

# Impact of etcd deployment on Kubernetes, Istio, and application performance

Lars Larsson<sup>1</sup>  | William Tärneberg<sup>2</sup>  | Cristian Klein<sup>1</sup>  | Erik Elmroth<sup>1</sup>  | Maria Kihl<sup>2</sup> 

<sup>1</sup>Department of Computing Science, Umeå University, Umeå, Sweden

<sup>2</sup>Department of Electrical and Information Technology, Faculty of Engineering, Lund University, Lund, Sweden

## Correspondence

Lars Larsson, Department of Computing Science, Umeå University, 901 87 Umeå, Sweden.

Email: larsson@cs.umu.se

## Funding information

Excellence Center Linköping - Lund in Information Technology, Grant/Award Number: ELLIIT; Knut och Alice Wallenbergs Stiftelse, Grant/Award Number: Wallenberg AI, Autonomous Systems and Software Program; Stiftelsen för Strategisk Forskning, Grant/Award Number: SEC4FACTORY; Vetenskapsrådet, Grant/Award Number: eSSSENCE; VINNOVA, Grant/Award Number: 5G-PERFECTA Celtic Next

## Summary

This experience article describes lessons learned as we conducted experiments in a Kubernetes-based environment, the most notable of which was that the performance of both the Kubernetes control plane and the deployed application depends strongly and in unexpected ways on the performance of the etcd database. The article contains (a) detailed descriptions of how networking with and without Istio works in Kubernetes, based on the Flannel Container Networking Interface (CNI) provider in VXLAN mode with IP Virtual Server (IPVS)-backed Kubernetes Services, (b) a comprehensive discussion about how to conduct load and performance testing using a closed-loop workload generator, and (c) an open source experiment framework useful for executing experiments in a shared cloud environment and exploring the resulting data. It also shows that statistical analysis may reveal the data resulting from such experiments to be misleading even when careful preparations are made, and that nondeterministic behavior stemming from etcd can affect both the platform as a whole and the deployed application. Finally, it is demonstrated that using high-performance backing storage for etcd can reduce the occurrence of such nondeterministic behaviors by a statistically significant ( $P < .05$ ) margin. The implication of this experience article is that systems researchers studying the performance of applications deployed on Kubernetes cannot simply consider their specific application to be under test. Instead, the particularities of the underlying Kubernetes and cloud platform must be taken into account, in particular because their performance can impact that of etcd.

## KEYWORDS

cloud computing, distributed systems, etcd, Kubernetes, performance

## 1 | INTRODUCTION

The worldwide cloud Infrastructure as a Service (IaaS) market is worth \$32.4 billion<sup>1</sup> and containerization penetration is predicted to reach 75% by 2022.<sup>2</sup> Consequently, there is considerable academic and industrial interest in achieving

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

© 2020 The Authors. *Software: Practice and Experience* published by John Wiley & Sons, Ltd.

cost-efficient use of cloud and containerization technologies. Containers, in the context of this article, leverage features of the Linux operating system kernel that increase runtime isolation toward other processes on the same machine and operating system with less overhead than full virtual machines. This makes it possible to host several containerized processes on a single machine. However, doing so requires some orchestration, especially when operating at scale with multiple (virtual) host machines. Kubernetes has emerged as a de facto standard<sup>2</sup> for container orchestration because it simplifies deployment of containerized applications across a cluster of machines in a cloud provider-agnostic way.

Cloud-native applications rely on characteristics such as horizontal scaling and services offered by the underlying platforms, such as service discovery. There is growing interest in offloading application-level networking functionality such as traffic and security token management onto service meshes such as Istio, which are becoming critical elements of cloud-native infrastructure.<sup>3</sup>

Containerized platforms and the clouds upon which they are deployed are complex pieces of software, and complexity inevitably impacts performance. Research is therefore needed to identify and quantify sources of performance impact in these systems. To predictably achieve cost-effective deployments, it is necessary to understand the platforms underpinning containerization and the ways in which their characteristics affect their performance and ability to operate correctly as well as the performance of applications deployed on them.

In this article, we present experimental findings showing how underlying platform constitution and deployment can affect application performance. The original goal of this work was to quantify what effects, if any, the use of the Istio service mesh instead of native Kubernetes networking could have on the platform's ability to perform application autoscaling. Intuitively, if Istio provides smart traffic routing features and adds various request buffers, could these have an effect on autoscaling capability? We made thorough preparations to support this comparison by:

- conducting an in-depth exploration of the inner workings of our networking setup in Kubernetes with and without Istio, the outcome of which is presented in Section 2.1;
- developing a theoretically sound understanding of performance testing and load generation based on queuing theory for the general case and measurements of performance characteristics of the deployment under consideration (Section 2.2); and
- establishing an experimental framework for deploying tests in a repeatable manner that is resilient to errors that may occur due to frequent redeployment of software, and that randomizes the order in which experiments are run so that effects of momentary interference from noisy neighbors in the cloud are distributed across experimental runs and will not excessively impact any particular set of experiments (Section 2.3).

Despite these extensive preparations, the data obtained from a large number of experiments (over 1400 in total, including repetitions and iterations using different configuration parameters) were far too noisy to draw reliable conclusions about the original research goal. Essentially, we encountered a new case of “producing wrong data without doing anything obviously wrong.”<sup>4</sup> Our focus therefore shifted to identifying the source of the noise, and that is where the main contributions of this experience article lie.

Log file analysis showed that the etcd database experienced significant temporary slowdowns. Documentation indicates that etcd is I/O-bound, so we studied the effects of two different backing data stores for etcd with vastly different performance characteristics: the original slow network-backed storage and a very fast RAM-disk. These experiments showed that when etcd had sufficiently fast backing storage, the level of noise due to variation in application performance was significantly reduced. Unsurprisingly, faster I/O thus benefited the I/O-bound etcd process. More importantly, however, these results provided *new* knowledge by revealing that the performance of etcd affects application performance.

Like many research experiments conducted today,<sup>5-8</sup> we used a privately owned but shared cloud infrastructure to carry out our experiments.

This experience article presents detailed descriptions of how to prepare for and conduct performance tests, as well as an in-depth discussion of Kubernetes networking. However, its main contributions are:

- A set of experimental results showing the impact of etcd performance on the correct functioning of Kubernetes-internal functionality and how this gives rise to nondeterministic behaviors that in turn affect the performance of a deployed application (Section 5.1).
- An open data set comprising several thousand experiments and millions of measurements used to support the arguments made in the article, ready for further analysis by the research community.

Based on our findings, we argue that performance analyses of containerized cloud applications must take the underlying platform into account and treat the application and the platform *together* as the system under test.

This lesson has important implications for cloud researchers and industry practitioners alike, particularly in cases where the etcd database and Kubernetes control plane are offered together as a service by a cloud provider and are thus outside the cloud customer's control.

## 2 | BACKGROUND

In order to understand the technical concepts related to our original research question, this section presents material relating to both networking in Kubernetes-based environments (with and without Istio) and probes necessary for horizontal autoscaling. Then we discuss queuing theory-based experiment sizing. Finally, we introduce our experimental framework, which employs various techniques to avoid interference from other cloud users and simplifies execution of large-scale experiments.

### 2.1 | A networking-focused overview of Kubernetes

Kubernetes is an orchestrator for containerized applications. As such, it provides functionality required for configuring and deploying applications, automatically managing their life-cycles, service discovery, and managing storage. Of particular interest to us is how network traffic is routed both in native Kubernetes (Section 2.1.1) and with the service mesh Istio (Section 2.1.2). We are also interested in studying how the number of self-management orchestration tasks Kubernetes has to do affects application performance (Section 2.1.3).

Kubernetes clusters logically consist of sets of master and worker nodes. Master nodes contain the Application Program Interface (API) service, which performs authentication and authorization on a per-operation level. They communicate directly with an etcd database, a distributed key-value store. The master nodes also contain core components such as the scheduler and various Controllers, which automate tasks that (among other things) manage the life-cycle of deployed application instances, known as Pods. A Pod is an ephemeral encapsulation of one or more containers with a shared process namespace, file system, and network context. Communication between Pods is typically done via the Service abstraction, which provides service discovery and a longer life-cycle than Pods. A group of Pods backing a Service is usually managed by the Deployment Controller and referred to as a Deployment.

The set of Pods eligible to handle traffic to a Service are called the Endpoints of that Service. Kubernetes uses various probes to determine eligibility; see Section 2.1.3 for details.

The networking abstraction in Kubernetes is the Container Network Interface (CNI), which provides conceptually simple L2 or L3 networking functionality. For cloud-native applications that require and benefit from higher level functionality such as routing performed on L7 (application-level) values such as HTTP paths or header values, service meshes act as L7 overlay networks with these additional features. Istio is a service mesh that provides advanced traffic management and observability features as well as the ability to centrally determine security policies and rate limitations for the Pods in the mesh, among other things. Advanced functionality such as the ability to configure automatic conditional retries for requests (and more) has made Istio popular in the cloud-native community.<sup>3</sup>

How network traffic is routed to members of the Deployment depends on whether the native Kubernetes Service abstraction or the Istio service mesh is used. Our implementation-specific cluster setup of these two approaches is described in detail in Sections 2.1.1 and 2.1.2.

In our descriptions of network traffic flows, we assume that the Flannel CNI provider is used, and that it operates in Virtual Extensible LAN (VXLAN) mode. Furthermore, we assume that Kubernetes Services are exposed as NodePort type services. The use of NodePort-type services does not cause any loss of generality because even if a cloud provider offers a load balancing service, this is how Kubernetes will expose that service to such load balancers. Traffic to Services is proxied by kube-proxy, which we configured to use IP Virtual Server (IPVS) mode. The alternative iptables mode routes traffic via iptables rules that randomly choose among eligible endpoints, but is otherwise conceptually similar.

The descriptions in this section apply specifically to the combination of Flannel in VXLAN mode, NodePort-type services, and IPVS services. Descriptions of other implementation choices, such as Calico and its reliance upon border gateway protocol (BGP) for routing network traffic, would differ on a detailed technological level, but be conceptually similar. We therefore do not consider this description broadly applicable; rather, it is a detailed description of the specific

implementation we used. However, the use of Istio will add some overhead to the networking stack regardless of the chosen CNI implementation because Istio is added on top and no CNI provider at the time of experimentation offered native in-depth Istio integration.

The description is therefore primarily useful for the purpose of outlining the experiences presented in this article because it shows the additional steps required to interconnect Pods when Istio is used. It also shows that Istio relies on the Kubernetes control plane (and thus ultimately on etcd). This, as demonstrated by the experimental results presented below, has significant implications when the system is under load.

