Characterising Resource Management Performance in Kubernetes

Víctor Medel[a], Rafael Tolosana-Calasanz[a], José Ángel Bañares[a], Unai Arronategui[a], Omer Rana[b]

[a] *Aragon Institute of Engineering Research, University of Zaragoza, Spain*
[b] *School of Computer Science & Informatics, Cardiff University, UK*

Abstract

One of the challenges for enabling elastic automated resource management in cloud computing is to accomplish effective automated resource management actions, which include provisioning, maintaining, and de-provisioning of computing power. Among the cloud resources currently available, containers are rapidly replacing Virtual Machines (VMs) as the compute instance of choice in cloud-based deployments. One of the reasons is the significantly lower overhead of deploying and terminating containers in comparison to VMs. Understanding performance associated with deploying, terminating and maintaining a container is therefore significant. In this paper, we analyse performance of the Kubernetes system and develop a Petri net-based model of resource management within this system. Our model is characterised using real data from a Kubernetes deployment, and can be used as a basis to design scaleable applications that make use of Kubernetes.

Keywords: Performance Models, Cloud Resource Management

*Corresponding authors
Email addresses: vmedel@unizar.es (Víctor Medel), rafaelt@unizar.es (Rafael Tolosana-Calasanz), banares@unizar.es (José Ángel Bañares), unai@unizar.es (Unai Arronategui), ranaof@cardiff.ac.uk (Omer Rana)

1. Introduction

Cloud systems enable computational resources to be acquired (and released) on-demand and in accordance with (changing) application requirements. Users can rent computational resources of different types: virtual machines (VMs), containers, specialist hardware (e.g. GPU or FPGA), or bare-metal resources, each having their own characteristics and cost. An effective automated control of cloud resource (de-) provisioning needs to consider [**?**]: (i) resource utilization, (ii) economic cost of provisioning and management, and (iii) the resource management actions that can be automated. Increasingly, many cloud providers support resource provisioning (and billing) on a per second or even per millisecond basis, such as GCE[1], or Amazon Lambda[2] – referred to as "serverless computing". Therefore, understanding performance associated with deploying, terminating and maintaining a container that hosts that function is significant, as it affects the ability of a provider to offer finer grained charging options for users with stream analytics/ processing application requirements. Provisioning and de-provisioning actions are subject to a number of factors [**?**], mainly: (i) the *overheads* associated with the action (e.g. launching a new VM can often take minutes [**?**]); and (ii) the actual processing time required can vary due to resource contention – leading to uncertainty for the user.

Kubernetes [**?**] is a system that enables a container-based deployment within Platform-as-a-Service (PaaS) clouds, focusing specifically on cluster-based systems. It can provide a cloud-native application (CNA) [**?**], a distributed and horizontally scalable system composed of (micro)services, with operational capabilities such as resilience and elasticity support. From an architectural point of view, Kubernetes introduces the *pod* concept, a group of one or more containers (e.g. Docker, or any OCI compliant container system) with shared storage and network. In this paper, we investigate deploying, terminating and maintaining performance of (Docker) containers with Kubernetes, identi-

[1] `https://cloud.google.com/`

[2] `https://aws.amazon.com/lambda/`

fying operational states that arise with the associated *pod*–container. This is achieved through Reference Nets (a kind of Petri-Net (PN) [**?**]) based models. The models can be further annotated and configured with deterministic time, probability distributions, or functions obtained from monitoring data acquired from a Kubernetes deployment. It can also be used by an application developer / designer: (i) to evaluate how pods and containers could impact their application performance; or (ii) to support capacity planning for application scale-up / scale-down.

This paper extends [**?**] by: (i) the inclusion of additional experiments in a larger cluster; (ii) considering the impact of variable latency/Round-Trip Time (RTT) in the communication network; (iii) analysing the impact of varying the number of containers inside a pod; (iv) analysing the impact of downloading a container image at deployment time; (v) using rules to assist developers to better structure their Kubernetes deployment. The paper is organized as follows. In Section 2, we describe our model. Section 3 shows our pod abstraction overhead characterization. We discuss the deployment results in Section 4 and related work in Section 5. The conclusions are outlined in Section 6.

2. Kubernetes Overhead Analysis & Performance Models

The Kubernetes architecture incorporates the concept of a pod, an abstraction that aggregates a set of containers with some shared resources at the same host machine. It plays a *key* factor in the overall performance of Kubernetes.We make use of Reference Nets to model pods and containers and to conduct performance analysis. Reference Nets can be interpreted by Renew[**?**], a Java-based Reference net interpreter and a graphical modelling tool.

2.1. Kubernetes Performance Model

Kubernetes supports two kinds of pods: (i)*Service Pods*: They are run permanently, and can be seen as a background workload in the cluster. Two key performance metrics are associated with them: (i.a) availability (influenced by

faults and the time to restart a pod/container) and (i.b) utilisation of the service (impacting response time to clients). For example, high utilisation leads to an increased response time. Several Kubernetes system services (e.g. container network or DNS) and high level services (e.g. monitoring, logging tools) are provided by Service Pods. (ii) *Job/batch Pods*: They are containers that execute tasks and terminate on task completion. For a Job pod, both deployment and total execution time (including restarting, if necessary) are important metrics. The restart policy of these containers can be *onFailure* or *never*.

