# A Two-Tier Storage Interface for Low-Latency Kubernetes Deployments

by

Teodor Alexandru Ionita

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2022

## Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

**Abstract**

Orchestration systems are responsible for automating the management of software deployments on computer systems, managing large numbers of machines. Edge computing is a decentralized compute model describing how compute and storage resources can be placed closer to the end-user to reduce latency. Despite its geographically disaggregated nature, edge computing still retains a need to be managed. It takes one of its shapes as Multi-access Edge Computing (MEC), a telecommunications technology that integrates edge computing with the radio base sites scattered throughout metropolitan areas. The MEC installations, in order to offer low latency, face the user handoff challenge, which involves the end user being serviced by different MEC servers as they transit, hopping from one MEC to another. This handoff requires that the orchestration system handling the network of MEC servers be able to handle, in its control plane, this churn of applications. In this work, the popular orchestration system Kubernetes sees its application deployment throughput improved by up to 1.87X, as well as an improvement in the latencies for deployment requests, through a storage layer rearchitecture that involves categorizing internal message types and deploying a secondary store.

**Acknowledgments**

I would like to thank all the people who made this thesis possible. I would like to acknowledge Rogers Communications for their support.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Edge computing is a decentralized computing model that places compute and storage resource closer to end users and is contrasted with traditional centralized datacenters [12]. Edge computing addresses the issue that network latency becomes for latency-sensitive applications when they must communicate with datacenter-hosted services as cloud datacenters remain geographically sparse [4] and latency suffers both from geographical distances and packet switching delays along many network segments. For example, the nearest Canadian datacenter to Vancouver for users of Google Cloud Platform is in Toronto [18], over 3000 kilometers away – a minimum roundtrip time of 20 milliseconds at the speed of light without considering packet switching delays or network congestion.

Multi-access Edge Computing (MEC) [16] describes how compute and storage resources can be placed within the cellular network infrastructure, adding edge computing capabilities to telecommunication networks, which have historically only served as a data transmission mechanism that connects end-user devices to other end-user devices or to the Internet. The exact form of MEC installations, i.e., which compute and storage resources and where they are placed in the topology, is flexible and depends on the application, end-users and cellular network topology. Existing MEC installations include Amazon's Wavelength service [42], offered in partnership with cellular telecommunication companies, where the edge servers are deployed at city-level granularity [44]. Amazon customers can deploy their applications to a particular city – the edge servers are running inside the carrier's metropolitan datacenter [43] – but do not have access to a finer geographical selection, e.g., they cannot specify edge servers running within a district of the city. A MEC installation even closer to the edge – and to the end-user – would be running alongside the radio towers spread throughout the city and would provide lower latency than metropolitan ones.

MEC installations co-located with radio towers experience handoff challenges, the same type experienced by cell towers with cellular connections. When traveling, an end-device's cellular connection is handed off from one cell tower to another, according to signal quality considerations, e.g., as distances to neighbouring cell towers change. Similarly, according to application latency considerations, MEC connections are also handed off [33], as to avoid excessive geographical distance from their associated MEC and to avoid network congestion stemming from traversing network segments. MEC handoff can also occur in non-mobility scenarios, such as MEC failure, QoS prioritization and load balancing.

There are several frameworks that tackle managing the deployment and lifecycle of services on MEC installations [7, 8] or more generally Edge systems [6, 9, 10]. The frameworks work in a controller-agent pattern, where the controller instructs the agents – the MEC installations – to deploy the applications relevant to the end-device associated with the MEC. As MEC handoffs take place, it is the controller's responsibility to request the application be stopped on one MEC and started on another. The surveyed frameworks re-use software components, mainly Kubernetes – an open-source container orchestration system [28] – as the core controller logic that is subjected to the MEC handoff scenario.

An important performance dimension of the Kubernetes control plane is its ability to perform writes to its persistent storage, an operation that is core to several functions, including application deployments in the cluster. In the context of MEC handoffs, application deployment throughput and latency is an essential metric to the control plane's role.

## 1.1 Contributions

This thesis investigates the ability of MEC frameworks that use Kubernetes as their control logic to handle the frequent application migration within the cluster that occurs during MEC handoff and present an architectural change that touches the storage layer and that improves MEC handoff performance. First, I present the subset of Kubernetes operational concepts necessary to describe the components relevant to the discussion. I then describe the Kubernetes architecture, the interaction between the Kubernetes parts at play during the start and stop of an application and general communication patterns within Kubernetes. Next, I explain the changes to the architecture that improve the performance of Kubernetes' controller component. Finally, I present a comparison between the initial Kubernetes performance and the performance stemming from the architectural changes.

# Chapter 2

# Background

## 2.1 Kubernetes

Kubernetes is an open-source platform, written by Google in Go, that orchestrates the deployment and management of containers. Kubernetes documentation indicates that it can be responsible for up to 5000 servers or virtual machines (VMs) and up to 300,000 containers [23]. Out of all the machines under its control, collectively called a Kubernetes cluster, some are dedicated to the control plane while the rest of the machines, called nodes (and not part of the control plane) are dedicated to running containerized applications. A cluster must have an agent installed on all cluster nodes to control the creation and deletion of containers, and generally monitor the health of the machine and its processes. The per-node agent is also responsible for configuring certain OS-level networking rules. Kubernetes handles the deployment of containers on its nodes but also handles some of the networking aspects (e.g. port and address translation) to enable external requests to reach the correct container on the correct node inside the cluster, to allow the reply to reach the external client and to facilitate inter-container communication both inside a same node and between nodes.

### 2.1.1 Control Loops For Declarative Configurations

The Kubernetes platform removes the explicit association between a container and where precisely it is being deployed. Deployment of containers does not necessitate that the operator, when requesting a deployment, also specify on which nodes the containers must

run. The platform internally keeps track of the different containers and other configurations that are active, and is able to report state and react to events. An unexpectedly dead container leads to a node starting a new instance of the container image, while a dead node leads to its containers being restarted on a healthy node.

Kubernetes adopts a declarative approach to configurations: the configuration describes a desired final state and Kubernetes utilises control loops in its control plane and nodes to reach it. Once the state is reached, Kubernetes monitors it and reacts if changes in the cluster lead to the state changing. For example, if a configuration specifies two instances of a container image must exist in the cluster, and supposing there are zero instances currently running, the control loop attempts to bring the current state to match the desired state by starting two containers. In effect, this means that deploying an application entails indicating the application container image should be running, thus changing the desired cluster state. Kubernetes internally executes the intermediary steps necessary to start the application and make the current state reflect the desired state.

### 2.1.2 Kubernetes Objects

Kubernetes uses objects that describe entities within the cluster. An entity could be a group of containers, or a particular network configuration that allows two containers to communicate with each other across the cluster. The collection of all Kubernetes objects, persisted in the control plane's storage, represents the state of the cluster. Nearly all Kubernetes objects contain a *spec* field and a *status* field. The *spec* field describes, using several subfields, a desired state related to that entity. When a cluster operator creates or edits an object, this field must be set. A *status* field describes the current state of the entity. This field is updated by components within Kubernetes as they work to match the current state to the desired state.

Commonly, there are one or more control loops that react to an object being created or modified. Some object types act as starting objects and are normally exclusively created by an operator, not the Kubernetes control plane. The control loops associated with these starting object types react by creating/modifying their own objects, an action that may in turn trigger further control loops to react.

#### 2.1.2.1 Deployment

The deployment object generally describes all the facets related to deploying one or several containers (a group of containers is called a pod, see 2.1.2.3) on nodes. It identifies, among

other things, what container image to run, which version, how many instances, whether there is a rollout strategy (e.g. in case of a service version upgrade), whether there are any additional constraints on the nodes which might run the containers (e.g. the node might need a minimum amount of memory) and any additional networking configurations (e.g. the container needs to be reachable by clients external to the cluster).

The deployment object is generally a starting object (though not the only one) created by a cluster operator that interfaces with the Kubernetes control plane. This object's creation or modification brings different control loops into action that eventually lead to particular nodes running containers and changing their networking configurations. For example, an operator could decrease the number of instances of a particular container in the cluster by editing the persisted deployment object which was used to start them; this editing action results in control loops being triggered and in the modification of other objects until ultimately containers are stopped.