### 2.1.1 | Native Kubernetes deployment

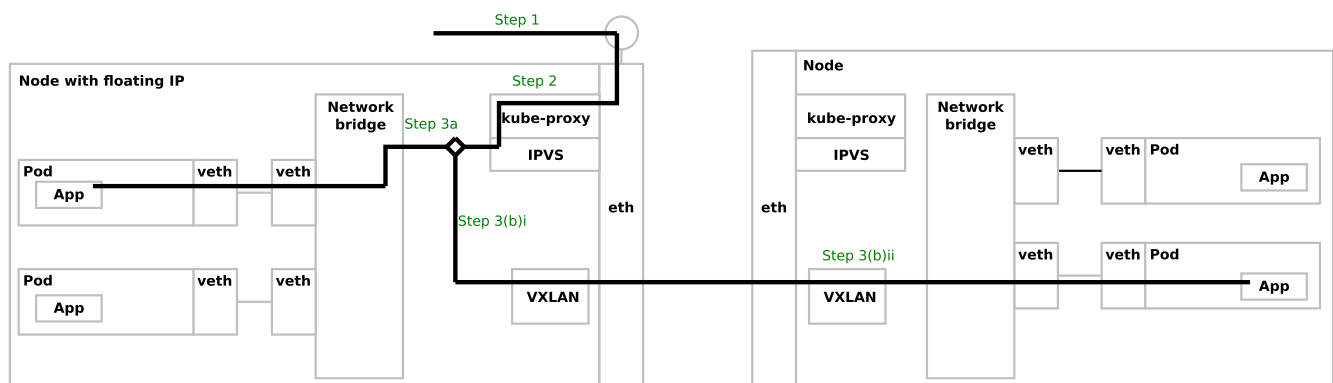
In native Kubernetes, incoming traffic flows as follows to reach a Pod (cf Figure 1):

1. The workers' traffic reaches the port specified in the NodePort Service. Any public IP handling via, for example, OpenStack floating IP or AWS Elastic IP is performed by Network Address Translation (NAT) by the cloud provider.
2. kube-proxy listens to the specified port and proxies packets to IPVS in the worker node. IPVS then chooses which Endpoint (ie, Pod) to forward the traffic to, according to its configured scheduler.
3. Local and remote traffic are treated differently:
  - (a) If the destination Pod is deployed locally, traffic is routed by IPVS onto a network bridge, where a set of paired virtual Ethernet adapters forward traffic from the host into the Pod.
  - (b) Alternatively, the destination Pod could be deployed on a different node. Thus, the CNI provider must ensure it reaches the target node.
    - i Flannel's VXLAN mode relies on handling in the Linux kernel, which essentially creates a UDP tunnel between nodes. Other CNI providers operate differently, and may even integrate natively with the virtual networking system used by the cloud provider.
    - ii The target node unpacks the VXLAN packets and its Flannel forwards them to the target Pod via a network bridge on the target node. Paired virtual Ethernet devices (veth) with one end on the node's network bridge and the other in the Pod ensure that the traffic reaches its destination.

Matters are kept relatively simple because Kubernetes' network model dictates that no NAT shall be required for node-to-Pod network communication. Thus, a network bridge is used instead. We see, however, that many steps are needed even in this case, which arguably represents the simplest way to expose a Kubernetes Service because it does not involve a separate load balancing service on top of the above.

### 2.1.2 | Istio-enabled deployment

The Istio service mesh is divided into two logical parts: a control plane and a data plane. Core Istio components manage, for example, configuration and security tokens on the control plane level, and configure the underlying data plane with



**FIGURE 1** Network traffic flow for a deployed application exposed using a native Kubernetes NodePort Service [Colour figure can be viewed at [wileyonlinelibrary.com](http://wileyonlinelibrary.com)]

user-specified policies. Traffic within the data plane flows between Envoy proxies, each deployed as sidecar containers in Pods alongside the main application containers. These proxies capture all traffic into and out of the Pod, making it controllable from the control plane.

Network traffic enters the Istio service mesh via an Istio *Ingress Gateway*. The Ingress Gateway is a Pod that is exposed externally via a Kubernetes Service. Upon receiving traffic, it uses configuration data from the Istio control plane to determine the correct destination within the service mesh. Such routing can be arbitrarily complex and based on IP addresses and endpoints (as in Kubernetes) or even application-level information such as HTTP headers and request paths.

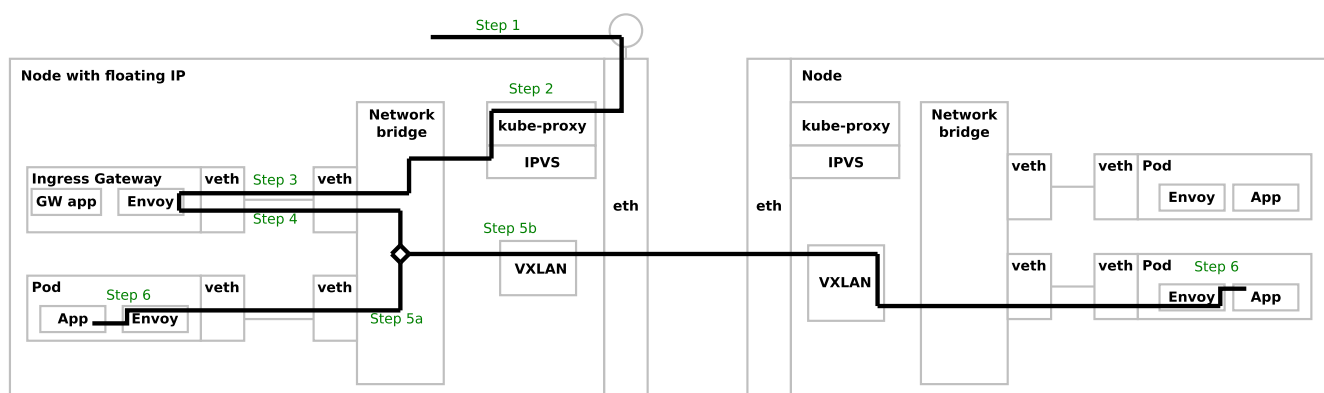
A feature of particular interest is that Istio can conditionally issue automatic retries in case of error, for example, only for HTTP status codes of 500 or above (which indicate server-side errors). Should such an error be encountered, the Envoy proxy in the Ingress Gateway can be configured to choose a different endpoint any number of times without returning a result to the worker. Error handling of this type is transparent from the worker's perspective, and is noticeable only as a prolonged waiting time ( $t^{\text{resp}}$ , cf Section 2.2) for a response.

The network traffic flow in an Istio-enabled deployment involves additional steps not required in native Kubernetes (cf Section 2.1.1). To increase efficiency when using Istio, we pinned the Ingress Gateway Pod to the node with floating IP and exposed it via Kubernetes NodePort Service. This ensured that at least the initial network hop would always be a local hop to the same node. The network traffic flow when using Istio is as follows (cf Figure 2):

1. As in Native Kubernetes Step 1, but the Service in question is the Istio Ingress Gateway.
2. As in Native Kubernetes Step 2, but the Service in question is the Istio Ingress Gateway.
3. As in Native Kubernetes Step 3a, but by design the Ingress Gateway Pod is known to be on the same host because it was pinned there using a Node Selector.
4. Envoy uses its configuration data (stemming from the Istio control plane) to determine the IP address of an application's service endpoint target Pod. The Envoy proxy sidecar container in the Ingress Gateway Pod has configured iptables to ensure that all Pod traffic is passed through the Envoy proxy.
5. Local and remote traffic are treated differently:
  - (a) If the Pod was deployed on the same node, traffic is routed locally to it via the virtual Ethernet adapters and local network bridge. Again, the Envoy proxy sidecar container is the initial recipient of all network traffic.
  - (b) The Pod that shall receive traffic may be deployed on a different node. If so, traffic is passed through a virtual Ethernet tunnel onto a local network bridge and then passed to the target node via VXLAN, where it is then unpacked, similarly to Native Kubernetes Steps 3(b)i and 3(b)ii. A difference is that the recipient is again the Envoy proxy sidecar container in the Pod, as dictated by iptables rules.
6. Regardless of how traffic reached it, the Envoy proxy sidecar container makes a localhost network call to the main application container.

As the list shows, including Istio in an application's deployment adds

- (a) an additional network hop and full network stack due to the Ingress Gateway, and
- (b) additional routing, network stack, and CPU processing due to the use of the Envoy proxy sidecar container.



**FIGURE 2** Network traffic flow for an application deployed using the Istio service mesh, where the Ingress Gateway service is exposed as a Kubernetes NodePort service [Colour figure can be viewed at [wileyonlinelibrary.com](http://wileyonlinelibrary.com)]



The possible benefits of Istio thus come at the cost of additional resource requirements. This is well known in the Istio community, and is discussed in their official documentation<sup>1</sup>. To our knowledge, there have been no studies on how this additional resource use affects nondeterministic behavior in the underlying Kubernetes platform or influences performance degradation.

To understand the experiments presented below and their results, it is important to know how Istio depends on the Kubernetes API. Istio self-configures its control plane by subscribing to update/change events emitted by the Kubernetes API for, for example, Pods, Services, and Endpoints. Istio is thus notified of updates first after the Kubernetes control plane has detected changes. Once that happens, a reconciliation loop must be triggered to ensure that the current state of the Kubernetes API objects is accurately reflected in that of Istio's control plane. This imposes a delay and makes responses to changes strongly dependent on the Kubernetes API (and therefore indirectly on the underlying etcd).

### 2.1.3 | Interplay of probes and resource limitations

Kubernetes uses probes to determine whether Pods are operational and able to handle incoming requests. There are three probe types that operate on a per-container level in a Pod: *readiness*, *liveness*, and *startup*. Whenever a container is ready to accept requests, it should respond positively to the readiness probe. Once all containers in a Pod do so, the Pod can be added as an Endpoint of a Service, and requests can be routed to it. The liveness probe verifies that the container is working as intended (eg, not in a deadlocked state). If the liveness probe fails a configured number of times, the container in the Pod will be restarted. Startup probes (not used in this work) hold off the other probes until successful, which may be useful if the other probes could interfere with the container's startup. In the worst case, such interference could incorrectly mark it as failing liveness tests and thus cause a recurring initialization and termination loop.

While probes operate on a per-container level, readers should note that overall Pod readiness status requires all containers to be ready.

Should one or more container readiness probes fail, the Pod is no longer considered a valid Endpoint of any Service it belongs to, and should therefore no longer receive traffic. This implies that there is a time window where a Pod may be in a state where it is unable to handle traffic but Kubernetes has not yet probed it. In such cases, the HTTP-level response will be 503 Bad Gateway.

Aside from application failure, one reason why a container may fail to respond affirmatively to its readiness probe is that it has been terminated by Kubernetes. Kubernetes offers the ability to limit the amount of RAM and CPU used by containers in Pods. Violating the memory limitation by allocating more results in immediate termination by the Linux Out-Of-Memory (OOM) Killer process. By contrast, violating the CPU limit is impossible because the cgroup functionality in the kernel limits the amount of CPU time a process gets scheduled for. When memory violations occur, corrective actions such as restarting the container must follow. We can therefore increase the number of corrective actions that Kubernetes can perform by modifying application deployment such that an application will violate its RAM limit under high load.