When a pod is launched in Kubernetes, it requests resources (RAM and CPU) to the Kubernetes scheduler. If enough resources are available, the scheduler selects the *best* node for deployment. The requested CPU could be considered as a reservation in contingency situations. For instance, when a container is idle (e.g. it is inside a service pod and the service has low utilisation), other containers can use the CPU. With this resource model, the overall performance of the pod depends on its resource requests and on the overall workload. We could define a CPU usage limit, but then some resources might remain unused.

We model a pod's life cycle in order to estimate the impact of different scenarios on the deployment time and the performance of the applications running inside a pod. In Kubernetes, a pod's life cycle depends on the state of the containers that are inside it. For instance, a pod has to wait until all its containers are created. With the Reference Nets abstraction, we can provide an unambiguous hierarchical representation of the Kubernetes manager system as the System Net and the Pods (with the containers) as the Token Nets. The tokens inside our Token Net represent containers and the tokens inside our System Net represent Pods, as illustrated in Figures 1 and 2. The models were derived from the Kubernetes documentation [3]; specifically, from the Pod Lifecycle section [4] and from the Resource Management section [5]. Details about places and transi-

[3] `https://kubernetes.io/docs`

[4] `https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/`

[5] `https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/`

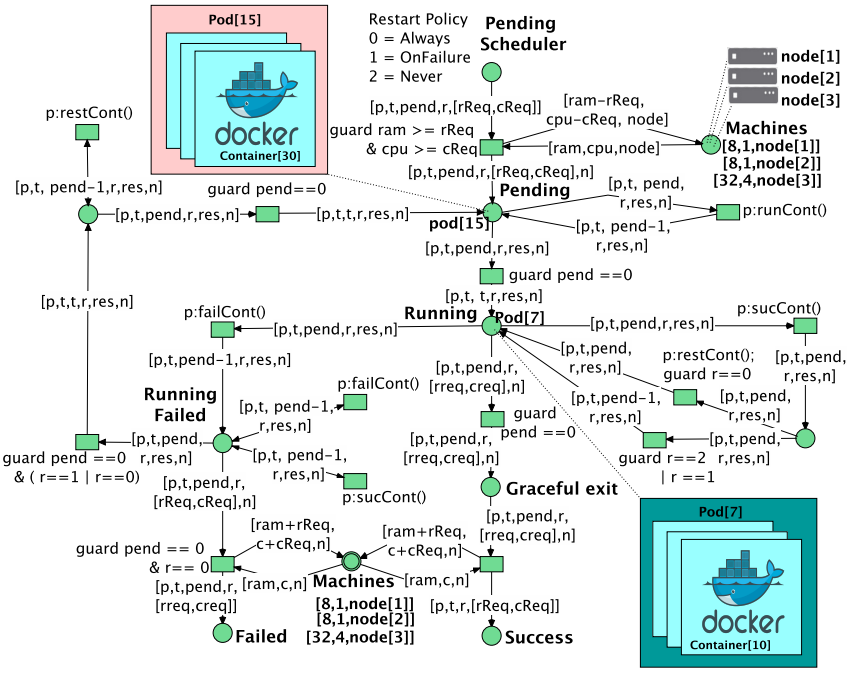

Figure 1: Model of the life cycle of pods in Kubernetes.

tions, needed to specify the initial marking, are hidden to improve legibility. In addition, we assume that the scheduler assigns a pod to a single node arbitrarily, as long as the machine has enough resources available. If there are not enough resources in the cluster, the pod waits in **Pending Scheduling** place. This behaviour could be refined by introducing more sophisticated policies and a rejection place for pods. **Machines** place [6] represents the resources managed by the scheduler. For each machine, there is a tuple token with the identification of the node, the available RAM size and number of available cores. Figure 1 shows three machines ranging from 8GB to 32GB, with 1 to 4 cores. The resources

[6]It should be noted that **Machines** place appears twice: One with a single circle (actual definition) and with a double circle (a duplication to simplify the model). Reference nets support double circle to simplify the model and to improve its legibility. If it were not used, several arcs would cross the model with their corresponding arc labels

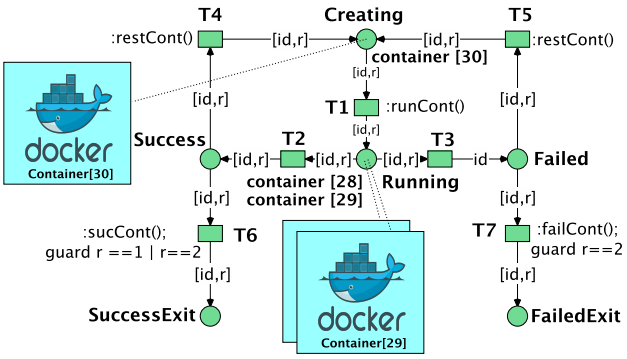

Figure 2: Model of the life cycle of containers inside a pod. r models the restart policy of a container – Always = 0, OnFailure=1, Never = 2.

assigned to a pod are only released when the pod restart policy is "never" or "onFailure".

Once the pod has been assigned to a machine, Kubernetes starts creating the containers – it is in **Pending Scheduler** place – while the pod waits in its **Pending** place. Both nets are synchronized through the inscription *runCont*. In this way, when a container in a Pod changes to **Running** place in Figure 2, the number of pending containers in this pod is decremented in the **Pending** place of Figure 1. When all containers are running in the pod, the transition with the **guard pend==0** is fired and the pod states changes to **Running**.

While the pod is in **Running** state, it is waiting for its containers to terminate. If a container fails, the pod goes to **RunningFailed** place where it waits for the termination of all containers (with a potential restart action). If there are no failures, the pod will be in **Running** place or eventually will reach **Success** place when all containers have finished.