#### 2.1.2.2 Replicaset

The replicaset object is typically created by one of the control loops monitoring deployment objects. It is mainly used to implement deployment strategies involving different versions of the same container, such as rollbacks or updates; when using deployment strategies, a deployment object can lead to the creation of more than one replicaset objects. The simplest deployment strategy creates one replicaset object for every deployment object.

#### 2.1.2.3 Pod

The pod object is ordinarily created by a control loop monitoring replicaset objects. A pod represents a group of one or more containers that run on the same node and is the smallest unit in the deployment process. Once the pod object is created, the control plane scheduler is responsible for associating the pod with a node in the cluster; this association is added to the pod object itself. The chosen node is notified of this new pod that falls under its responsibility and starts the appropriate containers per the pod's description.

### 2.1.3 Label

Labels are key-value pairs that can be included in Kubernetes objects to identify salient properties. Labels can be operator-created to organize and create configurations that target specific entities. For example a group of nodes in the cluster could be carrying a label

'environment=production' while another group carries a label 'environment=QA'. When pods are deployed to the cluster, the operator can constrain the scheduling of the pods to a node carrying the appropriate 'production' or 'QA' label value for the 'environment' label key. Some labels are built-in, called well-known labels: for instance, nodes have automatically created labels describing the running OS e.g. 'Kubernetes.io/os=linux'. Certain objects, such as the replicaset object, inherit the labels of its parent deployment object; pod objects also inherit its replicaset parent labels.

### 2.1.4 Ownership and Garbage Collection

Kubernetes supports the concept of ownership, where one object owns another, which is used during object garbage collection. When deleting an object, such as a deployment object, the replicaset object it had engendered is also deleted. In turn, the pod object is likewise deleted once its parent replicaset object is removed from the cluster state. The control loops are responsible for marking which object owns another; this information is embedded as a field within the owned object. When an owner object is deleted, control loops are notified of this object change and proceed to delete owned objects; this deletion cascade continues until the final owned object is reached. There is no explicit separate schema describing the owner-owned relationship parameters or the actions to be taken upon object deletion; this information is embedded within the different control loops' logic.

### 2.1.5 Manifest Files

Manifest files, generally *YAML Ain't Markup Language* (YAML) files, are created by the operator and represent a Kubernetes object. Figure 2.1 illustrates an example of a deployment object manifest YAML file, as seen on Kubernetes.io [32]. The manifest spec field uses the replicas subfield to indicate 3 pods will be started. The spec.template.spec field describes the container image to be used.

### 2.1.6 kubectl

The kubectl utility is a command-line interface (CLI) application that interfaces with the Kubernetes control plane. It allows a user to modify the cluster state by creating or modifying Kubernetes objects e.g. deployment objects. Manifest files, along with other CLI arguments, are provided to kubectl which then creates appropriate *representational*

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Figure 2.1: Deployment manifest example

*state transfer* (REST) requests directed at the control plane. The tool also allows users to query the state of the cluster, either synchronously with a *get* command or asynchronously through notifications of changes.

### 2.1.7 client-go

Client-go is a Go client library for applications wishing to communicate with the Kubernetes control plane. Internally, all Kubernetes components use the client-go library to issue internal requests.

### 2.1.8 Official Kubernetes Performance Objectives

The Kubernetes performance documentation [29] specifies *service level objectives* (SLOs) that the Kubernetes development team aims to meet. There are currently none that relate to application deployment or deletion throughput in the control plane or cluster as a whole. In terms of end-to-end application deployment/deletion latency, the closest SLO relates to mutating a single object in the cluster (i.e., writing to etcd) but does not specifically involve deployment-related objects or any end-to-end process, be it in the control plane or the cluster. As such, no existing Kubernetes SLOs attempt to define throughput or end-to-end performance characteristics.

## 2.2 Kubernetes Architecture

The Kubernetes platform is a collection of stateless processes and etcd, which is used for persistent storage. The platform is composed of processes running as control plane components while other processes run as node components. Every component described here can be mapped to a separate Linux process. While Kubernetes can be extended to include additional components that integrate with cloud providers, the core components and architecture are described.

The node components run as agents on baremetal servers or inside virtual machines. On a given machine, there is a *kubelet* process and a *kube-proxy* process. The *kubelet* is responsible for handling container resources: starting, stopping and monitoring containers running locally on the machine. The *kube-proxy* is responsible for setting up OS-level networking rules (e.g. using iptables [24]) that ensure containers are reachable throughout the cluster, e.g., Machine A could receive a request meant for Machine B and this request must be forwarded by Machine A to Machine B. Both the kubelet and the kube-proxy attempt to locally maintain a desired state that is the sum of Kubernetes objects received from the control plane for their node. For example, when the control plane schedules a pod object on a node, effectively dictating new containers need to be started on that machine, the kubelet starts the number of containers needed to satisfy the desired state and reacts to changes to that number, starting more containers if any fail. Finally, the kubelet is responsible for reporting the health of the machine on which it is running as well as the state of all the pods under its responsibility.

The control plane components (summarized in Table 2.1) are responsible for receiving declarative objects from operators of the Kubernetes cluster and ensuring that the correct node components receive pod and network objects whose desired state they maintain. The

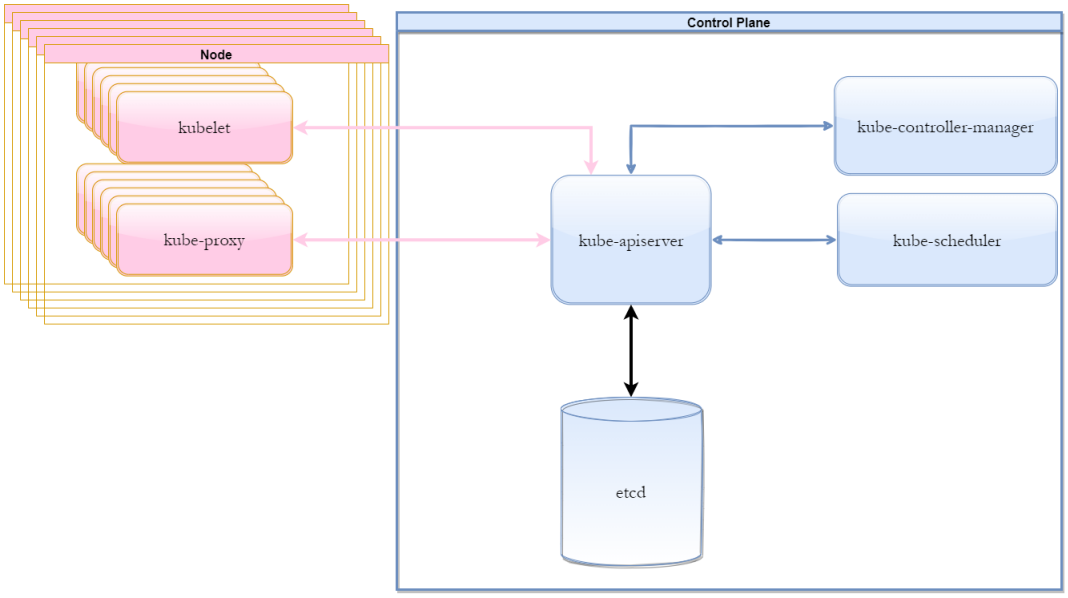


Figure 2.2: Kubernetes Spoke Hub Architecture.

platform follows a hub and spokes architecture [27]; all control plane and node components exclusively communicate with the kube-apiserver which resides in the control plane. The kube-apiserver has exclusive access to the etcd-backed storage where all Kubernetes objects are stored. While all components communicate with the kube-apiserver, they do so to indirectly write to or read from etcd, for which the kube-apiserver plays the role of a REST front-end. Being the focal point of communication, the kube-apiserver is also responsible for authentication, authorization, auditing, object validation, object transformation and delivering etcd notifications. The kube-controller-manager component is composed of subcomponents call controllers, all running within the same Linux process. Generally, every controller subcomponent is responsible for maintaining the state of a specific type of Kubernetes object but can also create or modify other types, e.g., the deployment controller is responsible for deployment objects but can create replicaset objects. Controllers rely on etcd notifications to trigger their control loop (more in 2.3). Finally, the kube-scheduler is responsible for selecting which pod gets assigned to which node in the cluster, a process called binding. The kube-scheduler binds pods to nodes based on constraints which are in part expressed as labels on the objects. Some constraints are related to node health and node resources, e.g., available node memory, while others are operator-defined, e.g., a QA-labeled pod must not run on a production-labeled node. Figure 2.2 illustrates the inter-component communication in the Kubernetes cluster.

| | Component | Functions |
|---|---|---|
| Control Plane | etcd | Store the cluster state, notify of object changes |
| | kube-apiserver | Front-End API to etcd; validate requests |
| | kube-controller-manager | Run controllers that each create, delete, and modify their assigned object types |
| | kube-scheduler | Select node for pod while respecting constraints |
| Node | kubelet | Create or delete pods machine, report pod state and node health |
| | kube-proxy | Modify machine OS-level networking rules |

Table 2.1: Kubernetes control plane and node components.