Although killed and until operational, the container will also fail its readiness and liveness probes, which should affect network traffic routing. If the routing is not updated in a timely fashion, the errors mentioned above should occur and be noticeable by workers or load generators. Note that we do not directly inject faults (see, for example, the work of Natella et al<sup>9</sup>). Rather, by increasing the likelihood of containers getting killed, we can study how the underlying Kubernetes platform (with and without Istio) operates under extreme conditions and how that affects applications deployed onto it.

## 2.2 | Performance and load dimensioning

A model representation of a system under test can be a very useful aid to reasoning.

In this section, we briefly describe performance testing by subjecting a networked application to requests generated by a set of workload generators. Additionally, a simple set of queuing theory tools for reasoning about how to dimension and load such a system is presented.

Workload generators are commonly classified as operating in either *open-* or *closed-loop* fashion.<sup>10</sup> An open-loop workload generator initiates requests in a system at a given volume, regardless of job completion or response times. By

<sup>1</sup><https://istio.io/docs/concepts/performance-and-scalability/>, accessed 22 October 2019.

contrast, a closed-loop workload generator waits for completion of a given request before issuing the next one. We use a closed-loop workload generator in this work for two reasons. First, it matches the original intended use of the application examined in our experiments (Section 3.2). Second, as shown in the remainder of this section, it allows us to use formal models from queuing theory to reason about the load that the application is subjected to.<sup>11</sup> By construction, closed-loop workload generators cannot significantly overload the system under test, because load generators will wait until they have received a response before issuing a new request. However, precisely this property makes them useful in determining the maximum service handling capacity of the system under test. In the section that follows, we will briefly review the formal mathematical model that gives us this reasoning power.

By contrast, open-loop workload generators run the risk of resource exhaustion if the targeted system becomes overloaded: the workload generator will have to keep file handles, network connections, and threads open until responses arrive. And if there are hundreds or thousands requests per second and an overloaded system can take many seconds to respond, the bottleneck may turn out to be the workload generator, rather than the targeted system. As our intention is to precisely subject our targeted system with given load levels, this risk is unacceptable to us. For a more detailed discussion of the merits of closed vs open loop (and a half-open hybrid thereof), readers should read the work of Schroeder et al.<sup>12</sup>

On a high level, the system under test consists of a Kubernetes cluster (here denoted as k8s) with a set of nodes,  $\mathcal{N}^{\text{k8s}}$ , and a separate JMeter load generator cluster with a set of worker nodes,  $\mathcal{N}^{\text{load}}$ . An application is hosted on the Kubernetes nodes and consists of a set of Pods,  $\mathcal{P}$ . These sets are defined in Equations (1)-(4).

$$\mathcal{N}^{\text{k8s}} = \{n_i^{\text{k8s}} | i = 1, 2, \dots, I\}, \quad (1)$$

$$\mathcal{P} = \{p_j | j = 1, 2, \dots, J\}, \quad (2)$$

$$\mathcal{N}^{\text{load}} = \{n_k^{\text{load}} | k = 1, 2, \dots, K\}, \quad (3)$$

$$\mathcal{W} = \{w_l | l = 1, 2, \dots, L\}. \quad (4)$$

The load-generating workers in the set  $\mathcal{W}$  are identical agents that operate in a closed loop and continuously generate HTTP requests. In a closed-loop system, a worker  $w_l$  sends a request and then waits for a response of duration  $t^{\text{resp}} = t^{\text{exec}} + t^{\text{rtt}} + Z$  or a time-out  $t^{\text{time-out}}$  before sending the next request.<sup>12</sup> Note that  $t^{\text{time-out}}$  is set in the load generator. Here  $t^{\text{exec}}$  and  $t^{\text{rtt}}$  are the expected Pod execution time and the round-trip-time between the load generator nodes  $\mathcal{N}^{\text{load}}$  and the cluster nodes  $\mathcal{N}^{\text{k8s}}$ , respectively.  $Z$  is uncharacterized noise. Furthermore, a worker  $w_l$  generates HTTP requests at a rate of  $\frac{1}{\min(t^{\text{resp}}, t^{\text{time-out}}) + t^{\text{delay}}}$ , where  $t^{\text{delay}}$  is a user-configurable parameter. Decreasing  $t^{\text{delay}}$  increases the request rate. The total request rate  $\lambda$  imposed upon  $\mathcal{P}$  is thus:

$$\lambda = \sum_{w_l} \frac{1}{\min(t^{\text{resp}}, t^{\text{time-out}}) + t^{\text{delay}}}. \quad (5)$$

A Pod  $p_j$  has an expected service rate (HTTP responses/sec) of  $\mu_j = \frac{1}{t^{\text{exec}} + t^{\text{rtt}} + Z}$ . The service rate for the whole system is therefore  $\mu = \sum_j \mu_j$ .

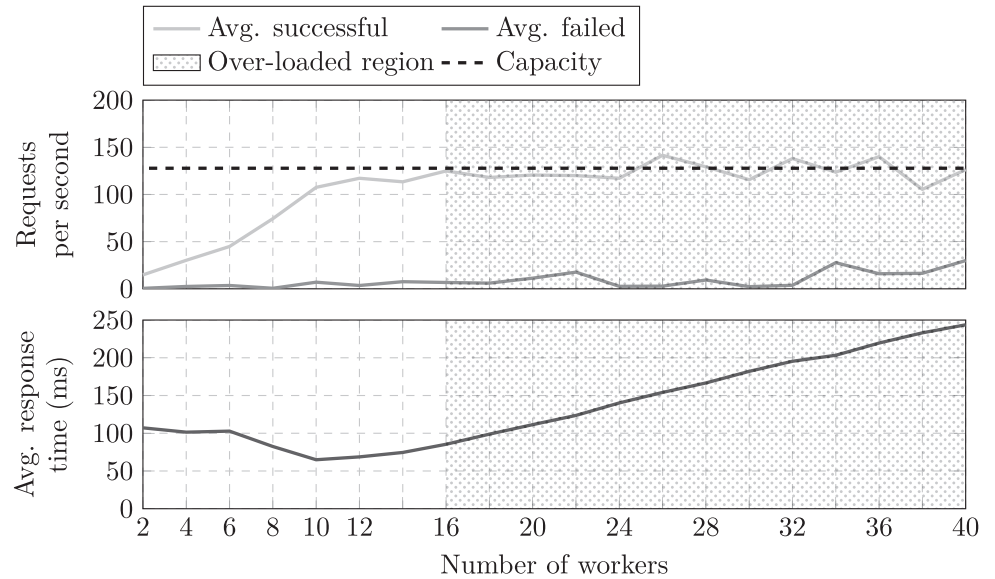
The utilization of  $\mathcal{P}$  is defined as  $\rho = \lambda / \mu$ . Note however that since this is a closed-loop system, the value of  $\rho$  is only valid when  $< 1$ .<sup>13</sup> As such the reasoning in this section will enable us to determine *how* to overload the system but not to *what* extent.

We assume that the set of Pods  $\mathcal{P}$  in the system are Resource Sharing (RS) and each have a finite queue of an unknown size. Here, the set of shared resources consists of memory, CPU, and I/O.

In a closed-loop system, at  $\rho \approx 1$ ,  $t^{\text{resp}}$  will increase and thus the  $\lambda$  will decrease. As more Pods are added,  $t^{\text{resp}}$  will reseed and  $\mu$  increase. At some point the noise of the system and the capacity of the network will diminish the return for each additional Pod. As such the reasoning in this section will enable us to determine *how* to overload the system but not to *what* extent. In essence, the upper performance bound of the system at the current number of Pods.

We assume that the set of Pods  $\mathcal{P}$  in the system each have a finite queue of an unknown size. A Pod  $p_j$  represents a job or a process. It is assumed that resources are dynamically managed and scheduled in each node  $n_i^{\text{k8s}}$ . Additionally, even when given the opportunity, applications in such systems do not necessarily consume all resources. Therefore, the service rate  $\mu_j$  for each Pod  $p_j$  does not necessarily vary linearly with resource availability.<sup>14</sup> Consequently, the system is represented by a

**FIGURE 3** The observed performance of 20 Pods for numbers of workers in the range  $[2,40]$  when  $t^{\text{time-out}} = \infty$



closed  $\langle T/M/M/c \rangle$  – RS-type queuing-system, where  $c = J$  and  $T = L$ . The Processor Sharing (PS)  $\langle T/M/M/c \rangle$  – PS model presented in Reference 11 can be seen as a basis on which to begin estimating the properties of such a system. However, the actual service rate  $\mu^*$  and actual request rate  $\lambda^*$  will be less than their theoretical counterparts,  $\mu^* \leq \mu$ ,  $\lambda^* \leq \lambda$ .

### 2.2.1 | Dimensioning the system: an example

There are two fundamental factors that must be determined when dimensioning an experimental setup: the *capacity* of your system  $\mu^*$  and *to what degree the system is subject to the intended load*  $\lambda^*$ . We use the data presented in Figure 3 as a basis for reasoning about these parameters. In the scenario<sup>2</sup> depicted in Figure 3,  $I = 6$ ,  $J = 20$ , and  $t^{\text{delay}} = 20$  milliseconds. Furthermore,  $L$  is varied in  $[2,40]$ , in steps of two, with seven repetitions for each value of  $L$ . Note that the objective is to load the system to  $\rho \geq 1$ . The duration of each experiment is 20 minutes, with seven repetitions.

The basic performance assumptions are as follows. Queuing theory tells us that the successful response rate should plateau at the  $L$  where  $\rho = 1$ . Around that point, the requests will begin to be either queued or rejected. As requests are queued, the mean waiting time will be greater than  $\min(t^{\text{resp}}, t^{\text{time-out}})$ . This can be seen in the lower half of Figure 3, where the waiting time strictly increases with  $L$  when  $L > 10$ . Thus, as shown in the upper half of the figure, the system of 20 Pods cannot produce more than an average of 127.8 responses per second.

The best case mean response rate can be estimated from the response time chart in Figure 3. The lowest achieved response time ( $t^{\text{resp}}$ ) is 63 milliseconds. Analogously, each worker  $w_i$  can generate  $\approx 12$  requests per second at most. To achieve full utilization ( $\rho = 1$ ) of the 20 Pods ( $I = 20$ ), at least 16 workers ( $L = 16$ ) are needed, assuming  $Z = 0$ . As this is a closed system,  $t^{\text{resp}}$  is representative of the service time for a Pod when  $\rho \leq 1$ . The expected service rate per Pod is then  $\approx 16$  responses per second. Note that the dip can be attributed to node-level scheduling prioritization decisions. Again, the system under test does not behave in strict accordance with theory. However,  $\rho = 1$  occurs somewhere in  $L = [10,16]$  and is guaranteed at  $L > 16$ . This discrepancy is due to noise ( $Z$ ) from the policy events, execution scheduling, and difficult-to-predict circumstances. Nevertheless, using the queuing model above allowed us to reason about the system's properties and estimate its performance bounds. Additionally, in most scenarios we want to operate the system in the overloaded region  $\rho \geq 1$ , which we can now identify.