Figure 2 illustrates the behaviour of a container. A token in that net represent a container. A pod's restart policy is included in the net. A created pod enters the **Running** place, and may reach the **Success** or **Failure** place. The firing of the corresponding Transitions (**T1**, **T2** and **T3**) is synchronised with the System Net. According to the restart policy, the containers might return to

Transition	Variable	Trans.	Variable
T1	Container creation	**T2**	Container execution
T3	Time to Container failure	**T4**, **T5**	Container termination time if the container is restarted
T6, **T7**	(Graceful) Container termination		

Table 1: Timed transitions in the model

Running place or they might finish in **SuccessExit** or in **FailedExit** places. We include several timed transitions, as summarised in Table 1. By default, the firing of **T2** and **T3** is arbitrary and non-deterministic; however, with Renew, it is possible to simulate any probability distribution for them in order to simulate a failure. Additionally, it is possible to assign different random distributions for the timed transitions. In the next sections, we describe different experiments to obtain the real value of these metrics. The termination time (**T6** and **T7**) and the termination time when a container is restarted (**T4** and **T5**) do not depend on the success of the container, so both transitions are modelled with the same distribution. When a container is restarted, the total restarting time can be calculated as **T4** – or **T5**– + **T1**.

2.2. Experiments to feed the performance model

We conducted several experiments to estimate the value of transitions in Table 1 by deploying Kubernetes on a cluster with eight physical machines – $n = 8$–, each with 32GB of RAM and 4 Intel i5-4690(3.500GHz) cores. The results are shown in the next subsections. The performance metrics of the high level model (Figure 1) are determined by the firing sequences of transitions in the Token Net (Figure 2). For example, if there is a pod with three containers, the **T1** transition of this pod is fired three times. The pod is waiting this time in the **Pending** place.

2.2.1. Benchmarking Starting Time

To estimate the value of Transition **T1**, we launched a variable number of containers, whose image was preloaded at all the machines, and measured

the total deployment time. Each experiment was repeated 30 times, and we calculated the mean, the standard deviation, and the confidence intervals (for $\alpha = 0.05$). In order to calculate the confidence interval, we assumed (by the Central Limit Theorem) that the underlying distribution of the sampled mean follows a normal distribution.

In Figure 3, we show how different variables influence the deployment time. These variables are: (i) The *number of machines* available in the cluster (n). We observe that the scheduler launches pods sequentially, without multi-threading (red line in Figure 3), showing a linear total deployment time with the number of deployed containers. (ii) The *number of containers C inside a pod* (ρ factor). The ρ factor is calculated as follows: $\rho = \frac{\#Pods}{C}$. For instance, a ρ factor of 0.25 implies that there are 4 containers inside each pod. We can see that the time to deploy 10 pods with 4 containers in a single machine is 25.89 s., which is higher than the time to deploy 10 pods in a single container on a single machine (16.01 s.). (iii) *Cluster constraints* as the number of nodes.

The total provisioning time (T_t) is calculated as: $T_t = T_d + T_{down}$ Where T_d is the time to deploy pods and containers on physical machines and T_{down} is the time to download the needed container image to the involved machines. Our experiments show that T_d has a linear behaviour. Therefore, T_d depends on the number of deployed containers C and on the number of deployed pods, $\#Pods$: $T_d = \frac{C\ T_c(\rho,n,C)}{\min\{\#Pods,n\}}$. As the scheduler manages the pod as the minimal schedulable unit, the maximum number of pods deployed in parallel in a cluster is given by $\min\{\#Pods, n\}$. T_c is a function that returns the time to create a single container. This value depends on how the deployment is structured – ρ and C parameters – and the number of machines in the cluster (n). In Figure 4, we show different values for T_c, obtained experimentally. We can see that for large C values, and as ρ approaches 0, T_c becomes constant. Therefore, we can write: $\lim_{\rho\to 0, n\to\infty, C\to\infty} T_c(\rho, n, C) = t_c$. Under these assumptions, the T_c value could be considered as a constant – attached to **T1** transition.

In order to characterise the impact of container image download time, we repeated the previous experiments without preloading the image. The image is

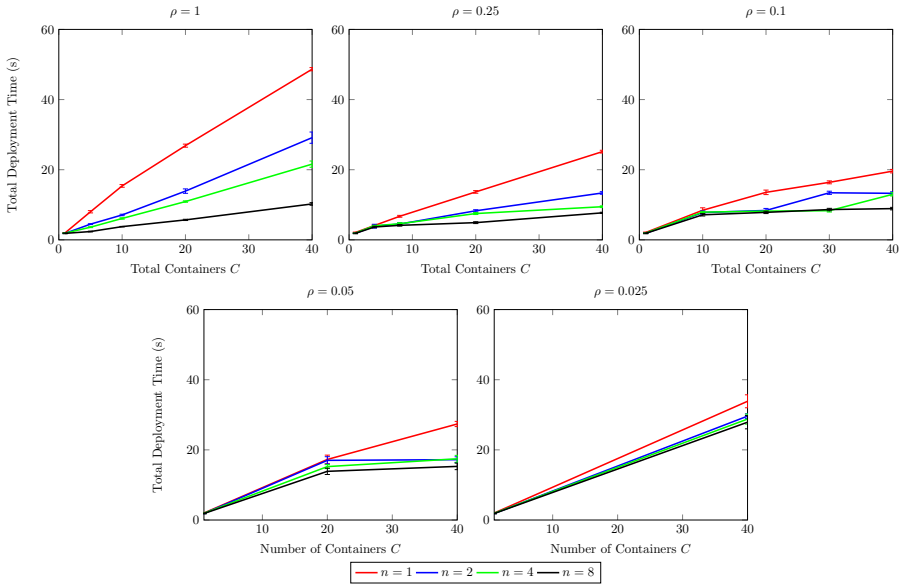

Figure 3: Total deployment time (T_d) vs. Number of deployed containers (C). Each graph shows: mean time, confidence interval for the mean for a varying number of machines in cluster, n. The results are grouped by the number of containers inside a pod, $\frac{1}{\rho}$