### 2.2.1 etcd

Etcd is the key-value store used to persist Kubernetes cluster objects. Etcd can run as a distributed storage system where a write operation is considered complete once committed through consensus and written to disk [14]. Etcd also offers atomicity, ensuring no partial results are observed, as well as a simple transaction system based on if/then/else constructs that can group several modifications and atomically condition a write based on values inside the store. Etcd also supports a notification system: etcd clients can ask to be notified when a key within the store is modified.

Kubernetes generally selects the key for its objects following the schema `/registry/{object type}/{object namespace}/{object name}`. For example, a deployment object named 'nginx-1' is stored at key '/registry/deployments/default/nginx-1' if no specific namespace is provided (a default namespace exists) while a replicaset object named 'nginx-1-1234ab567c' is be stored at key '/registry/replicasets/default/nginx-1-1234ab567c'. As such, when Kubernetes components wish to receive notifications of changes to any deployment object in the cluster, they register for notifications of changes to the keyspace prefix '/registry/deployments' which generates a notification for a change performed to the object at key '/registry/deployments/default/nginx-1'.

## 2.3 The Application Deployment Process

The kube-apiserver serves as an Hypertext Transfer Protocol (HTTP) REST front-end to etcd and therefore handles read/write requests to etcd from controllers or cluster operators, returning success or failure for the requests and data from the etcd store. The kube-apiserver is also responsible for being a notification middleman: it requests that etcd send notifications when modifications occur on certain keyspace prefixes within the etcd store. These notifications contain the exact modified key, the modification type (added, modified, or deleted) and the latest value of that object if applicable. The notifications are then in turn pushed to controllers who have registered – with the kube-apiserver – interest in particular keyspace prefixes. Depending on its responsibilities, a controller can register for notifications for several keyspace prefixes; for example, the replicaset controller expresses interest in changes to both replicaset objects and pod objects which means notifications for the keyspace prefixes '/registry/replicasets' and '/registry/pods'.

Deploying an application in the Kubernetes cluster control plane has four phases, illustrated in Figure 2.3: the creation of a deployment object (1), then a replicaset object (2), followed by a pod object (3) and finally an update to the pod object to indicate to which node it is bound (4). Each of these phases involves a roundtrip between etcd and a controller, with the kube-apiserver interposed between them. To simplify the discussion, the term controller includes the kube-scheduler component alongside the deployment controller and replicaset controller, both subcomponents of the kube-controller-manager. The application deployment process depends on the initial deployment object. The one in this section (Figure 5.4) describes a straightforward deployment scenario that ultimately involves the four phases mentioned and recruits the participation of the deployment controller, replicaset controller and kube-scheduler; however, more complex initial deployment objects can result in numerous other phases, more Kubernetes objects being created and updated, and the activation of supplemental controllers.

For all of the four phases, the component interactions are similar and can be summarized in four code paths and inter-component communication. In Figure 2.3, [A] represents the kube-apiserver receiving and processing a REST request for the creation/update of a Kubernetes object, be it deployment, replicaset or pod. After internal processing that validates and transforms the object for etcd storage, the code path [A] ends with a call to write the transformed object to the etcd store. [B] represents two consecutive but separate etcd actions: first, etcd stores the data as requested by the kube-apiserver. Second, etcd emits a notification because the write has modified a keyspace prefix for which the kube-apiserver has requested notifications. The notification reaches the kube-apiserver and travels through code path designated by [C]. Code path [C] performs object transforma-

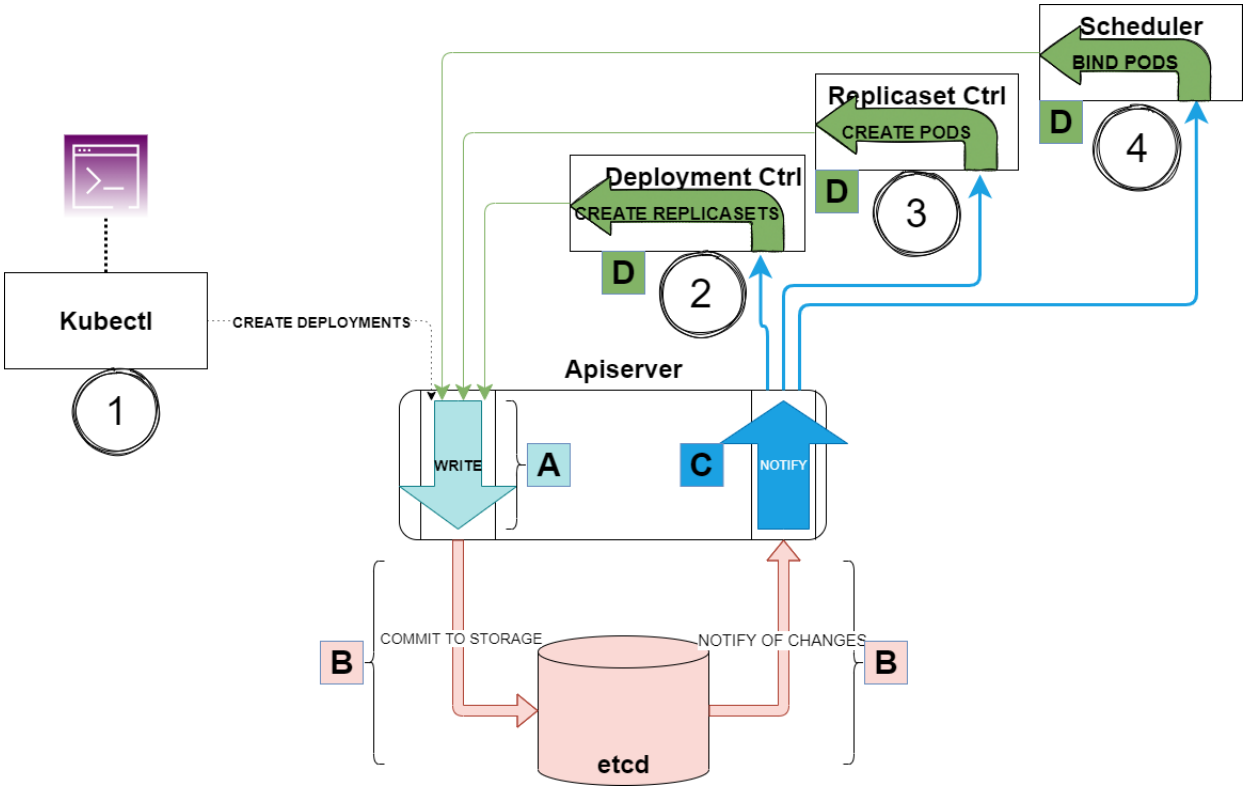


Figure 2.3: The Kubernetes application deployment process. The four phases are labeled 1 to 4, while the different parts of the roundtrip path are labeled A-D.

tions and delivers notification copies to the different components who have registered their interest. Code path [D] represents a controller receiving a notification about a change to an object, its internal processing and the result – which is the object for the next phase – followed by its request to the kube-apiserver to write this newly created object.