## 2.3 | Experiment framework

When studying real-world application performance, it is difficult if not impossible to know in advance *what* to look for. A good data collection and experiment framework must therefore yield results amenable to postexperiment data exploration.

<sup>2</sup>Scenarios prefixed loadgeneration-reduced-workers in the data set.



In this section, we present the framework we have developed and used to conduct large numbers of experiments running in real time over the course of several months with minimal operator intervention.

Nondeterministic behaviors of Kubernetes clusters manifest themselves in all aspects of practical experimentation and their effects are not necessarily limited to impacts on application performance; they may also impact the ability of the cluster to function at all, for instance by causing software, node, or network failures. Experimentation in a Kubernetes environment thus requires rigorously controlled processes to ensure repeatability and reproducibility (as defined by Feitelson<sup>15</sup>), lest results from one experiment affect another. Cloud environments in general are time-shared systems with high variance, so cloud performance testing has a particularly strong requirement for repeated experiments and statistical analysis.<sup>16</sup> By necessity, the data sets resulting from such experiments are very large.

We therefore developed an experimental framework with the goals of:

1. ensuring experiment repeatability and reproducibility;
2. maintaining control over processes before, during, and after experiments;
3. enabling data collection from various sources; and
4. facilitating data exploration.

This section outlines the design and implementation choices made when developing this framework and explains how they relate to the stated goals.

### 2.3.1 | Architecture and components

The main infrastructural components of our setup are:

- (a) the Kubernetes cluster, upon which the system under test is deployed;
- (b) the separate load generation cluster, which issues requests toward the system under test and emits test results; and
- (c) the experiment framework *Control Node*, which configures, controls, and collects data from experiments.

Our experiment framework makes no assumption about how Kubernetes is deployed, but does require a way to programmatically reboot nodes for mitigation if they are marked as “Not Healthy” by Kubernetes.

This functionality is offered by the command line interfaces to OpenStack and by all other popular cloud infrastructure as a service providers.

Load is generated using the popular Apache JMeter suite. Apache JMeter is deployed as a distributed system, with each node in the load generation cluster running JMeter in slave mode. The load generation cluster is a set of Virtual Machines (VMs), possibly running in the same cloud as the Kubernetes cluster. Crucially, the load generation cluster is *not* running in or on a Kubernetes cluster. Instead, it uses minimal VMs to prevent resource contention from affecting the test results.

The Control Node acts as a one-stop shop for defining, deploying, and tearing down experiments in a controlled way. It also stores all data and metadata related to the experiments. The following components are deployed onto it to enable its operation:

- our software for controlling the Kubernetes and load generation clusters (Design Goal 2) and recording metadata about the experiments (Design Goal 1);
- cloud-specific tooling to allow rebooting of VMs, should the need arise (Design Goal 2);
- the JMeter master process, which issues commands to the load generation cluster slaves (Design Goal 2);
- data gathering software, which both queries a Kubernetes-deployed Prometheus server and parses JMeter result files (Design Goal 3);
- a PostgreSQL database to store all data (Design Goal 4).

The separation of functions and isolation of the system under test from the system that performs the tests make this environment suitable for the testing process as described by Edwards.<sup>17</sup>

### 2.3.2 | Repeatability and reproducibility

Since performance experiments using cloud software involve many components with nondeterministic behaviors, several repeated experiments with fixed parameters are required. In our work and experimental framework, we refer to specific combinations of parameter settings as *scenarios*. A particular scenario might, for instance, stipulate that autoscaling should be used for deployment sizes between 10 and 20, and that 120 workers should generate load according to a specific JMeter Test Plan.

For each scenario, the researchers define a number of *experiments*. Each experiment has a unique ID and relates to a particular scenario. All measurements and metadata will be associated with the experiment they belong to. We keep track of the status of the experiment, and it will not be marked as *finished* until the Control Node can assert that all postconditions have been fulfilled. The most important postcondition is that all data and metadata have been gathered, parsed, and recorded successfully.

If unrecoverable errors occur during an experiment, the experiment will not be marked as finished. The Control Node will then ensure that the experiment is reattempted at a later time.

The source code of the application under test and the experiment framework themselves are both stored in version controlled Git repositories. Since code modifications to either may occur during the experimental phase of the research process, the Git commit hashes of each repository are included in the stored metadata for each experiment. There is thus a permanent record showing which experiments were performed with which versions of all relevant software, and the researcher can make statistically verifiable judgment calls concerning the impact of any version changes on the collected data. This also helps with reproducibility (Design Goal 1).

To mitigate the possibility that noisy neighbors<sup>18</sup> might affect too many experiments belonging to a single scenario, our experimental framework chooses experiments such that scenarios with few completed experiments are prioritized. If, for instance, the entire suite of scenarios consists of scenarios A-H, and there are four finished experiments for scenarios A-D, but only three for E-H, an experiment from the E-H set will be selected. Naively running (for example) all 10 experiments of scenario A followed by all 10 experiments of scenario B would run the risk of some noisy neighbor problem affecting a large portion of experiments of any one particular scenario.

Randomization of this sort is commonly applied in the field of statistics. In addition to the aforementioned best effort attempt at mitigating transient problems due to noisy neighbors, it has the benefit of quickly giving the researcher at least some data on all scenarios and then performing additional replicate experiments. An overview is thus quickly generated based on experiments from all scenarios. Then, as time goes on, more experiments provide additional data.

It should be noted that while these experiments were conducted on a private cloud, it was a private cloud shared with a very active research and development department, so no resources were dedicated to our exclusive use. Nor did we enjoy the privilege of additional “under the hood” insight that a private cloud can offer, due to a lack of administrative access and permissions. The noisy neighbor mitigation strategies employed and described here were simple ones that we as users of the cloud could apply without insight into the overall state of the cloud infrastructure.

### 2.3.3 | Controlling experiment processes

Controlling experimental processes is key to ensuring that tests are repeatable and results are reproducible by other researchers. To that end, our experiment framework:

- refuses to continue unless Git reports no uncommitted changes to the working directories of both the system under test and the experiment framework itself;
- ensures that a RAM disk is used to store intermittent JMeter results, to avoid additional I/O delays as much as possible;
- asserts that all Kubernetes nodes are reporting in as “Healthy,” restarting them via the cloud’s API if this is not the case;
- ensures clean process states and a lack of Java Virtual Machine (JVM) garbage by restarting all JMeter slave processes in the load generation cluster;
- asserts that Prometheus is accessible, because it will hold performance measurements from the experiment;
- asserts that the PostgreSQL database is working and able to store data; and

- removes all traces of the system under test from the Kubernetes cluster, including supporting services such as the Istio service mesh, and any stored data relating to previous experimental runs to minimize result contamination between experiments.<sup>4</sup>

Once these preconditions have been satisfied and the relevant processes have been conducted, the environment is ready to deploy the application under test for the experiment. When that point is reached, metadata such as the starting time are recorded, the application under test is deployed, and the load generation cluster becomes controlled by the JMeter master process running in the Control Node.

### 2.3.4 | Data collection

Data are collected from two main sources: the load generation cluster reports how the system under test behaves from the perspective of an external observer, and a Prometheus system deployed in the Kubernetes cluster reports on the internally observable behavior.

The load generation cluster uses JMeter, which is deployed in distributed mode, using the Control Node as the master of the JMeter cluster. As previously mentioned, the master process stores all runtime measurement results on a RAM disk to prevent needless disk I/O slowing down the reporting and subsequent load generation by the cluster of slaves. This is a safe and reasonable approach because a reboot of the Control Node during testing is treated as an unrecoverable error that requires the experiment to be rerun, as per Section 2.3.2.

The JMeter master configures its slave nodes to buffer large numbers of data points to reduce additional network traffic overhead when reporting results to the master node. Such reporting is costly in terms of network resources, which are arguably among the most precious resources one would not want to waste during a load generation test. Because we use a closed-loop workload generator, we want there to be both as little additional uncharacterized noise in our measurements as possible, and that the mean interrequest delays should match the Poisson distribution as well as possible. Thus, buffering reporting information avoids needlessly adding traffic to the network.

JMeter operates by using a Test Plan encoded as a file that specifies all aspects of how a single slave should perform a test. These files typically specify delays between requests, request rates, and simulated user behavior. The total number of workers (cf Section 2.2) is divided equally among JMeter slaves. For performance evaluation purposes, the test plan can be arbitrarily complex but it must be finite: it completes after a predetermined number of seconds or issued requests.

The Prometheus database is installed on the Kubernetes cluster and configured using the Prometheus Operator Helm Chart. Prometheus is a monitoring solution and database that is enjoying widespread adoption in the Kubernetes community. It operates in a pull fashion, where a master process pulls data from all its configured data sources. The Prometheus Operator includes such data sources and can be configured to expose crucial metrics regarding node and Kubernetes Pod resource usage. Istio can also function as a pull source.

When an experiment is over, the experiment framework records the end time in the experiment's metadata in the PostgreSQL database. All requested metrics are then parsed from both the JMeter cluster and Prometheus. Metric data are recorded with a resolution of 1 second, calculated by Prometheus as rates based on data it pulls every 15 seconds. All measurements are parsed using purpose-built parsers and subsequently stored in the PostgreSQL database. To facilitate comparisons between experiments, we normalize timestamps by recording them relative to the start of the experiment, which is thus assigned a timestamp of 0 seconds. Actual wall-clock times for specific points in the experiment can be determined by simply adding the corresponding timestamp to the recorded start time.

The clocks of all virtual machines were synchronized using the Network Time Protocol (NTP) and differences in timestamps should therefore be minimal.

The data collection procedure described in this section imposes a minimal operational overhead cost. Container-level runtime metrics are pulled from containers by the Kubernetes kubelet process using the cAdvisor software, which has low overhead.<sup>19,20</sup> Prometheus pulls this data from the kubelet every 15 seconds, which should have a negligible performance impact on the overall system when amortized over experiments lasting 20 to 30 minutes. Results for the entire experiment are queried from Prometheus only after the experiment is finished.

Because JMeter was deployed on separate virtual machines and operates in closed-loop fashion (Section 2.2), the internal processes of these virtual machines should not be able to affect the performance of the application under test. JMeter was also configured to hold off as much of internal communication and state reporting as possible until the experiment

had finished (control messages from the master to the slaves are unavoidable, considering how JMeter implements its distributed testing mechanism).

### 2.3.5 | Data exploration

Large sets of data are generated when experiments are repeated and wide parameter spaces are explored, particularly when metrics are recorded at a resolution of 1 second. Understanding such data sets requires iteration and the ability to test hypotheses not known at the time of data collection. This is best done using data exploration-centered workflows similar to those adopted by data scientists, which often rely on tools such as Jupyter notebooks.

Our experiment framework stores all data and metadata related to scenarios and experiments in a PostgreSQL database. This facilitates integration with tools like Jupyter notebooks as well as data analysis libraries such as NumPy and pandas. By storing data in a relational database rather than in, say, a large set of CSV files, we enable rapid exploration and the ability to ask new queries without having to create single-purpose scripts that parse CSV files *en masse*.