downloaded from a machine located inside the cluster connected directly to the same switch. The results are shown in Figure 5. The results of the homogeneous latency scenario are better than the ones of the variable latency scenario – as ρ tends to zero, the latency impact on the results decreases. We can see that as the number of machines increase, the total deployment time also increases, because the image needs to be downloaded by all the machines in the cluster, and all the machines are connected to the same server. When the number of deployed pods is greater than the number of machines, the deployment time remains stable, so we can conclude that Kubernetes only downloads the image once per machine.

Several variables related to the cluster architecture impact the deployment time, such as parameters of the physical machines and the network topology. To assess the network topology impact, we repeated the experiments in a cluster

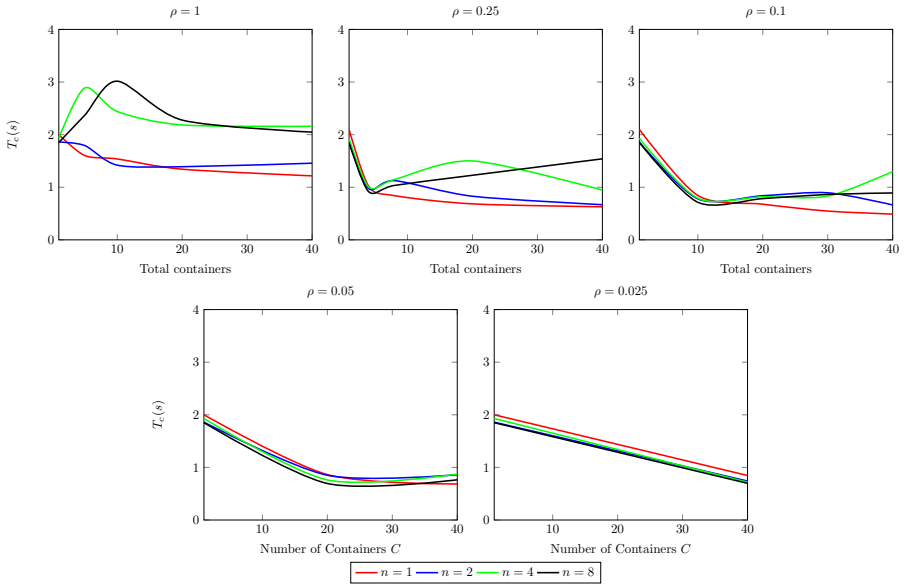

Figure 4: Time to create a *single* container (T_c function) vs. Number of deployed containers (C). Each graph shows: mean time, confidence interval for the mean for a varying number of machines in cluster, n. The results are grouped by the number of containers inside a pod, $\frac{1}{\rho}$

with heterogeneous latency. We simulated that half of the machines are in a different network area, so that their Round Trip Time (RTT) is about 100ms. The RTT for the rest of the machines is 0.25ms. Table 2 depicts the results for $\rho = 1$ and $n = 8$. The results for other values of ρ and n are quite similar. We can see that the latency has not a significant impact on T_d – and neither on T_c. As in T_t is included the time to download the container image, this value is higher. However, the size of the image mitigates the latency impact.

2.2.2. Benchmarking Termination Time

A Pod is expected to be terminated at some time. If it is a service, and consequently it has to be running all the time, the termination may be due to a failure and the pod has to be restarted. This philosophy also applied to containers, as we discussed previously in the Container Net model (transitions

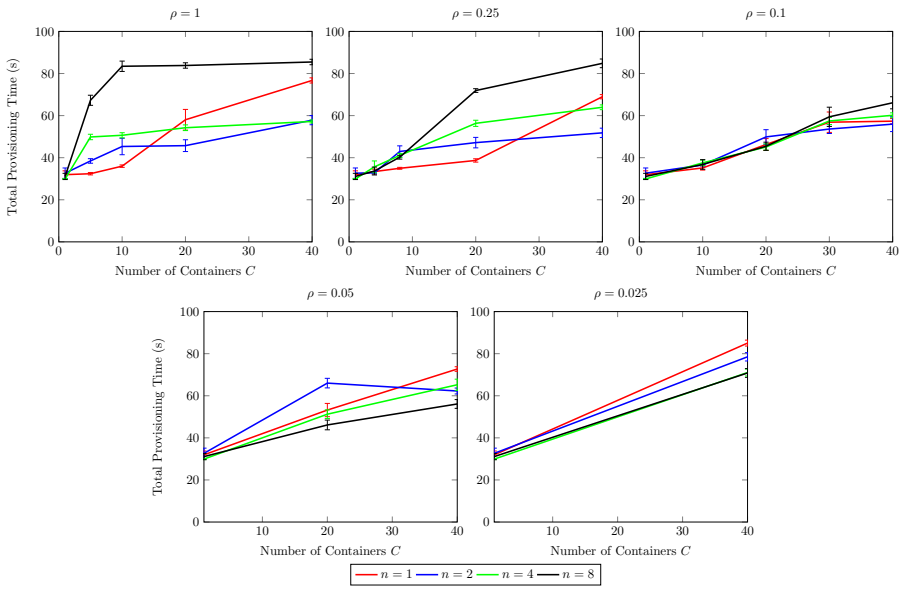

Figure 5: T_t vs. C. Each graph shows: mean time, confidence interval for the mean for a varying number of machines in cluster, n. The results are grouped by the number of containers inside a pod, $\frac{1}{\rho}$. The container image (1.225 GB) is not present in the machines.