The full process for an application deployment is described below with a summary in Table 2.2.

(1) The operator makes a request for the creation of a deployment object to the kube-apiserver. In [A], the kube-apiserver validates, transforms the deployment object and calls etcd to perform a write to the store. In [B], etcd stores the object and then notifies the kube-apiserver of this new write that occurred to the '/registry/deployments' keyspace prefix. In [C], the kube-apiserver receives the notification, transforms the object, finds that the kube-controller-manager has registered for notifications to

| Phase | Summary |
|---|---|
| ① | Deployment object created in etcd by the operator, deployment controller is notified and generates a replicaset object |
| ② | Replicaset object created in etcd by the deployment controller, replicaset controller is notified and generates a pod object |
| ③ | Pod object created in etcd by the replicaset controller, kube-scheduler is notified and updates pod object to bind it to a node |
| ④ | Pod object is updated by the kube-scheduler in etcd and appropriate node is notified of the new pod |

Table 2.2: Kubernetes application deployment process, phases summary.

changes to the '/registry/deployments' keyspace prefix and delivers the notification. In [D], the kube-controller-manager internally forwards the notification specifically to the deployment controller subcomponent. The deployment controller generates a replicaset object and [D] ends with a request to the kube-apiserver for the creation of the replicaset object.

② The replicaset creation request is received in [A], the object is written by etcd and the notification of its creation emitted to the kube-apiserver in [B]. In [C], the kube-apiserver transforms the object and delivers the notification to the kube-controller-manager. In [D], the kube-controller-manager internally delivers the replicaset creation notification to the replicaset controller which generates a pod object and requests the pod object be written to the store through a REST request to the kube-apiserver.

③ Path [A] and [B] result in the pod object being written and a notification of its creation delivered for processing in [C]. The kube-apiserver delivers the notification of the pod creation to the kube-scheduler, and in [D], the kube-scheduler finds an appropriate node for the newly created pod; it then proceeds to request an update to the pod object to add the information concerning the bound node.

④ Path [A] and [B] result in the pod object being updated with node binding information and a notification of its update delivered for processing in [C]. At this point, the deployment process is complete in the control plane and the node is notified of the new pod it has been assigned.

### 2.3.1 The Deletion Process

The removal of a deployed application works similarly to the deployment: the operator requests the deletion of the deployment object stored in etcd. This action triggers a deletion cascade that deletes all objects created during the application deployment. The deletion process involves only three phases, the equivalent of phase (1), (2) and (3). The last phase is not required as the scheduler is not involved in the deletion process.

# Chapter 3

# Related Work

Kubernetes is a large system with many performance characteristics that can be evaluated. In this large space, there is little existing literature that touches on the topic of application deployment throughput with a focus on control plane performance. First, this section presents work related to general Kubernetes benchmarks and performance improvements, including some benchmarks on end-to-end deployments and the importance of etcd in different behaviours of the Kubernetes systems. Second, there is an overview of work comparing the performance between Kubernetes and other orchestration systems. Finally, there is specific work that discusses potential changes in etcd and specific guidelines on etcd usage from a Kubernetes controller point of view so as to maximize performance.

**Kubernetes performance and benchmarks.** An Uber team has evaluated the impact of event object creation in Kubernetes during pod deployments [45]. They observed a significant change in deployment time – going from several minutes to 30 seconds with a 40k pods and 8k node setup – when events were no longer generated at all in the control plane. The organization of containers into pods [35] is explored, e.g., the cost of one container per pod compared with all containers in a single pod, and a model is offered so that Kubernetes users can choose to combine pods and containers to best suit their applications; the authors perform benchmarks to measure deployment time and provide brief end-to-end latency measurements but retain the focus of their measurements on pod and container combinations, not on end-to-end latencies.

A development team at PayPal explores increasing the number of nodes in their production environment to 4000 nodes and 200,000 pods. They find that tuning different hardware components and Kubernetes configurations allows a deployment rate of 3000 pods/minute with a 99th percentile latency of 5 seconds, which includes control plane as

well as container deployment on the worker node. Other benchmarks are performed that compare pod routing technologies (native vs Itsio [26]) and their relationship to etcd for apps already deployed within the cluster [31]. They find that etcd performance, based on the underlying disk IOPS, impacts the performance of both cluster routing technologies for already deployed applications.

**Other orchestration systems.** A comparison between Nomad [37] and Kubernetes is performed to evaluate which system has less overhead when deploying a *job* type of deployment [5], finding that Nomad outperforms Kubernetes when deploying *jobs* in their respective cluster.

Another comparison is performed [40], between Kubernetes and Docker Swarm [41]; the author measures latency for container deployment times under different cluster loads, i.e., number of containers deployed per node. They find that Docker Swarm outperforms Kubernetes at even low cluster load; the author notes that from a performance perspective, Docker Swarm's much simpler architecture affords it less inter-component communication.

**Etcd store.** A proposal for rearchitecting Kubernetes [22] outlines swapping out etcd store for an equivalent data store that presents similar APIs and uses *conflict-free replicated datatypes* to eliminate write latency caused by the inter-etcd coordination required to reach consensus. There are also recommended ways to use the kube-apiserver API – which in turn affects how etcd is accessed – to improve etcd performance [15].

# Chapter 4

# Architectural Change

In this chapter, changes to the Kubernetes architecture are described that increase its control plane's ability to handle application starts and stops. The fundamental use case, explored in detail in Chapter 5, is that the control plane receives a stream of requests to start and/or to stop applications in the Kubernetes cluster. The changes laid out in this chapter increase the deployment and deletion throughput that the control plane can sustain as well as improving the per-request latencies.

## 4.1 Essential Etcd Writes

The application deployment process outlined in Section 2.3 describes how the deployment of one application involves four writes to etcd in total: the creation of three objects (deployment, replicaset, pod) and an update to the pod object assigning it to a node. These writes are necessary but are not the only ones that etcd experiences during an application deployment. Controllers generate non-essential etcd writes, both object creation and object updates. A write is considered non-essential if it is not required for the application deployment process to complete.

One type of non-essential etcd write is the creation of 'event' objects by controllers as part of their normal operation, which are not unlike log entries. For instance, the deployment controller generates informational events (such as indicating that a replicaset has been successfully created) or warning events (indicating there was an issue with creating a replicaset).

The other type of non-essential etcd write involves controllers updating objects to include status or progress updates. These updates do not impact the deployment process and are human-readable. For example, the deployment controller updates the deployment object to show progress of the associated replicaset. Presumably, keeping the deployment object up-to-date with the most recent state of the application deployment process as a whole (including replicaset and pod status) allows the operator to only have to consult the deployment object to glimpse the deployment progress; alternatively, this exempts Kubernetes clients (e.g. kubectl) from having to inspect several objects before being able to present the state of the application deployment process.

For the scenarios investigated in this work, the ratio of essential to non-essential etcd writes is approximately 1:2, such that about only 33% of etcd writes are essential to the deployment process.

## 4.2 Storing Non-Essential Writes in Secondary Store

The architectural change to Kubernetes is the use of an additional etcd instance to store non-essential etcd writes. This only changes the relationship between kube-apiserver and etcd, adding a second etcd instance to the cluster, while no other cluster component communication patterns are affected. This change implies modifying kube-apiserver such that it classifies a component's write request as being either essential or non-essential, and then proceeds to write essential writes to the primary etcd store and non-essential writes to the secondary etcd store.

## 4.3 Implementation

The implementation of this architectural change involves running a secondary etcd instance, categorizing essential and non-essential etcd writes and modifying Kubernetes code such that non-essential writes are sent to the secondary etcd instance. The use of both a primary and secondary etcd store within the Kubernetes cluster, which improves cluster performance, is called dual-store configuration (DSC) while the traditional use of a single etcd instance (i.e., solely using the primary etcd store) is named single-store configured (SSC). To toggle Kubernetes between using and not using the secondary store (i.e., staying in SSC mode or switching to DSC and using the secondary store), an additional HTTP endpoint is added to kube-apiserver.