The database schema is normalized such that measurements belong to experiments, which belong to scenarios. This makes it trivial to, for example, obtain the maximum number of requests served per second in a particular scenario by using the appropriate SQL JOIN statements.

## 3 | EXPERIMENTAL SETUP

In this article, we study the effects of the constitution and deployment of the underlying Kubernetes (and Istio) platform on application performance.

The application under test must be *simple* enough that interesting behavior can be correctly attributed to the surrounding environment into which it is deployed, yet sufficiently *complicated* that it consumes a realistic set of computational resources.

### 3.1 | Kubernetes cluster and cloud infrastructure

Motivated primarily by lower operational expenses, we used a private OpenStack-based cloud for our experiments.

This particular private cloud was built for research purposes with a heavy focus on fault-tolerance. It was therefore designed such that all disk access is actually backed by a Ceph cluster of redundant network-attached storage nodes. Consequently, it has no truly local instance storage. This property can have a significant impact and, as shown in Section 5, is the cause of some nondeterministic behavior. However, as Section 5 also shows, this nondeterministic behavior provides valuable insight into the relationship between the performance of Kubernetes and that of deployed applications.

Kubespray was used to install a version of Kubernetes (1.12.6) that was stable when experimentation began in early 2019. The cluster consisted of:

- One master node with a floating IP attached, 8GiB of RAM, and four VCPUs. The master node is a dedicated master (does not permit application Pods to be scheduled), and also includes the etcd database.
- One worker node with a floating IP attached, 4GiB of RAM, and two VCPUs.
- 5 worker nodes without floating IPs attached, with the same amount of resources as the other worker.

For networking, we used the Flannel CNI provider, and the Kubernetes cluster was configured to use the IPVS implementation of Services; according to the official documentation, this implementation provides higher performance than the iptables-powered one. Our setup closely resembles that shown in Figures 1 and 2 in that it does not use a cloud-provided load balancing service. This corresponds to the state of the art for OpenStack: as of the time of writing, only two of 22 public providers offer support for the Octavia load balancing service<sup>3</sup>. We used version 1.1.5 of the Istio service mesh.

<sup>3</sup><https://www.openstack.org/marketplace/public-clouds/>, accessed 15 October 2019.

### 3.2 | Application under test

The application under test in our experiments is an intentionally simple single-tier stateless function that performs an image manipulation task. A request to the application specifies an image and a rectangular cropping area defined by four coordinates. The application crops the image to the specified rectangular area, encodes the response in a Base64-encoded PNG image representation, and returns the resulting data via a HTTP response.

The application is *simple* in the sense that a single-tier application does not have multiple layers of intercomponent communication over a network, and is therefore not delayed by, for example, blocked upstream network connections. This implies that any nondeterministic behavior exhibited by the application should largely be due to the Kubernetes and cloud infrastructure. However, it is sufficiently *complicated* because it consumes a realistic amount of I/O (loading a specified image), CPU resources (selecting the rectangle and encoding the result), and memory (holding the image and intermediary results during processing), in addition to, of course, servicing requests over the network (I/O).

In accordance with common practices for web applications, we use the Flask web framework for Python 3 to serve over HTTP. Furthermore, image manipulation is done using pillow and Base64 encoding by the Python 3 standard library. The application is packaged as a Docker image and, to simplify testing, includes a predefined set of images. The application has been made fully open source and is available online.

We chose to use this single application for our experiments because the only desirable properties of the application were those described: a combination of sufficient simplicity and complexity. Recalling that the original research goal was to investigate which effects, if any, the use of Istio rather than native Kubernetes networking could have on the platform's ability to perform autoscaling, the application fulfills fundamental needs: it is both easy to reason about and to overload. It will consume a realistic amount of CPU cycles for an image processing task, and its use of memory will make it an easy target for the Out-Of-Memory killer process in the Linux kernel. Thus, under high load (when autoscaling would be most beneficial), the application is at a significantly heightened risk of exceeding memory limitations and therefore triggering Kubernetes self-healing processes, informed by the aforementioned probes. Although the test application mimics a realistic one, it is used primarily to help understand the behavior of the underlying Kubernetes and Istio networking systems (and, as it turned out, the effects of the etcd deployment on the platform and application). Therefore, as long as the desired properties are present, the choice of application is largely inconsequential.

#### 3.2.1 | Application deployment

The application is deployed as a Kubernetes Deployment. To make it possible to provoke errors and corrective actions by Kubernetes, the Pod specification in the Deployment includes resource limitations. These limitations and their intended effects during load testing are described in more detail in Section 2.1.3. Resource limitations are also required for the subset of scenarios that use the Horizontal Pod Autoscaler (HPA) to scale the size of the Deployment based on CPU load.

Recalling Section 2.1.3, we aim to increase the likelihood of corrective actions being required by Kubernetes. Thus, in our experiments, we limit the amount of RAM allocated to our main application container. Under low load, the limit should have no effect, but under high load, the Python garbage collector will not be fast enough to deallocate memory used to handle previous requests, and the process will eventually use more memory than it is allowed. This limit is obviously application-dependent; in our case, 64MiB of RAM was experimentally determined to provide the desired effects. Rather than deterministically terminating containers, for example, after X requests or after X seconds of CPU load over a given threshold, we chose this approach because we do not want to control any nondeterministic behaviors and thereby risk causing them to disappear. Instead, we want to study them as they occur.

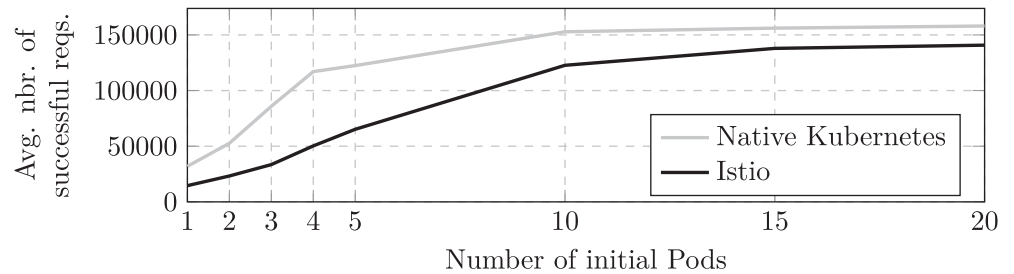
Our application, when deployed on top of Istio, defines an Istio *Virtual Service* that includes an application-level routing rule. Additionally, our application requires Istio to issue automatic retries in case of error if the error is of HTTP status 500 or above (cf Section 2.1.2).

## 4 | EXPERIMENTS COMPARING ISTIO AND NATIVE KUBERNETES TRAFFIC ROUTING

The original intent of our research was to compare Istio to native Kubernetes networking with respect to both the performance achieved and other effects that might arise. To that end, we conducted a suite of 1400 experiments spanning 320



**FIGURE 4** Average number of successfully handled requests as a function of the number of initial Pods in an autoscaled suite of experiments



scenarios<sup>4</sup>, in which we varied the numbers of Pod instances  $J$  and workers  $L$ , as well as the network deployment (native Kubernetes or Istio service mesh).

Of particular interest were effects on the ability to correctly calculate load for autoscaling purposes, and to make required scaling adjustments. Preliminary experimental findings obtained before starting this work suggested a link between use of Istio and improved ability to perform autoscaling under load. This seemed counterintuitive to us and warranted further investigation. Use of Istio should make raw networking performance worse, given that strictly speaking more work is required for a request to reach its destination when Istio is used (see Sections 2.1.1 and 2.1.2). But could buffers or smarter traffic routing in Istio provide an overall improvement anyway?

## 4.1 | An experiment on Istio's performance

The load generation cluster is configured to generate a consistent load level in a closed-loop manner, as defined in Section 2.2. The number of *initial* Pods used to start each scenario for autoscaling varies in the range ( $J \in [1, 2, 3, 4, 5, 10, 15, 20]$ ). To reduce the overall running time of the entire suite of experiments and because we expect the successful request rate to plateau with larger numbers of initial Pods, the experiment's resolution is reduced when  $J > 5$ . The upper limit for the HPA is fixed at  $\hat{J} = 20$  Pods for all scenarios. Our scenarios also define a varying number of workers that generate load, ranging from  $K \in [1, \dots, 20]$  per JMeter slave. With six JMeter slaves, the total number of workers  $L$  is thus 6, 12,  $\dots$ , 120. The workers are configured with  $t^{\text{delay}} = 20\text{ milliseconds}$ , where  $t^{\text{delay}}$  is the mean of a Poisson process, and  $t^{\text{time-out}} = \infty$ . In accordance with Figure 3, the experiments encompass both underloaded ( $\mu < 1$ ) and overloaded ( $\mu \geq 1$ ) regions. The duration of each experiment is 20 minutes.

### 4.1.1 | Results

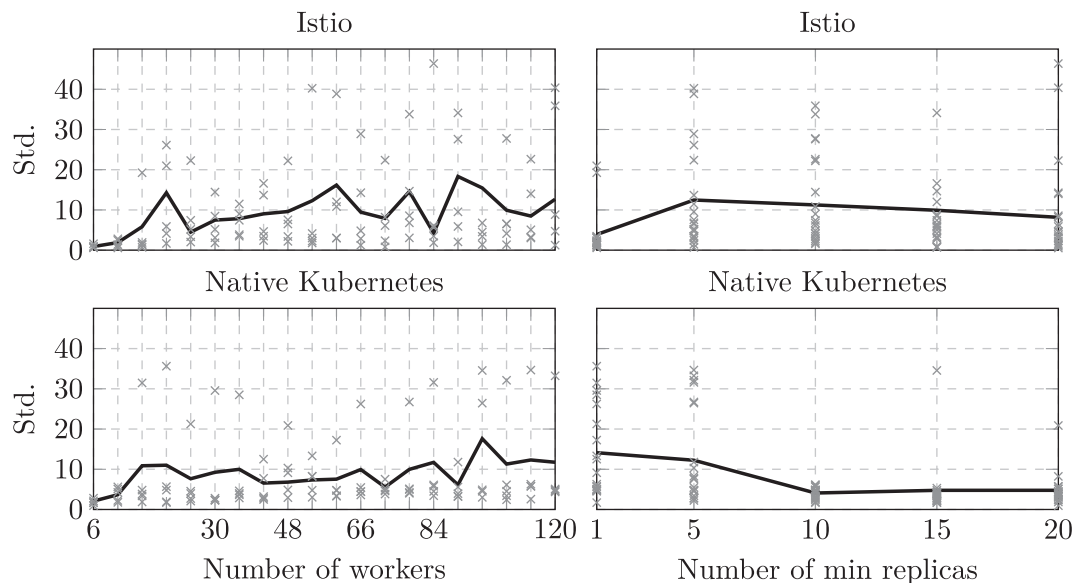
Figure 4 shows the average number of successfully served requests as a function of the number of initial Pods.