T4, **T5**, **T6**, **T7**). We consider **T3** to depend on the application, and it represents the failure rate (or the time between failures). When a pod terminates, Kubernetes waits for a grace period (which, by default, is 30 seconds) until it kills any associated container and data structures.

As far as we have tested, the only variable that affects termination time of a pod is the number of containers in that pod. This occurs because when a pod finishes, all its containers have to be finished; however, in a normal scenario, pods finish – or restart – asynchronously. Therefore, the overhead caused by finalising several pods on several machines is negligible. As ρ approaches zero, the mean time to stop a single container inside a pod remains constant. The way in which these times are aggregated and synchronised depends on the scenario, and the specific performance metrics can be derived from the complete Petri Net Model.

	Homogeneous RTT		Heterogeneous RTT	
C	T_d	T_t	T_d	T_t
1	1.85	31.09	1.94	33.26
5	2.37	67.28	2.66	66.79
10	3.77	83.44	3.87	82.66
20	5.69	83.81	5.56	86.93
40	10.24	85.47	10.21	91.02

Table 2: T_d and T_t values from a Kubernetes cluster with homogeneous RTT (0.25ms) and from a Kubernetes cluster with heterogeneous RTT. $\rho = 1$ and $n = 8$. The container image is 1.225 GB. The results are in seconds.

In order to associate the corresponding metric for the transitions, we perform the experiments shown in Table 3 in the same cluster, as in the previous section. We present the results for **T4** and **T6** which correspond to a successful scenario. Without taking into account the time to detect the failure, the behaviour of transitions **T5** and **T7** is similar: (i) *Transitions **T4, T5***: These transitions measure the time to stop a container when it is going to be restarted. We have deployed pods with a variable number of containers to measure the time. The results are shown in column "**T4** per Container" in Table 3. When we decrease ρ, the mean time to terminate a container remains constant. Additionally, the highest measured mean time is $\sim$ 0.3s and 80% of sampled times are $<$ 0.22s. (ii) *Transitions **T6, T7***: These transitions model the normal behaviour of Kubernetes. On successful completion, Kubernetes waits for the grace period and deletes all the data structures associated with a container. We measured these variables in columns "**T6** Graceful termination" and "**T6** per container" in Table 3. For these experiments, we set the grace period to 30s (the default). We can observe that the stopping time remains constant for more than 10 containers in a pod (Column "**T6** per container") and for low values is negligible. It is interesting to note that the time to stop a container is higher when the container is going to be restarted. This overhead is about 10ms.

C	ρ	**T4 (T5) per Container**	**T6 (T7) Graceful termination**	**T6 (T7) per Container**
1	1	0.01	30	0
10	0.1	0.11	30.99	0.10
20	0.05	0.12	32.8	0.14
40	0.025	0.15	34.69	0.11
60	0.016	0.16	37.04	0.11

Table 3: **T4** and **T6** experimental results (in seconds).

3. Overhead Analysis of the Pod Abstraction

The pod abstraction allows several containers to be grouped together sharing different resources. However, the way in which resources are shared between containers in the same pod and the impact on the performance of a container are not easy to determine. In this section, we analyse this performance change based on how the deployment is structured (e.g. the number of containers inside each pod), using the total execution time as a metric. We performed several experiments to measure the overhead induced by the pod abstraction. The aim is to measure how transition **T2** is affected by the deployment configuration.

Let us consider the following scenarios: (i) *Scenario 1*: A pod is deployed and all the containers are inside that pod – $\rho = 1/c$. (ii) *Scenario 2*: Several pods are deployed and there is exactly one container inside each pod – $\rho = 1$.

The total number of containers deployed is given by C and all pods are deployed on the same machine. The machine has 12 Intel Xeon E5-2620 (2.00GHz) cores and 32GB of RAM. Each experiment, one for each scenario, was repeated 30 times, so that we can consider that the probability distribution of both means follows a normal distribution (by the Central Limit Theorem). We present the mean execution time (μ_i) and the standard deviation (σ_i). In order to compare both scenarios, we propose the following statistical hypothesis test:

$$\begin{cases} H_0 : \mu_1 - \mu_2 = 0 \\ H_1 : \mu_1 - \mu_2 \neq 0 \end{cases}$$

As we assume that both means follow a normal distribution and they have

	Scenario 1		Scenario 2		
C	μ_1	σ_1	μ_2	σ_2	$\mu_1 - \mu_2 = 0$?
1	123.47	0.43	123.38	0.39	Yes
4	473.65	0.96	475.15	0.62	No
8	946.90	0.72	946.63	0.69	Yes
12	1417.76	1.67	1420.40	1.35	No
20	2370.21	1.16	2374.36	3.89	Yes

Table 4: Pov-ray experiment. Comparison between the execution time (s) for Scenarios 1 (μ_1) and 2 (μ_2) and hypothesis testing.

the same variance, we can use the Student t test [**?**]. Additionally, as there are several resources shared between containers, we can expect different behaviour for each one. In the following subsections, we accomplished a hypothesis test for applications with high CPU usage – Pov-Ray 3.7–, high I/O usage – IOzone benchmark – and high network usage –netperf.