### 4.3.1 Tagging Requests for Secondary Store

Tagging write requests as essential or non-essential requires minor code changes performed in three places: kube-controller-manager, the client-go library, and kube-apiserver. Figure 4.1 illustrates the process described in this subsection. In kube-controller-manager, through code inspection, the calls that request kube-apiserver to perform etcd writes are identified as either essential or non-essential. Because this write categorization (essential/non-essential) happens in kube-controller-manager, the information as to the essentiality of each etcd write must be included with the write request to kube-apiserver. Kube-controller-manager uses the client-go library to generate write requests to kube-apiserver and there is also a context object passed from kube-controller-manager to client-go library.

As such, the first code change is in kube-controller-manager: it involves the inclusion of the essential/non-essential nature of write requests in the context object which is passed on to the client-go library. The second code change is in the client-go library: it involves inspecting the context object (that may be empty), and if it identifies the essentiality of the write request, the client-go library will include an additional URL parameter in its HTTP-based write request to kube-apiserver. Finally, the last code change is in kube-apiserver: it must now inspect incoming HTTP write requests for the presence of an additional URL parameter, which indicates whether the write request is meant for the primary or the secondary etcd store. The only exception to this process are 'event' writes, which create event objects. Because all event writes are non-essential, no code inspection is required; instead an additional code change is performed in kube-apiserver that identifies event writes – event writes all use a distinct HTTP endpoint which facilitates their identification.

Finally, within kube-apiserver, the received write requests are now identified as essential or non-essential at the HTTP request handler layer of the code. The write request is wrapped by kube-apiserver in an internal *write object* which also embeds a context object. A code change is made such that that context object carries with it the essentiality of the write request. The write object is passed to the lower levels of kube-apiserver until it reaches the storage layer, where kube-apiserver interacts directly with both etcd stores. The context object also reaches the storage layer and is used to make a decision as to which store will receive the write.

### 4.3.2 Connecting Kubernetes to Secondary Etcd Store

A secondary etcd store instance is run alongside the primary etcd instance. Kube-apiserver code is changed, reusing code that initialises the primary etcd client to create a second

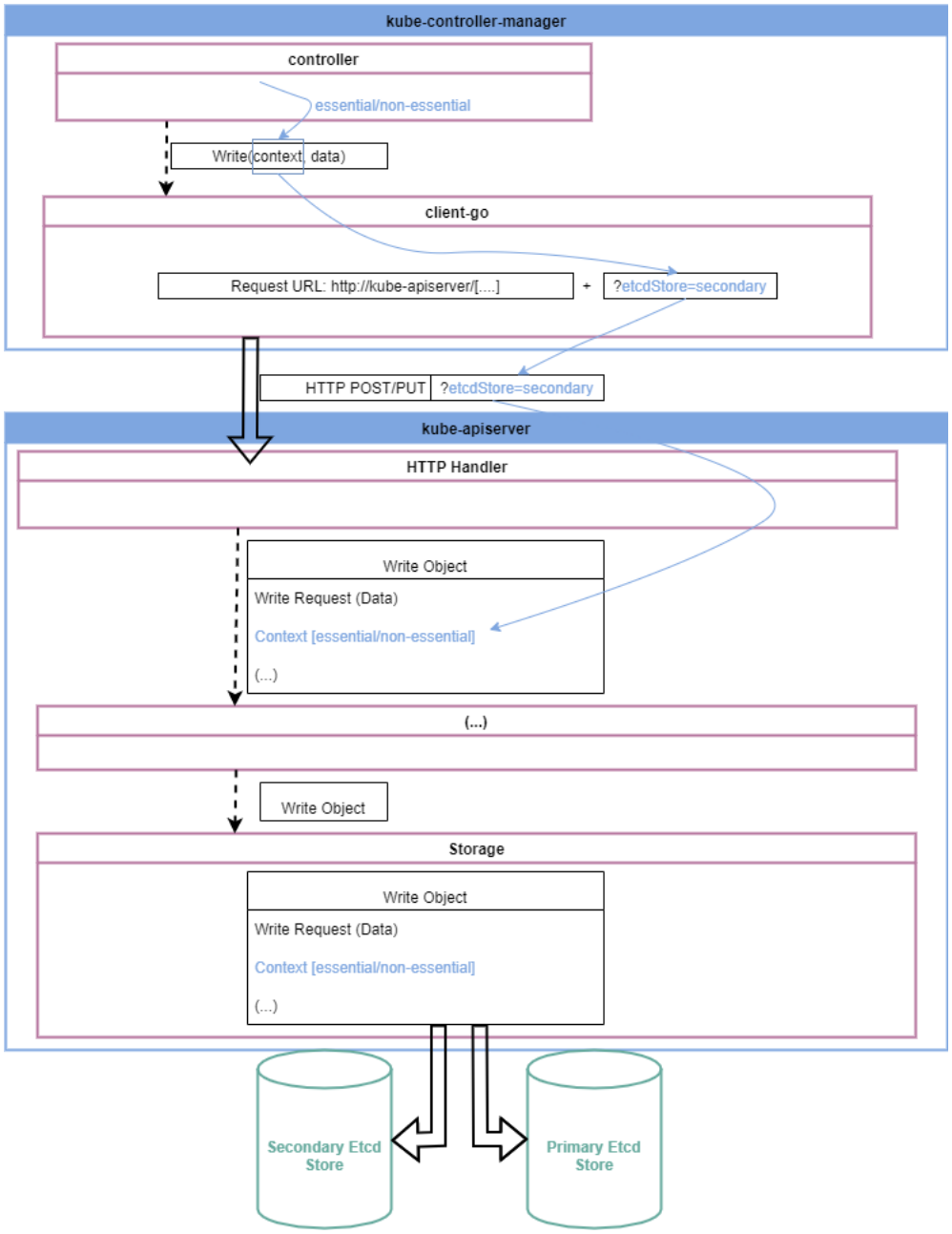


Figure 4.1: Tagging write requests, request flow.

etcd client for the secondary instance. The code for the storage layer in kube-apiserver – the code accessing etcd – is further changed with conditional statements that take into account whether the system is running in SSC mode or DSC mode. If running in SSC mode, all write requests received from upper layers and meant for etcd are sent to the primary etcd instance. If running in DSC mode, additional code in the storage layer also verifies whether the context object embedded within the write object indicates whether the write is essential or not essential. A write can only be sent to either the primary store or the secondary store, not both. An essential write is sent to the primary store while a non-essential write is sent to the secondary store. Kube-apiserver has also been modified to include an additional HTTP endpoint that sets an indicator on whether the system is running in DSC mode or SSC mode.

# Chapter 5

# Evaluation

## 5.1 Setup

The experiments performed measure the performance characteristics of the Kubernetes control plane when deploying and deleting applications from the cluster. They showcase the difference in deployment and deletion throughput and request latencies when comparing the single-store configuration (SSC) with the dual-store configuration (DSC). A *creation burst* experimental run starts with the Kubernetes cluster devoid of any application; over a 5 minute period and at regular time intervals (e.g., every 10ms), the client requests the deployment of an application. A *steady state* experimental run starts with a Kubernetes cluster already containing deployed applications; over 5 minutes and at regular intervals, the client alternates between requesting a new application be deployed to the cluster and requesting an existing application be deleted from the cluster. The underlying storage that etcd uses in these experiments is configured to be memory-backed, eliminating disk latency considerations. Disk latencies can vary (e.g., SSD, RAID, cloud remote block storage). Eliminating it allows the focus of the experiment to be on the latencies incurred by both the Kubernetes internals and etcd internals during application deployments and deletions. Additional experimental runs are performed with a synthetic latency added to the memory-backed etcd storage to simulate rotating disk latencies. Both the Kubernetes code [1] and the tools are available [2].

