtualization technology. Note that a malicious Pod that takes control of a Worker Node could leak information about any application running on the same Kubernetes cluster. Advanced attacks can also be done by simply checking the performance of the Pod, as it has been studied in the context of web browsers [29]. Finally, *i2kit* uses Linuxkit to install just the minimum set of dependencies to run the user Pods, greatly reducing the attacking surface of the system.

VI. Conclusions and Future Work

This paper has presented *i2kit*, a deployment tool that pursues the following main goals: (a) to preserve the docker development workflow untouched; (b) to reuse popular parts of Kubernetes by accepting a simplification of Kubernetes Manifest Files, which also eases the adoption of *i2kit*; (c) to eliminate the complexity of maintaining a Kubernetes cluster by running a single Pod per virtual machine; and (d) to improve the security concerns of running containers in production.

The *i2kit* tool eliminates some of Kubernetes complexity for production environments by running on public cloud vendors. The results in Section V suggest that the memory consumption of *i2kit* is comparable to Kubernetes, that the network performance of *i2kit* is better than the one of Kubernetes, and that the price of having slower booting times and larger resource fragmentation seems acceptable in production environments.

The tool *i2kit* is a deployment tool that tries to exploit synergies between the world of containers (widely used for development) and the world of virtual machines (widely used for production). Our tool creates a feedback loop where developers are able to improve their local workflows using containers, and operators transform those containers into production-ready minimal virtual machines. Current and future research includes trying to optimize further the generation of minimal virtual machines by reducing the size of our base Linuxkit distribution, and by using Unikernels. Another research line is to integrate *i2kit* with server-less architectures [24] provided by cloud vendors.

Finally, *i2kit* not only reduces the complexity of the deployment system by using cloud vendor technology not managed by the end user, but *i2kit* also brings a significant security improvement for two reasons. First, malicious Pods are isolated by secured and proven hypervisor technology. Second, the use of specialized Linuxkit distributions reduces the attacking surface of the resulting system. The tool *i2kit* can be improved even further to limit the set of capabilities needed by the user Pods. For example, the *nginx* container of the Linuxkit template shown in Fig. 6 could also limit its capabilities to `CAP_NET_BIND_SERVICE`, `CAP_CHOWN`, `CAP_SETUID`, `CAP_SETGID` and `CAP_DAC_OVERRIDE`.

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] A. Mouat, "Five security concers when using docker," https://www.oreilly.com/ideas/five-security-concerns-when-using-docker.

[10] 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.

[11] 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.

[12] 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

[13] *Etcd*, https://github.com/coreos/etcd.

[14] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. of USENIX ATC'14*. USENIX Assoc., 2014, pp. 305–320.

[15] *Cloud Formation*, https://aws.amazon.com/cloudformation/.

[16] 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.

[17] *LinuxKit*, https://github.com/linuxkit/linuxkit.

[18] *Auto Scalability Groups*, https://aws.amazon.com/autoscaling/.

[19] *Elastic Load Balancing*, https://aws.amazon.com/elasticloadbalancing/.

[20] *Building Large Kubernetes Clusters*, https://kubernetes.io/docs/admin/cluster-large/.

[21] *Kata Containers*, https://katacontainers.io.

[22] 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.

[23] J. F. Morales, M. Carro, and M. V. Hermenegildo, "Description and optimization of abstract machines in a dialect of Prolog," *TPLP*, vol. 16, no. 1, pp. 1–58, 2016.

[24] *Serverless Architectures*, https://martinfowler.com/articles/serverless.html.

[25] 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.

[26] 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.

[27] *Docker Security Non-Events*, https://docs.docker.com/engine/security/non-events/.

[28] *Linux Kernel Security Vulnerabilities*, https://www.cvedetails.com/vulnerability-list.php.

[29] P. Vila and B. Köpf, "Loophole: Timing attacks on shared event loops in Chrome," in *Proc. of USENIX Security Symposium*. USENIX Assoc., 2017, pp. 849–864.