3.1. CPU intensive application

We used the multi-threaded pov-ray 3.7 application as a benchmark to measure the overhead of pods for CPU intensive use. Kubernetes inherits from Docker the CPU quota reservation. This contingency mechanism allows a container to reserve a maximum CPU quota. However, the quota is only used when there is contingency in the machine, otherwise, all available CPU is used. The comparison between Scenarios 1 and 2 is presented in Table 4. We can see that when the number of containers increases, the null hypothesis should be rejected. Additionally, when H_0 is rejected, Scenario 1 is faster than Scenario 2. The overhead caused by having one container inside each pod is about 0.01%.

3.2. I/O intensive application

We used IOzone as a representative benchmark of an I/O application. Table 5 depicts the results (in seconds) for the execution of the iozone benchmark – the benchmark was executed as follows: *iozone -a -i 0 -i 1 -g 4M*. If we compare both scenarios, we can conclude that there is enough statistical evidence to accept H_0: As the number of pods in a machine increases, the caused overhead

	Scenario 1		Scenario 2		
C	μ_1	σ_1	μ_2	σ_2	$\mu_1 - \mu_2 = 0$?
1	23.52	0.82	23.19	0.64	Yes
4	60.85	1.45	65.02	1.25	No
8	85.98	2.25	91.36	2.24	No
12	108.54	4.14	91.36	3.40	No
20	153.51	6.47	170.99	5.28	No

Table 5: IOzone experiment (iozone -a -i 0 -i 1 -g 4M). Comparison between the execution time (s) for Scenario 1 (μ_1) and for Scenario 2 (μ_2) and hypothesis testing.

	Scenario 1			Scenario 2			
C	μ_1	σ_1	$\sum \frac{BW_i}{C}$	μ_2	σ_2	$\sum \frac{BW_i}{C}$	H_0?
1	1.88	0.06	1.88	1.90	0.04	1.90	Yes
4	8.61	0.21	2.15	8.82	0.05	2.20	Yes
8	15.53	0.12	1.94	16.26	0.20	2.03	No
12	14.99	0.21	1.25	16.42	0.38	1.37	No
20	15.10	0.19	0.75	18.32	0.91	0.91	No

Table 6: Hypothesis test for Network bandwidth (GB) for C iperf Clients. Iperf server & client are on the same machine.

is higher. The conclusion of these experiments is that it is better to group all the containers in the same pod.

3.3. Network intensive application

The network infrastructure of a machine is shared by all the containers inside a pod. All the containers in a pod share the port space and the pod has only one IP address. Sharing the access to the network by several containers might cause an overhead on the container performance. To measure that overhead, we deployed an `iperf` server inside a pod and several clients with the previous scenario configuration. All test measure the network bandwidth for a 30 second interval of TCP traffic.

The first experiment schedules all the containers at the same machine. In a real scenario, this situation might arise when the scheduler groups pods / containers together with high network traffic among them. In Table 6, we show the average bandwidth per container and the hypothesis test. When the number

	Scenario 1			Scenario 2			
C	μ_1	σ_1	$\sum \frac{BW_i}{C}$	μ_2	σ_2	$\sum \frac{BW_i}{C}$	H_0?
1	108.26	0.04	108.26	108.26	0.04	108.26	Yes
4	110.53	0.39	27.63	110.17	0.84	27.54	Yes
8	113.81	0.47	14.23	115.64	0.51	14.46	No
12	117.42	0.92	9.78	117.53	4.01	9.79	Yes
20	124.74	1.22	6.24	126.52	2.09	6.33	No

Table 7: Hypothesis test for Network bandwidth (MB) for C iperf Clients. Iperf Clients are on a different physical machine from the server one.

of containers inside the pod is above 4, there is enough statistical evidence to reject H_0. The best results are achieved when each pod has an isolate container (Scenario 2).

We repeated the experiments with the `iperf` server placed on a machine and the clients scheduled in another machine. Table 7 shows the results, which are similar to the previous ones. The bandwidth values from Scenario 2 are higher than the values from Scenario 1. From these experiments, we can conclude that deploying several pods with a few coupled containers is better than a single pod with a large number of containers.

4. Discussion

We demonstrated that the deployment of an application on a specific infrastructure can impact its overall performance. We used results from our experiments to derive rules that try to improve it. Since in Kubernetes the minimal schedulable unit is the pod, then ρ is the parameter which has the highest impact on performance. We assume that the Kubernetes nodes are homogeneous and that all the containers can be distributed across physical nodes, improving the performance of the application – i.e. there is no coupling between containers, and this is considered as a design restriction. Figure 6 summarises the rules to choose the best ρ from our experiments.