To study the ability of the SSC and the DSC to maintain different application deployment/deletion throughputs, the interarrival time of requests is varied, generating different

[1] https://git.uwaterloo.ca/taionita/kubernetes-thesis-code
[2] https://git.uwaterloo.ca/taionita/kubernetes-thesis-tools

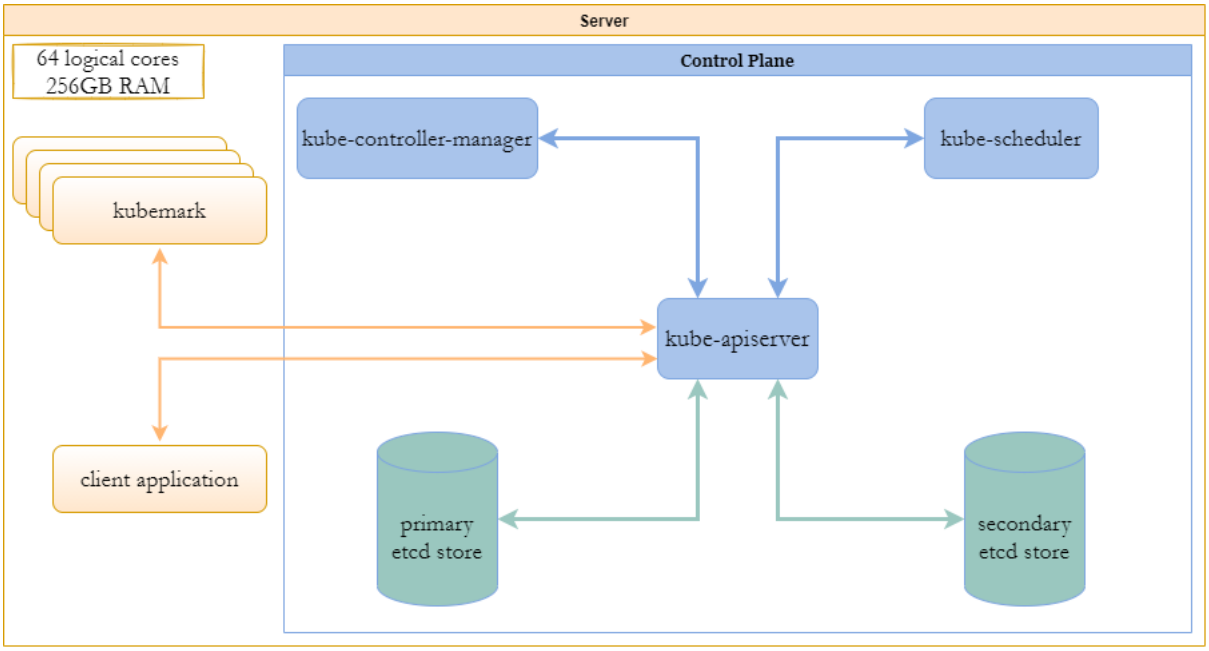


Figure 5.1: Experimental setup.

loads in different experiments. Finding the threshold after which the control plane is unable to sustain a higher client load requires exploring a plenitude of interarrival times in different experiments, for both the SSC and the DSC. Each experiment – for a particular store configuration (SSC, DSC), mode (creation burst, steady state) and interarrival time – is repeated three times to ensure the observed throughput is not an anomaly. Repeating an experiment more than thrice did not bring significant changes to the results: repeated experiments with upwards of seven runs were performed at several interarrival times and the resulting throughput (e.g. failure threshold) and latencies did not change significantly when compared to three runs.

All experiments are run on a single server machine, equipped with 256GB of RAM and four Intel E5-4610 v2 @ 2.30GHz CPUs, for a total of 32 physical cores and 64 logical cores. All control plane components, including etcd, are run as single instances; there is no cluster of etcd instances nor are any Kubernetes components running in a high-availability or hot-standby mode. Memory-backed directories serve as storage space for the primary and secondary etcd stores. Figure 5.1 illustrates the modified inter-component communication in the cluster with the presence of a secondary store.

The core part of an experimental run is the kube-apiserver receiving requests from the

client application to deploy and delete applications from the cluster. Before starting the client application, the experimental setup script cleans up any previously created application deployments, as well as cleaning up any remaining data stored in the primary and secondary etcd instances using the *etcdctl* tool. It then compacts and defragments etcd, restarts all Kubernetes control plane components (and allows them to repopulate their cache) to ensure all experimental runs start on equal footing. Finally, it starts several kubemark instances, which register as kubelet-running nodes in the cluster. This makes the nodes available to kube-scheduler so that it can assign pods to them. After the kubemarks have registered themselves, they are shut down; they continue to be seen as available by kube-scheduler – and thus available for scheduling purposes – due to a 1 hour grace period despite the lack of heartbeats.

While the client application is generating requests, CPU and memory utilisation is measured for every Kubernetes component using the perf and pcp-pidstat utilities. Both utilities support measuring the usage of a specific process (or group of processes) as well as an interval mode which measures utilisation of the resource over a period of time; thus, both perf and pcp-pidstat measure (CPU and memory respectively) resource utilisation every second. To ensure that the measurement tools themselves do not interfere with the experiment, the CPU utilization of the perf and pcp-pidstat tools are measured and they were found to consume less than 1 CPU core. Finally, the CPU utilization of the client application is also measured, finding that it consumed less than half of a CPU core when generating requests at 3ms intervals.

### 5.1.1 Client Application

The client application is a custom Kubernetes load generator, written in Go, that makes use of Google's client-go library, the library used for inter-component communication within the Kubernetes cluster by the control plane components as well as kubectl. The client application has two modes of operation: *creation burst* and *steady state*. In both modes, the client application generates requests at a specific interval measured in milliseconds (e.g., one request every 5ms) as an open-loop system, to avoid coordinated omission. In the creation burst mode, the client is started with two arguments: a time interval in milliseconds and a total number $D$ of applications to be deployed. At every time interval, the client requests that the kube-apiserver deploy an application in the cluster; it continues until it has requested $D$ applications be deployed. All deployed applications are identical except for their name, which serves to uniquely identify them within the cluster in the default namespace.

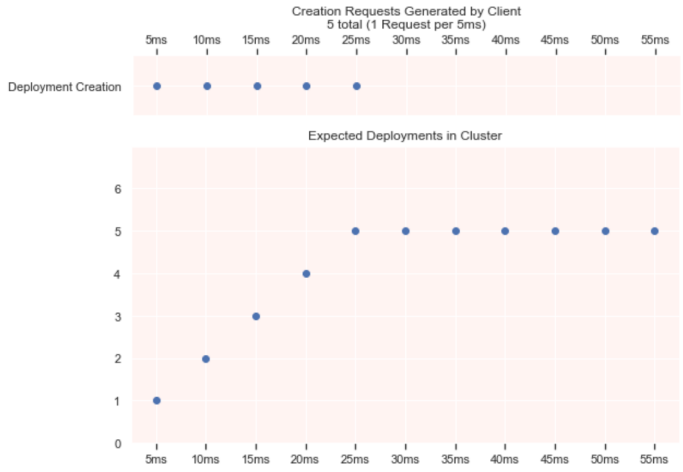


Figure 5.2: Client Requests for Creation Burst.

In the steady state mode, the client first requests the creation of a number of initial application deployments, behaving much like in the creation burst mode during this initial phase. After the initial phase, the client alternates between creating an application deployment and deleting an application deployment at every request interval. With this behaviour the client maintains a fixed number of deployments in the cluster. Steady state mode requires three arguments: the time interval, total amount of deployments, and the number of deployments when alternating between creation and deletion.

In Figure 5.2, for the creation burst mode, there is an example of a timeline for 5 generated requests and the resulting expected number of applications deployed in the cluster.

In Figure 5.3, the graph is an example of a timeline of for 15 generated requests, with a total of 11 creation requests and 4 deletion requests. The steady state's initial number of deployments is 7 deployments, while the number of alternating deployment is 2 deployments (2 deletions followed by 2 creations).

### 5.1.2 Requested Deployments

The deployment requests generated by the client application specify several fields and options. The corresponding manifest is shown in Figure 5.4 in its YAML format for ease of reading.