Note that autoscaling is used, and while each scenario stipulates a number of initial Pods as shown in Figure 4, all Deployments were allowed to scale to 20 Pods in total. Failure to do so under certain circumstances may partially explain why the average number of successfully handled requests differs between Istio and native Kubernetes networking.

As hypothesized, the performance of Istio is lower than that of native Kubernetes in terms of throughput because Istio performs more work per request (cf Section 2.1.2). A native Kubernetes deployment is strictly more performant than an Istio deployment. The discrepancy is at most more than 2-fold. To be more precise, a statistical analysis of the entire data set shows that there is a *significant difference* in performance between Istio and native Kubernetes networking ( $p = 8.8 \times 10^{-33}$  based on a Kruskal-Wallis H-test for independent samples).

However, the results of these initial experiments also revealed high variability within each scenario. Variability reflects the level of noise  $Z$  in the system. Figure 5 shows the standard deviation in successfully handled requests per second for each scenario. Small values indicate stability and a lack of nondeterministic behavior, corresponding to a situation in which reality behaves as theory predicts. Unfortunately, the data instead show that as the number of “moving parts” (ie,  $P$  and workers  $\mathcal{W}$ , as well as the use of Istio) in our system increases, so does the variability in our results. It is also evident that the standard deviation is significantly lower in underloaded systems ( $L < 16$ ).

<sup>4</sup>Scenarios prefixed *scaling* in the data set.



**FIGURE 5** Standard deviation in successful requests per second for each scenario. Xs represent data points for each scenario, while the solid line represents the mean of standard deviation over scenarios. Values further from zero indicate greater instability and thus stronger presence of nondeterministic behavior

#### 4.1.2 | Discussion

Readers should note that the data that let us produce Figure 4 seem to clearly support the intuition that Istio should perform worse than native Kubernetes networking. Rather than being satisfied with such results and trusting that our careful preparations were sufficient, we questioned the validity of our results and analyzed the data using statistical methods. The outcome (Figure 5) showed that we were risking publishing “wrong data without doing anything obviously wrong”<sup>4</sup> because of the system’s nondeterministic behavior. We therefore focused on finding the origins of this noise.

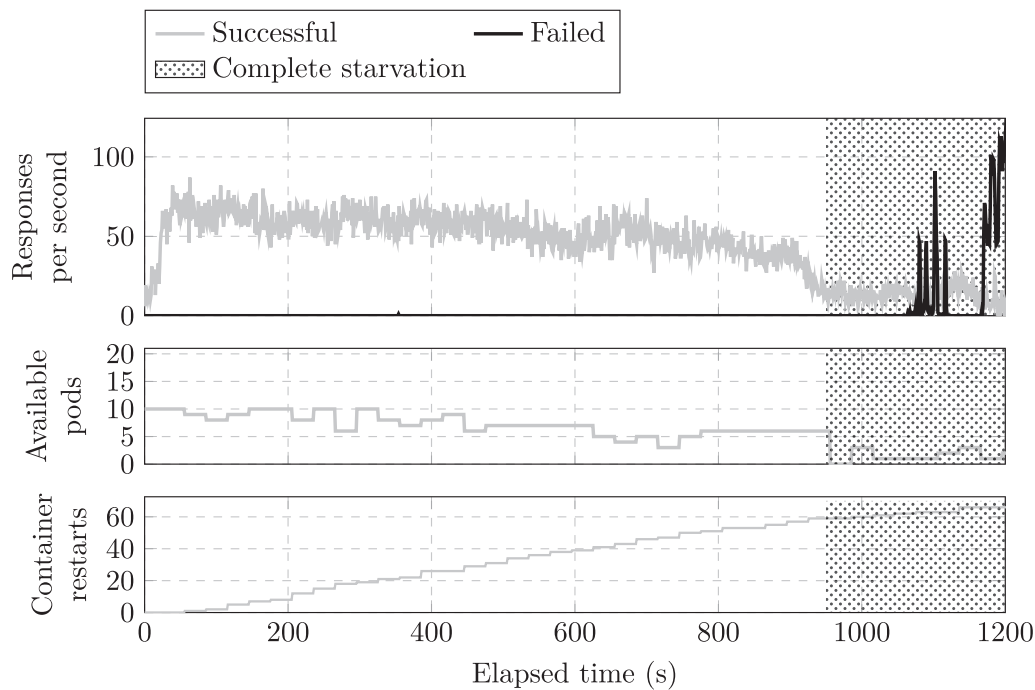
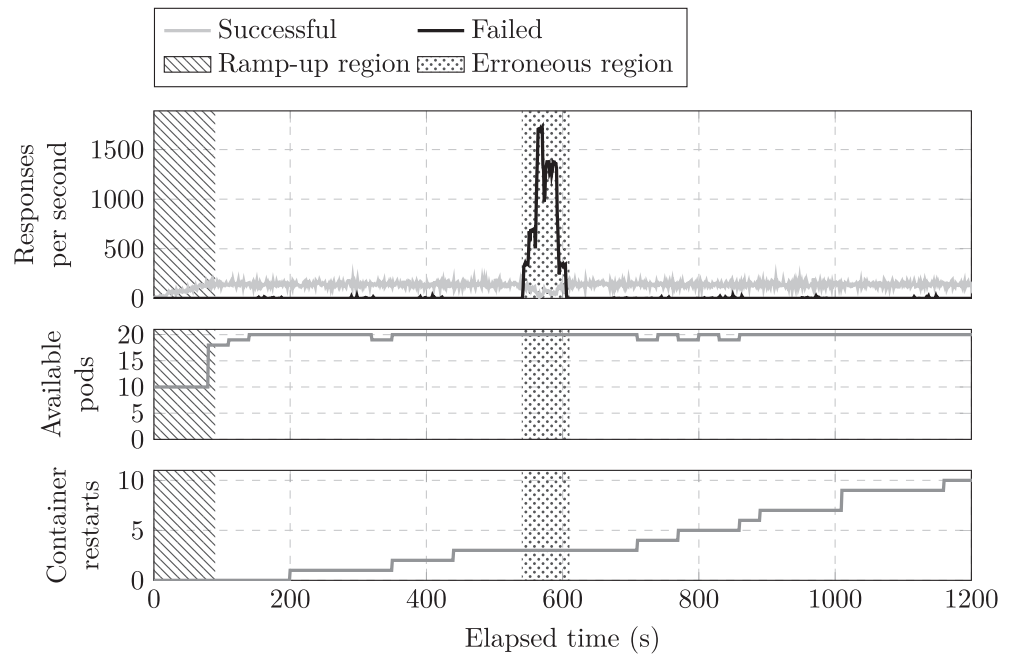
## 5 | NONDETERMINISTIC BEHAVIORS IN KUBERNETES CAUSED BY THE IMPACT OF ETCD PERFORMANCE

The large variability in results shown in Section 4 and Figure 5 prompted us to investigate the time series data more deeply. Experimental results illustrating two error archetypes of particular interest are presented in Figures 6 and 7. The corresponding experiments used the same basic parameters as those shown in Section 4. In the first case (Figure 6), it is apparent that *some* problem occurs and the system becomes unable to perform corrective action for a period of time. In the erroneous region highlighted in Figure 6, the successful response rate plummets and the failure rate increases rapidly, but is limited by  $\lambda^{\max} = 6000$  (the maximum of Equation (5) given  $t^{\text{delay}} = 20$  milliseconds). Consequently, application performance suffers and failures occur due to incorrect routing (cf Section 2.1.3). The second error archetype (Figure 7) involves frequent errors that render the HPA unable to request the correct number of desired Pods.

Realizing that these nondeterministic behaviors cast doubt on the results of our initial experiments, we sought to determine their root cause. Investigation along various paths ultimately led us to etcd Linux system journal entries, which showed a pervasive presence of warnings about operations taking longer than expected.

Recall that, at its core, Kubernetes consists of a database (etcd), an API server, and a set of components and controllers that perform orchestration tasks. These tasks and corrective actions are specified and triggered by data made accessible via the API and stored in the etcd database. The results presented here show that this dependence on etcd has a considerable performance impact. With all other parameters being as equal as possible in a cloud environment, etcd performance affects that of the application, particularly under high load. We are not aware of any other scientific or experiential work that studies and describes this relationship, and believe that performance tests that do not take this into account run a high risk of producing “wrong data without doing anything obviously wrong.”<sup>4</sup>

**FIGURE 6** An experiment where Kubernetes failed to take corrective action for an extended period of time (experiment ID 2c0609c0-9ef7-11e9-9783-4b7ce873c223)



**FIGURE 7** Compounded failures leading to starvation and the HPA being unable to alleviate the situation (experiment ID bbebcc1a-9ef6-11e9-82fc-870f50601c23)

## 5.1 | Factors impacting etcd performance

The etcd database is a distributed key-value store that uses the Raft algorithm for distributed consensus. To ensure that results are stable and persistent, it makes extensive use of disk I/O. It is known in the etcd community that poor I/O per second performance will negatively affect etcd<sup>5</sup>. However, to our knowledge there is no published information on

<sup>5</sup><https://github.com/etcd-io/etcd/blob/master/Documentation/platforms/aws.md>, accessed 22 October 2019

how such poor performance affects Kubernetes and applications deployed onto it. When deploying etcd, operators must choose one of two alternatives, neither of which is entirely satisfactory:

1. etcd can use only fast instance storage that is local to the VM and shares the VM's life cycle, making it susceptible to data loss or service disruption if the VM is terminated; or
2. etcd can use a network-attached block storage disk, which offers greater availability and independence from the life cycle of any VM it is attached to but has significantly worse performance.

In addition to the backing storage medium, CPU and network capacity of course also strongly affect etcd performance. However, unless the networks are truly saturated or greatly underprovisioned, the dominant factor is disk I/O performance.<sup>21,22</sup> We focus only on the dominant disk I/O factor, which proved to be trivial to modify and to have a significant impact.