If an application is CPU or I/O intensive, it is better to group all the containers together –from experiments in Tables 4 and 5. However, we want to

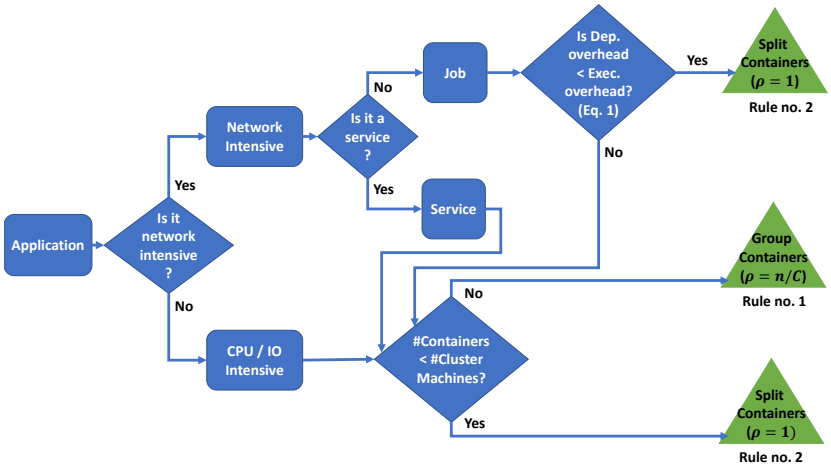

Figure 6: Flow diagram to choose the best ρ parameter. C is the number of containers to deploy and n is the number of machines in the cluster.

distribute the pods among as many machines as possible –through ρ parameter. If the number of containers is greater than the number of machines, ρ should be equal to $\frac{n}{c}$ (**Rule no. 1**) – we have n pods with $\frac{c}{n}$ containers at each pod. This rule tries to minimise the impact of T_d – which decreases for low values of ρ. If the number of containers to deploy is less than the number of machines, then $\rho = 1$ (**Rule no. 2.**) – we deploy a pod with a container at each machine.

The ρ choice is different if we consider an application that makes a high use of the network. If it is a service pod and there are few failures in the scenario –equivalently, T_d is negligible – the best choice is to set $\rho = 1$. The reason is that regardless of the machine where a pod is scheduled, the effective bandwidth is higher when there is only one container inside a pod (Tables 6 and 7). However, if T_d is relevant, we can calculate the total time T_t (deployment time T_d plus execution time T_e) as a function of ρ: $T_t(\rho) = T_d(\rho) + \alpha(\rho)T_e = \frac{c}{n}T_c(\rho) + \alpha(\rho)T_e$; where $\alpha(\rho)$ is the overhead caused by the pod abstraction (Section 3) and it can be calculated as $\frac{\mu_1}{\mu_2}$, where μ_2 corresponds to a scenario with $\frac{1}{\rho}$ containers per pod. In general, as μ_2 is expected to be greater than μ_1, then $\alpha > 1$. Additionally, $\alpha(1) = 1$. Figure 7 depicts an example of $T_c(\rho)$, calculated when

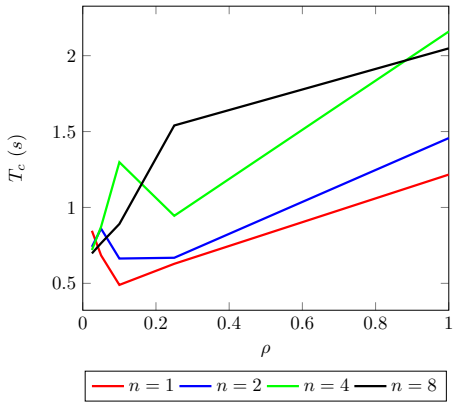

Figure 7: Function $T_t(\rho)$ for different values of n. The number of deployed containers is assumed to tend to infinity

$C \to \infty$, obtained from Figure 4.

It is a complex task to minimise the function $T_t(\rho)$. As a simplification, we can assume that $\alpha(\rho)$ remains constant and when n tends to infinity, the mean time to create a container also remains constant. In our experiments, for low values of ρ (Table 7), its value is about 1.01. Assuming that the major improvement in the execution time is achieved by executing tasks in parallel, we can compare the situation where $\rho = 1$ versus $\rho = \frac{n}{c}$. The first one will be faster than the second one when Equation 1 is satisfied – **Rule no. 2** should be applied. Otherwise, **Rule no. 1** will be more suitable.

$$T_t(1) < T_t(n/c) \implies \frac{c}{n}T_c(1) + T_e < \frac{c}{n}T_c(n/c) + \alpha T_e \implies$$

$$T_c(1) - T_c(n/c) < \frac{n(\alpha - 1)}{c}T_e \tag{1}$$

These rules are based on the experiments in Section 3. Other container technologies – such as Linux LXC or Core OS rocket – can be abstracted in a similar way. The use of a particular technology does not have an impact on our model, as many of these container framework will also share similar lifecycle states. However, the performance values may vary depending on the use of a particular container framework/ technology. In Section 5, we provide a comparison of the performance of different technologies. Additionally, there

Work	Model	Virtualisation infrastructure	Experimental framework
[?]	Experimental approach	VMs	Hyper-V, KVM, vSphere, Xen
[?]	Experimental approach	VMs	KVM, Xen, vBox
[? ?]	Experimental approach	Containers and VMs	Docker, KVM, Xen, LXC
[?]	Experimental approach	Containers and VMs	LXC, OpenVZ, VServer, Xen
[?]	Experimental approach	Containers	LXC
[? ?]	Experimental approach	Containers	Docker, KVM
[?]	Experimental approach	Containers	Docker, Weave
[? ?]	Continuous Markov Chains (Exponential PDF)	Containers over VMs	Docker Swarm over Amazon EC2
[?]	Nets within Nets (any PDF)	VMs	Simulations
Our work	Nets within Nets (any PDF) Experimental approach	Containers	Kubernetes over bare metal

Table 8: Summary of related work with the kind of model they proposed, the assumptions of the model, the virtualisation infrastructure that they used and the experimental framework.

are different container management systems such as Docker Swarm, or Apache Mesos. Although these other platforms do not have the pod abstraction, our models and results could be relevant to them in scenarios where $\rho = 1$.