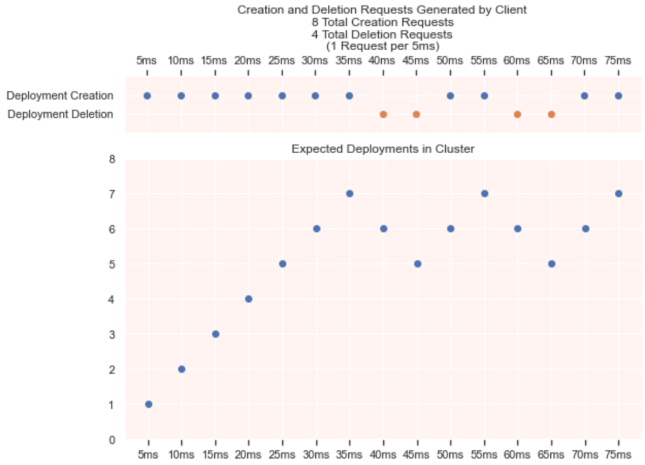


Figure 5.3: Client Requests for Steady State.

The ${INDEX} placeholder in the `metadata.name` field is used to avoid collisions between deployments, as Kubernetes does not accept two deployments with the same name; the client application substitutes ${INDEX} with a monotonically increasing integer as it requests new application deployments.

This manifest ultimately results in the creation of, at a minimum, one deployment object, one replicaset object and one pod object; the pod object receives at least one update that specifies which node it is bound to.