## 5.2 | Experiments on the effect of etcd deployment performance on that of the application

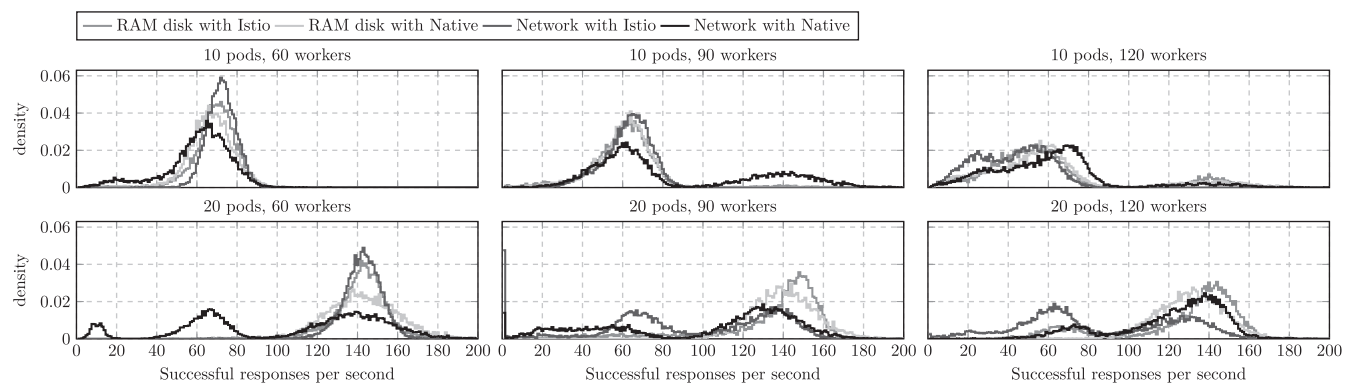
To determine the possible impact of etcd performance on that of our application, we devised a new suite of scenarios (cf Section 2.3.2). To quantify which the effects on both data plane (application) and control plane (Kubernetes platform) performance the backing data storage has on etcd, we decided to deploy it with two extremes in I/O performance. In one set of scenarios, we let etcd use a network-attached block storage disk (recalling Section 3.1, our private cloud provider did not provide local instance storage). In another set, we let etcd use a RAM disk as its data storage. This is *highly* inadvisable outside of research, as the loss of etcd data will render etcd (and thus Kubernetes) unable to recover. But its I/O performance is second to none!

The scenarios define experiments with fixed numbers of Pods ( $J \in [10, 20]$ ) and numbers of workers per JMeter worker node  $L \in [10, 15, 20]$ .

The results of these experiments in terms of the application's ability to successfully serve requests are shown in Figure 8. A statistical analysis of these results is presented in Table 1. Because the data are not normally distributed, the statistical analysis uses the Kruskal-Wallis H-test for independent samples instead of the T-test over all measurements of successful requests per second for all experiments belonging to the different scenarios.

### 5.2.1 | Interpretation of results

It is not, in itself, surprising that giving an I/O bound process faster backing data storage improves its performance. What was surprising to us, and what can be regarded as the main takeaway message of this experience article, is that poor etcd performance has a wide range of adverse effects on Kubernetes, both on the control plane and at the deployed application level.



**FIGURE 8** Impact on application performance due to different etcd deployment types, presented as normalized density plots of successfully served worker requests

**TABLE 1** Statistical analysis of application performance impact as a function of etcd deployment

Pods	Workers	Native Kubernetes RAM disk significant?	Istio RAM disk significant?
10	60	( $p = 4.07 \times 10^{-115}$ )	( $p = 7.36 \times 10^{-90}$ )
10	90	( $p = 3.48 \times 10^{-137}$ )	( $p = 5.37 \times 10^{-36}$ )
10	120	( $p = 1.68 \times 10^{-20}$ )	( $p = 2.02 \times 10^{-284}$ )
20	60	( $p = 0$ )	( $p = 0$ )
20	90	( $p = 0$ )	( $p = 0$ )
20	120	( $p = 2.21 \times 10^{-37}$ )	( $p = 0$ )

Note: All comparisons show that there is a statistically significant difference ( $p < .05$ ) between using a RAM disk or network-attached storage.  $p$ -values are calculated using the Kruskal-Wallis H-test for independent samples over all successful requests per second measurements for all experiments belonging to the different scenarios.

Figure 8 and Table 1 shows that when etcd performance suffers, so does that of Kubernetes and, in turn, that of any deployed application.

To our knowledge, these effects have not been reported in the research literature and are not discussed in any available documentation. As such, these findings should deepen the understanding of Kubernetes' internals within the community. Key Kubernetes functionality that depends on a rapidly responding and functional etcd demonstrably includes, but is not limited to, storage and retrieval of:

- Service Endpoint objects,
- scheduling decisions,
- measurements on which the HPA bases its actions, and
- leader election for components such as the Kubernetes Scheduler and Controller Manager.

By analyzing the results of our initial set of experiments and the associated etcd journal entries, we identified three serious problems directly related to poor etcd performance.

When the system is pushed to its limits, some or all of these problems can manifest, resulting in failures such as those shown in Figures 7 and 8. These problems are:

1. Slow etcd performance makes the set of Service Endpoints slow to update, causing kube-proxy to route traffic to Pods with containers that failed their probes. The response code generated by kube-proxy on behalf of the Service under such conditions is HTTP 503 Bad Gateway.
2. Control loops in Kubernetes (eg, in its Scheduler and Controller Manager) fail to acquire leadership leases, forcing them to exit and restart. They are then unable to resume operation until they have acquired a leadership lease, and therefore cannot make new decisions to help resolve ongoing problems.
3. A container that cannot report performance measurements for an interval of time will fail to cause the HPA to react, because the HPA refuses to scale if there are insufficient recent monitoring data upon which it can base such a decision.

The third problem state may manifest as a consequence of the first two, and the effect is that other Service Endpoints in a Deployment get higher loads than expected, and the HPA does nothing to resolve the situation. If more containers in the Service Endpoint Pods start to fail, the effect is a downward spiral of failures. Figure 7 shows what this looks like: the Deployment is unable to scale up from the 10 initial Pods and therefore the entire Deployment gets overwhelmed and gets no additional Pods from the HPA. Ultimately, the Deployment has only a fraction of the number of Pods it should have (other experiments in the same scenario scaled up to 20 Pods).

## 6 | RELATED WORK

Our experiments and results suggest that at least some of the observed nondeterministic behavior may have been caused by noisy neighbors in the cloud—performance interference due to poor inter-VM isolation. This problem is not new, and has been studied extensively by researchers and cloud practitioners.<sup>23-27</sup> It manifests itself across all virtualized



resource types: compute power,<sup>28,29</sup> network,<sup>29,30</sup> and storage.<sup>28,31</sup> The first step toward mitigation is to identify when there is a problem, and the second is to actually do something about it. Cloud infrastructure providers have monitoring tools that provide both the required insight into resource usage and the administrative powers required to perform high-level optimization of the infrastructure.<sup>32-35</sup> Although we used a private cloud in this work, we were neither the sole users (ie, we had no dedicated resources) nor did we have the administrative powers required for such mitigation techniques.

Cloud users have limited ability to detect and mitigate the effects of noisy neighbors because they lack access to the relevant data and have insufficient permissions to affect the deployment. However, some strategies exist, one of which is the “trial-and-better” strategy, wherein several VMs are provisioned, tested, and only those with good performance are kept.<sup>36</sup> We chose not to use this approach for two reasons. First, our experiments were long-running and while provisioning new VMs is fast, setting them up with kubespray as members of a Kubernetes cluster is sadly not. The process would have taken about 30 minutes in overall provisioning time alone, not to mention the performance testing required by the “trial-and-better” strategy. Adding an additional 40 minutes per experiment would have been too costly in terms of time. Second, the strategy only alleviates problems detectable during the trial period. Because our experiments lasted for many minutes, a sudden burst of network activity (eg, due to a neighboring VM pulling data for machine learning) could have a severe momentary adverse impact on performance at any point during an experiment.

Many studies have focused on benchmarking the control plane and the performance of the underlying cloud itself, and on using frameworks for conducting experiments. Systems developed for this purpose include C-Meter,<sup>37</sup> CloudCmp,<sup>38</sup> CloudGauge,<sup>39</sup> and CloudBench.<sup>40</sup> While there is conceptual overlap between these systems and our work in that we also measure performance in a cloud environment, our work focuses more on understanding the causes and effects of performance variations on the performance of the platform and deployed applications. In addition, our target domain is Kubernetes specifically. This particular domain appears to be less explored in the literature than that of general cloud environments, with simulation-based studies<sup>41-43</sup> being the most highly cited works in recent years.

That the Kubernetes control plane appears to function poorly under load, as our results also suggest, supports the work of Wei et al.<sup>44</sup> In that work, it was found that the scheduling component of Kubernetes struggled to keep up as 80% of node capacity was in use. However, their work uses a single-cluster minikube node, which is used both as a control plane and data plane, which more resembles a development cluster for a single developer (as indeed is the goal of the minikube project). By contrast, our work deploys a cluster closer to how is done in a production environment, with separate master and worker nodes. Furthermore, we also quantify the impact of the performance of the Kubernetes control plane on the data plane itself.

## 7 | CONCLUSIONS

Our work started with the best intentions and conditions: we sought to establish a solid in-depth understanding of networking in Kubernetes and Istio (Section 2.1), conduct theoretically sound performance tests (Section 2.2), and develop a framework to mitigate the adverse effects of shared cloud infrastructure (eg, noisy neighbors) when running experiments (Section 2.3). We performed experiments and got data that agreed perfectly with our expectations, but statistical analysis of our results showed that the data were fraught with noise and could therefore not be used in good faith to support our hypothesis. This prompted us to investigate the origin of the noise.

Our work shows that while Kubernetes ships with powerful abstractions that facilitate deployment across cloud providers, it adds a substantial number of “moving parts.” These moving parts cause nondeterministic behavior, particularly when under stress from high load. This impacts performance evaluations. Our data indicate that at the core of the problem is the etcd database and its performance. We have found no reports in the literature describing the relationship between poor etcd performance and poor performance of the application or the system as a whole. The implication of our findings is that performance tests on Kubernetes-based cloud environments must take into account the way etcd is deployed, and consider whether it has enough resources and sufficiently fast I/O to perform its work. Researchers conducting performance tests are therefore encouraged to consider their application and the Kubernetes platform itself together as their system under test.

Additionally, it behooves all users of Kubernetes to ensure that etcd has access to the fastest possible storage with reasonable availability guarantees. RAM disks are obviously unreasonable in practice, but local instance storage combined with a backup strategy to ensure that state changes have more persistence than the lifetime of the VM upon which it is deployed should constitute a reasonable trade-off that satisfies all needs.

## ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers who have helped improve the quality of this text substantially for journal publication. This work was supported financially in part by Vetenskapsrådet's eSSSENCE project, Knut och Alice Wallenbergs Stiftelse under the Wallenberg AI, Autonomous Systems and Software Program, Stiftelsen för Strategisk Forskning's SEC4FACTORY project, VINNOVA's 5G-PERFECTA Celtic Next project, and the Excellence Center Linköping-Lund in Information Technology (ELLIIT).


## DATA AVAILABILITY STATEMENT

The following assets are released into the open for researchers and industry practitioners to use and otherwise benefit from:

- <https://people.cs.umu.se/~larsson/etcd-performance-issues/> links to the entire data set of all experiments in PostgreSQL pgdump format, ready to be imported into a fresh PostgreSQL database installation for exploration.
- <https://github.com/llarsson/etcd-performance-issues-experiment-framework> contains the experiment framework we used to conduct experiments (Section 2.3). The code has been imported from its original repository to avoid inclusion of potentially sensitive information. Hence, git commit hash references in the data set are not going to match.
- <https://github.com/llarsson/etcd-performance-issues-application> contains the application used in the experiments. Like with the experiment framework, this repository has been recreated via importing data to avoid inclusion of potentially sensitive information. Similarly, git commit hashes referred to in the data set are not going to match.
- <https://github.com/llarsson/etcd-journal-entries> contains the journal entries from etcd, collected for some 1700 experiments (see data set for experiment IDs).