In our work, we have proposed a methodology to feed the model and to analyse the overhead of pod abstraction. This methodology should be applied to different configurations to be generalised. For instance, all our experiments were carried out within a private cloud and Kubernetes was deployed over a *bare metal* system. This configuration allows us to avoid the additional overhead caused by the execution of Kubernetes inside VMs. On the other hand, the Google Cloud Engine platform gives the possibility of running a Kubernetes cluster; however, the containers are run over VMs, which may have an impact on the performance and the hypothesis tests may change. Besides, the underlying service architecture is different. For example, since the storage service is accesses through the cloud, the I/O intensive application will have a different behaviour, and the overhead caused by pod abstraction may not be negligible.

5. Related Work

We summarised the most important references in performance evaluation of cloud environments in Table 8. Most of them focused on performance comparison of VMs rather than on containers as the unit of computation. A set of workloads were developed to get the usage of memory, CPU, networking, and storage [**?** **?** **?**]. However, all of them are based on experimental results that do not have an analytical model that supports reasoning about performance.

To the best of our knowledge, no work has tackled container performance from a rigorous analytical perspective. Even in previous cloud technologies, few studies are based on formal models [**?**]. A Markov Chain based analytical model is used in [**?**] to study performance analysis of microservices and implemented with Docker and Docker Swarm. However, their analytical model assumes that the workload generation rate follows a Poisson distribution – the time between arrivals has an exponential probability density function (PDF) – which may yield to non-realistic scenarios. In contrast, we can link Petri Net transitions with any PDF or with functions obtained from real application benchmarking.

In [**?**], an iterative, and step-wise refinement methodology was proposed for cloud applications, covering all the software lifecycle steps. The methodology is centered around a performance model that captures (non) functional requirements, control flow, data flow and the involved computing resources. As a result the performance of the distributed system can be formally analysed by considering all the aspects that affect performance.

All these models need temporal information for feeding them. Previous studies are focused on obtaining this kind of information [**?** **?**]. Some others provide a performance comparison between VMs and containers [**?** **?** **?**], concluding that there is better performance and resource usage in containers. Scalability performance in Kubernetes was studied in [**?**], where the results show that containers perform 22x times faster than VMs for the provisioning action. Container platform evaluation, as Docker and LXC in [**?**], shows performance issues being improved and this approach being considered for High

Performance Computing (HPC). They conclude that the performance is near-native for both technologies. In [**?**], the authors present several variables (e.g. Linux kernel version) having an impact on the performance of containers. However, both studies focus on the execution time as a performance metric and they do not consider other measures.

As we described, Kubernetes introduces the pod abstraction. To the best of our knowledge, a performance analysis of this abstraction has not been conducted, but the analysis of nested containers is the closest research field. In [**? ?**], network performance degradation was observed in some configurations because of a deployment based on full nested containers. This degradation is caused by the usage of network virtualization technologies – Linux Bridge or OpenvSwicht – twice or by the usage of Software Defined Networks and encryptation. However, the Kubernetes pod abstraction gives a common space port to all containers – and therefore the same IP address to all services – so the performance degradation may be different.

Finally, several performance metrics of Kubernetes [7] were reported by its team. In a cluster of 100 nodes, their results show a 99th percentile pod startup time below 3 seconds, which is consistent with our results. Their experiments show how a Kubernetes cluster behaves when the scale of the deployment is increased by pod start time, end-to-end response time and response time of different API operations. However, the way the deployment is structured and the impact of grouping containers inside a pod were not analysed.

6. Conclusions and Future Work

An efficacious automated resource management in cloud computing requires to launch, terminate and maintain computing instances rapidly, with a minimum overhead. In this paper, we conduct performance analysis over Kubernetes, achieved through a Petri Net-based performance model. It allows us to analyse

[7] `http://blog.kubernetes.io/2016/03/1000-nodes-and-beyond-updates-to-Kubernetes-performance-and-scalability-in-`
`html`

deployment and termination overheads of containers in Kubernetes, as well as understanding the performance of different configurations of a Kubernetes pod – i.e. the influence of the number of containers per pod. We conducted our analysis in a Kubernetes cluster of 8 machines. Our model can be exploited as a basis to improve two activities: (i) capacity planning and resource management; (ii) application design, specifically how an application may be structured in terms of pods and containers. From our experiments, we can see that a single container can be deployed in a time interval than ranges from less than a second to up to 3 seconds, depending on the circumstances, – i.e. the number of pods per container, the number of containers deployed simultaneously, the network latency, or the number of host machines. In contrast, the termination time is typically in the order of a tenth of a second. Moreover, we also provide a set of rules that assist in allocating the number of containers per pod that provides the best performance. These rules consider a number of characteristics of the application, such as the usage of CPU or network.

As future work, we expect to study the resource contention phenomenon in containers, which appears when multiple containers compete for the same computational resource. We plan to exploit our model with different purposes: (i) to estimate the overhead introduced by resource contention in containers, (ii) to improve the Kubernetes scheduler so that it is aware of resource contention problems and (iii) to undertake various *what-if* scenarios to investigate the behaviour of different resource management policies.

Acknowledgments

This work was supported in part by: The Industry and Innovation department of the Aragonese Government and European Social Funds (COSMOS group, ref. T93) and the Spanish Ministry of Economy ("Programa de I+D+i Estatal de Investigación, Desarrollo e innovación Orientada a los Retos de la Sociedad" –TIN2013-40809-R). V. Medel was the recipient of a fellowship from the Spanish Ministry of Economy.