## ORCID

Lars Larsson  <https://orcid.org/0000-0001-5860-6695>

William Tärneberg  <https://orcid.org/0000-0003-1316-8059>

Cristian Klein  <https://orcid.org/0000-0003-0106-3049>

Erik Elmroth  <https://orcid.org/0000-0002-2633-6798>

Maria Kihl  <https://orcid.org/0000-0003-3396-1652>

## REFERENCES

1. Gartner Inc *Market Share Analysis: IAAS and IUS, Worldwide, 2018 Market Report*. Stamford, Connecticut, USA; 2018.
2. Gartner Inc *Best Practices for Running Containers in Production Market Report*. Stamford, Connecticut, USA; 2019.
3. Gartner Inc *Innovation Insight for Service Mesh Market Report*. Stamford, Connecticut, USA; 2018.
4. Mytkowicz T, Diwan A, Hauswirth M, Sweeney PF. Producing wrong data without doing anything obviously wrong! *ASPLOS*. Vol XIV. New York, NY: ACM; 2009:265-276.
5. Maenhaut PJ, Volckaert B, Ongenaes V, De Turck F. Efficient resource management in the cloud: from simulation to experimental validation using a low-cost raspberry Pi testbed. *Softw Pract Exp*. 2019;49(3):449-477. <https://doi.org/10.1002/spe.2669>.
6. Rodrigues LR, Cardoso E Jr, Alves OC Jr, et al. Cloud broker proposal based on multicriteria decision-making and virtual infrastructure migration. *Softw Pract Exp*. 2019;49(9):1331-1351. <https://doi.org/10.1002/spe.2723>.
7. Varshney P, Simmhan Y. Characterizing application scheduling on edge, fog, and cloud computing resources. *Softw Pract Exp*. 2020;50(5):558-595. <https://doi.org/10.1002/spe.2699>.
8. Zhou H, Hu Y, Ouyang X, et al. CloudsStorm: a framework for seamlessly programming and controlling virtual infrastructure functions during the DevOps lifecycle of cloud applications. *Softw Pract Exp*. 2019;49(10):1421-1447. <https://doi.org/10.1002/spe.2741>.
9. Natella R, Cotroneo D, Madeira HS. Assessing dependability with software fault injection: a survey. *ACM Comput Surv (CSUR)*. 2016;48(3). <https://doi.org/10.1145/2841425>.
10. Schroeder B, Gibson GA. A large-scale study of failures in high-performance computing systems. *IEEE Trans Depend Sec Comput*. 2010;7(4):337-350. <https://doi.org/10.1109/TDSC.2009.4>.
11. Braband J. Waiting time distributions for closed M/M/N processor sharing queues. *Queue Syst*. 1995;19(3):331-344. <https://doi.org/10.1007/BF01150417>.
12. Schroeder B, Wierman A, Harchol-Balter M. Open versus closed: a cautionary tale. Paper presented at: Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3 (NSDI'06); 2006:239-252; USENIX Association, San Jose, CA.
13. Kleinrock L. *Queueing Systems: Volume 1: Theory*. Hoboken, NJ: Wiley-Interscience; 1975.
14. Kalmanek CR, Misra S, Yang Y. *Guide to Reliable Internet Services and Applications*. Springer: London, UK; 2010.
15. Feitelson DG. From repeatability to reproducibility and corroboration. *SIGOPS Operat Syst Rev*. 2015;49(1):3-11. <https://doi.org/10.1145/2723872.2723875>.

16. Papadopoulos AV, Versluis L, Bauer A, et al. Methodological principles for reproducible performance evaluation in cloud computing. *IEEE Trans Softw Eng*. 2019;1–1. <https://doi.org/10.1109/TSE.2019.2927908>.
17. Edwards S, Liu X, Riga N. Creating repeatable computer science and networking experiments on shared, public testbeds. *ACM SIGOPS Operat Syst Rev*. 2015;49(1):90–99. <https://doi.org/10.1145/2723872.2723884>.
18. Nathuji R, Kansal A, Ghaffarkhah A. Q-clouds: managing performance interference effects for QoS-aware clouds. Paper presented at: Proceedings of the 5th European Conference on Computer Systems; 2010:237–250; ACM, New York, NY.
19. Großmann M, Schenk C. A comparison of monitoring approaches for virtualized services at the network edge. Paper presented at: Proceedings of the 2018 International Conference on Internet of Things, Embedded Systems and Communications (IINTEC); 2018:85–90; IEEE, Hammamet, Tunisia.
20. Casalicchio E, Perciballi V. Measuring docker performance: what a mess!!! Paper presented at: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion; 2017:11–16, ACM, L'Aquila, Italy.
21. Olivi M, Spreitzer M. Using fio to tell whether your storage is fast enough for etcd web; 2019. <https://www.ibm.com/cloud/blog/using-fio-to-tell-whether-your-storage-is-fast-enough-for-etcd> IBM. Accessed May 28, 2020.
22. Healy M, Seelam S. Does Intel Optane™ persistent memory improve etcd performance or scalability? web; 2020. <https://www.ibm.com/cloud/blog/does-intel-optane-persistent-memory-improve-etcd-performance-or-scalability> IBM. Accessed May 28, 2020.
23. Yabandeh M, Knezevic N, Kostic D, Kuncak V. CrystalBall: predicting and preventing inconsistencies in deployed distributed systems. Paper presented at: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09); 2009:229–244; USENIX Association, Boston, MA.
24. Tan Y, Nguyen H, Shen Z, Gu X, Venkatramani C, Rajan D. PREPARE: predictive performance anomaly prevention for virtualized cloud systems. Paper presented at: Proceedings of the 2012 IEEE 32nd International Conference on Distributed Computing Systems (ICDCS '12); 2012:285–294; IEEE Computer Society, Macau, China.
25. Iosup A, Ostermann S, Yigitbasi MN, Prodan R, Fahringer T, Epema D. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Trans Parall Distrib Syst*. 2011;22(6):931–945. <https://doi.org/10.1109/TPDS.2011.66>.
26. Iosup A, Yigitbasi N, Epema D. On the performance variability of production cloud services. Paper presented at: Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2011); 2011:104–113; IEEE/ACM.
27. Leitner P, Cito J. Patterns in the Chaos—a study of performance variation and predictability in public iaas clouds. *ACM Trans Internet Technol*. 2016;16(3):15:1–15:23. <https://doi.org/10.1145/2885497>.
28. Silva M, Ryu KD, Da Silva D. VM performance isolation to support QoS in cloud. Paper presented at: Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS 2012) Workshops PhD Forum; 2012:1144–1151; IEEE, Shanghai, China.
29. Ericson J, Mohammadian M, Santana F. Analysis of performance variability in public cloud computing. Paper presented at: Proceedings of the IEEE International Conference on Information Reuse and Integration (IRI); 2017:308–314; IEEE, San Diego, CA.
30. Kumar P, Choudhary G, Sharma D, Mann V. Equinox: adaptive network reservation in the cloud. Paper presented at: Proceedings of the Sixth International Conference on Communication Systems and Networks (COMSNETS); 2014:1–8; IEEE, Bangalore, India.
31. Ghosh R, Quirk AJ. *Reducing the Impact of Noisy Neighbors via Pro-Active Log Offloading in Shared Storage Environment PatentUS20160164962A1*. United States Patent and Trademark Office; 2016.
32. Chen L, Viswanathan K, Ban K. *Identification of Incompatible Co-tenant Pairs in Cloud Computing PatentUS20180241811A1*. United States Patent and Trademark Office; 2018.
33. Tomás L, Vázquez C, Tordsson J, Moreno G. Reducing noisy-neighbor impact with a fuzzy affinity-aware scheduler. Paper presented at: Proceedings of the International Conference on Cloud and Autonomic Computing (ICCAC); 2015:33–44; IEEE, Boston, MA.
34. Lorigo-Botran T, Huerta S, Tomás L, Tordsson J, Sanz B. An unsupervised approach to online noisy-neighbor detection in cloud data centers. *Exp Syst Appl*. 2017;89:188–204. <https://doi.org/10.1016/j.eswa.2017.07.038>.
35. Kandalintsev A, Kliazovich D, Lo CR. Freeze'nSense: estimation of performance isolation in cloud environments. *Softw Pract Exper*. 2017;47(6):831–847.
36. Lloyd W, Pallickara S, David O, Arabi M, Rojas K. Mitigating resource contention and heterogeneity in public clouds for scientific modeling services. Paper presented at: Proceedings of the IEEE International Conference on Cloud Engineering (IC2E); 2017:159–166; IEEE, Vancouver, Canada.
37. Yigitbasi N, Iosup A, Epema D, Ostermann S. C-Meter: a framework for performance analysis of computing clouds. Paper presented at: Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID 2009); 2009:472–477; IEEE/ACM, Melbourne, Australia.
38. Li A, Yang Xi, Kandula S, Zhang M. CloudCmp: comparing public cloud providers. Paper presented at: Proceedings of the 10th ACM SIGCOMM conference on Internet measurement (IMC '10); 2010:1–14; ACM, Melbourne, Australia.
39. El-Refaey MA, Rizkaa MA. CloudGauge: a dynamic cloud and virtualization benchmarking suite. Paper presented at: Proceedings of the 19th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE '10); 2010:66–75; IEEE, Larissa, Greece.
40. Silva M, Hines MR, Gallo D, Liu Q, Ryu KD, Da Silva D. CloudBench: experiment automation for cloud environments. Paper presented at: Proceedings of the IEEE International Conference on Cloud Engineering (IC2E); 2013:302–311; IEEE, San Francisco, CA.
41. Medel V, Rana O, Bañares JÁ, Arronategui U. Adaptive application scheduling under interference in kubernetes. Paper presented at: Proceedings of the 2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC); 2016:426–427; IEEE/ACM, Shanghai, China.

42. Medel V, Rana O, Bañares JÁ, Arronategui U. Modelling performance & resource management in Kubernetes. Paper presented at: Proceedings of the 9th International Conference on Utility and Cloud Computing (UCC '16); 2016:257-262; ACM, Shanghai, China.
43. Piraghaj SF, Dastjerdi AV, Calheiros RN, Buyya R. ContainerCloudSim: an environment for modeling and simulation of containers in cloud data centers. *Softw Pract Exper*. 2017;47(4):505-521. <https://doi.org/10.1002/spe.2422>.
44. Wei T, Malhotra M, Gao B, Bednar T, Jacoby D, Coady Y. No such thing as a “free launch”? Systematic benchmarking of containers. Paper presented at: Proceedings of the 2017 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM); 2017:1-6; IEEE, Victoria, Canada.

**How to cite this article:** Larsson L, Tärneberg W, Klein C, Elmroth E, Kihl M. Impact of etcd deployment on Kubernetes, Istio, and application performance. *Softw Pract Exper*. 2020;50:1986–2007. <https://doi.org/10.1002/spe.2885>