## 5.2 Experimental Results

The experiments compare Kubernetes application deployment performance when using the single store configuration (SSC) versus the dual store configuration (DSC). Specifically, the experiments explore the ability of both configurations to maintain application deployment throughput (number of applications deployed per second in the control plane) as well as the application deployment latencies. Because the client operates by sending requests to the kube-apiserver at millisecond-precision intervals, the generated load and throughput are reported using the interarrival time of client requests as well as the application deployments (or deletions) per second (app/s) when appropriate. Figures showcasing throughput measurements and machine CPU utilisation feature a plotline with error bands coloured a solid blue and orange; the error bands are the shaded semi-transparent light-coloured areas above and below the solid plotlines.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-${INDEX}
  labels:
    app: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
        particular_instance: nginx-${INDEX}
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
```

Figure 5.4: Deployment Manifest.

Requests have two latency metrics associated with them. One is the latency of the total completion time of a request: the end-to-end latency. The second is the total etcd latency, which is the sum of all etcd latencies incurred when performing the writes to etcd required during an application deployment or deletion, i.e., the sum of the time elapsed for each phase during code path B (see Section 2.3). Finally, in the steady state scenario, the latencies are further split based on whether the request is for an application deployment or an application deletion. Unlike an application deployment request, a deletion request needs one less write to etcd and thus one less use of the notification pipeline (see Subsection 2.3.1).

### 5.2.1 Creation Burst Throughput Results

At a interarrival time of 26ms, Figure 5.5 illustrates both the SSC and DSC being able to keep up with the generated load and maintain a throughput of 38.46 app/s. At 25ms interarrival time (Figure 5.6), the SSC fails to maintain the expected throughput, unlike the DSC, throughout the length of the experiment lasting 300 seconds, slowing down around 285 seconds. The interarrival time of 14ms is the smallest time at which the DSC is still able maintain a steady throughput (Figure 5.7) for a load of 71.43 apps/s, 1.86X the maximum throughput that the SSC can sustain. Finally, the DSC is unable to keep up with the load generated by an interarrival time of 13ms, as seen in Figure 5.8.

### 5.2.2 Creation Burst Latency Results

The latency experienced by deployment requests for both the SSC and DSC is shown for an interarrival time of 26ms or 38.46 app/s – the highest throughput that the SSC can sustain. Figure 5.9 presents the end-to-end latency distribution: it shows the DSC has a tighter distribution. The 99th latency percentile are 144ms and 96.7ms for the SSC and DSC respectively, while the 99.9th latency percentile are 181.2ms and 119.2ms, SSC and DSC respectively. Figure 5.10 illustrates the total etcd request latencies: while the SSC has a better average latency, it has a much more pronounced tail latency. This is clear in the 99th latency percentiles, which are 9.5ms and 7.4ms (SSC and DSC respectively), the 99.9th latency percentiles, which are 26.4ms and 10.3ms (SSC and DSC respectively) and finally the max latencies are 107.8ms and 14.2ms (SSC and DSC respectively).

### 5.2.3 Steady State Throughput Results

Figure 5.11 shows that both the SSC and DSC are able to maintain an application deployment throughput at a request interarrival time of 15ms (resulting in 33.3 app/s). When decreasing the interarrival time to 14ms, the SSC fails to maintain the expected throughput (Figure 5.12), with a declining throughput starting at around 240 seconds and a catch-up period after the end of the experiment at the 300 seconds mark. All the while, at 14ms interarrival, the DSC is able to maintain a steady throughput. Further decreasing interarrival time to 13ms (Figure 5.13) shows the SSC throughput failing much earlier, around 150s, with an even longer catch-up period after the 300 second mark. At a interarrival time of 9ms (Figure 5.14), the DSC is still able to keep up, with a throughput of 55.56app/s which is 1.67X higher than the best throughput the SSC maintained. Finally, at a interarrival time of 8ms (Figure 5.15), the DSC fails to maintain the expected throughput. Figure 5.16 illustrates the throughput for at 6ms interarrival time: while both the DSC and SSC are unable to maintain throughput, the DSC is able to work through its backlog of requests significantly faster.

The performance for the application deletion throughput mirrors the application deployment results. The SSC and DSC both maintain a throughput at 15ms interarrival time with the SSC failing at 14ms. The DSC is able to match the load at 9ms and finally fails at 8ms.

### 5.2.4 Steady State Latency Results

The deployment request latency experienced by the SSC and DSC is explored for the load with 15ms interarrival time, the highest load that the SSC can sustain. Figure 5.17 presents the end-to-end latency distribution of all application deployment requests. The DSC distribution is bimodal with peaks around 80ms and 115ms, with a tail end reaching 200ms. The SSC distribution is tripodal, with two important peaks at 80ms and 120ms and a third smaller peak at 550ms, near the tail end that reaches 600ms. The 99th latency percentile are 176ms for the DSC and 569ms for the SSC.

The deployment etcd latency, at 15ms, experienced by both the SSC and DSC is shown in Figure 5.18. The DSC has a narrower distribution and a less prominent tail end than the SSC. The 99th latencies percentiles at this load are 9ms and 13ms for the DSC and SSC respectively, and the 99.9th percentiles are 17ms and 31ms (DSC and SSC respectively).

At 15ms interarrival time, the deletion requests have different 99th percentiles: the end-to-end 99th percentile are 48ms for the DSC and 437ms for the SSC, while the total etcd 99th percentile latencies are 6.7ms and 9.9ms for the DSC and SSC respectively.

### 5.2.5 Steady State Latency Results with Synthetic Disk Latency

Adding a synthetic disk latency of 8ms to etcd's memory-backed storage allows a comparison of the effects of a spinning disk drive. The comparison is performed for a 15ms interarrival time at which both the SSC and DSC are able to maintain the throughput. Figure 5.19 shows the end-to-end latency while Figure 5.20 shows the total etcd latency. As expected, the etcd latency changes significantly with the synthetic latency. The etcd latency distributions shift for both DSC and SSC by about one order of magnitude: from a range of approximately 4-8ms (without synthetic disk latency) to a range of 50-150ms (with synthetic disk latency). This additional etcd latency is reflected in the end-to-end request latency: the end-to-end latency distributions shift for both DSC and SSC by approximately the absolute amount of additional etcd latency caused by the synthetic disk latency. The supplementary etcd latency caused by the 8ms disk latency approximately doubles the end-to-end request latency for both DSC and SSC.

### 5.2.6 CPU and Memory Utilization

Throughout all experiments, both CPU and memory usage are recorded. When either the DSC or SSC are unable to keep up with the generated load, neither CPU nor memory

usage are the bottleneck; that is, of all the cores available to the Kubernetes control plane, less than 40 are used by the control plane in runs where throughput cannot be sustained. Similarly, memory usage is far under the memory available: less than 10GB are used. Figure 5.21 shows the CPU usage during a steady state experiment with a 13ms interarrival time, a load at which the SSC's throughput is unable to keep up as early as 120s into the 300s experiment.

Figure 5.22 shows the CPU usage at an interarrival time of 15ms in a steady state experiment; this is a load that both the DSC and SSC are able to sustain. We can see the DSC requires less CPU resources throughout the experiment. The SSC uses nearly 50% more cores than the DSC at the start of the experiment and reaches nearly double the core count at the end of the experiment.

## 5.3 Discussion

The CPU and memory measurements indicate that neither the SSC nor DSC are ever CPU or memory bound, and therefore performance differences between the two cannot be attributed to these resources; furthermore, there are no performance penalties owed to network communication. Finally, save for etcd, the control plane does not perform any disk I/O and therefore no costs are incurred from that. The disk storage etcd writes to is memory-backed so even this component suffers no disk I/O penalty. The remaining sources of slowdown can be attributed to Kubernetes architecture decisions and to etcd performance.

All things being equal, if etcd is subjected to fewer writes per second, it will perform better with respect to request latency. As the DSC decreases the amount of etcd writes that the primary etcd is subjected, etcd does show better overall latency distributions. In steady state mode, 15ms interarrival time, the 99th latency percentiles are 13ms and 9ms (SSC and DSC respectively) and the 99.9th are 17ms and 31ms (SSC and DSC respectively). In creation burst mode, at 26ms interarrival time, the 99th latency percentiles are 9.5ms and 7.4ms (SSC and DSC respectively), the 99.9th latency percentiles are 26.4ms and 10.3ms (SSC and DSC respectively) and finally the max latencies are 107.8ms and 14.2ms (SSC and DSC respectively).

Overall, while differing between the SSC and DSC, etcd latencies account for a small fraction of the time requests take to complete. They also do not account for the difference between SSC and DSC latencies, i.e., etcd latencies alone cannot explain why DSC performs better than SSC with respect to end-to-end request latencies. Indeed, when looking at the

99.9th percentile of etcd latencies, the SSC and DSC differ in the range of 10ms-20ms. However, in steady-state mode for the same interarrival times, the 99th percentile *end-to-end* latencies are 176ms and 569ms (SSC and DSC respectively).

Furthermore, as presented in Subsection 5.2.5, a 8ms synthetic disk latency was added to simulate a spinning-disk latency. The etcd performance page [**?**] recommends SSDs over spinning disks, which they claim exhibit a latency under 1ms. Nevertheless, even with these less-than-ideal 8ms disk latencies which significantly affect total etcd latencies, they only contribute 50% to the end-to-end request latency.

The most pertinent aspect of the Kubernetes control plane architecture that explains the improved performance is the notification pipeline which spans over the kube-apiserver, kube-controller-manager and kube-scheduler. Other parts of the system spawn a goroutine for new incoming requests or have a sizable pool of goroutines, but the notification pipeline in a series of stages where work is performed at each stage by a single goroutine. When using the SSC, all writes end up in the primary etcd store and the primary etcd store is directly monitored by the notification pipeline, which means that all writes result in a notification object traveling through the pipeline. The DSC on the other hand only subjects the primary etcd store to essential writes while non-essential writes end up in the secondary etcd store. Furthermore, the secondary etcd store is not monitored by the notification pipeline and therefore does not increase the number of notification objects being processed through it.

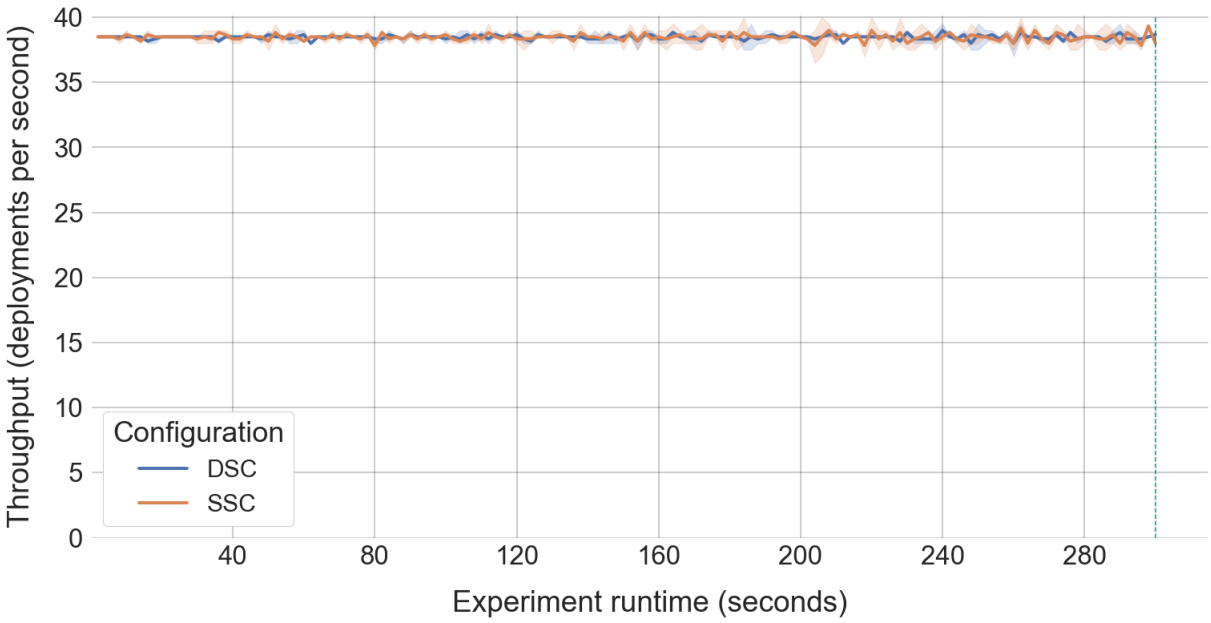


Figure 5.5: Creation burst mode, application deployment at 26ms interarrival time.

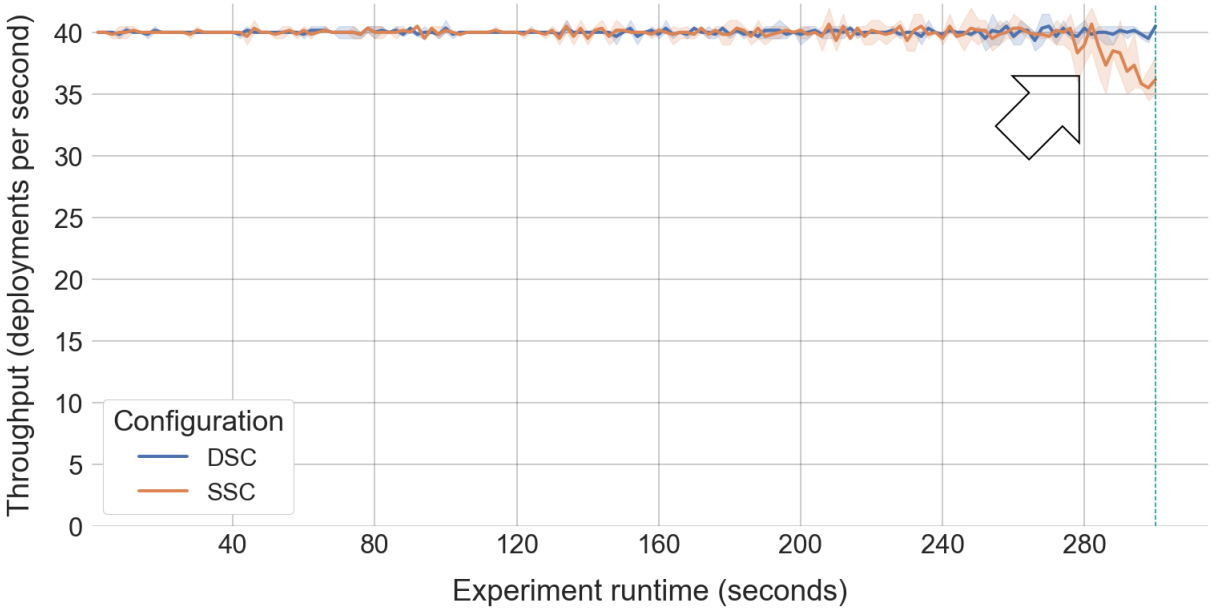


Figure 5.6: Creation burst mode, application deployment at 25ms interarrival time.

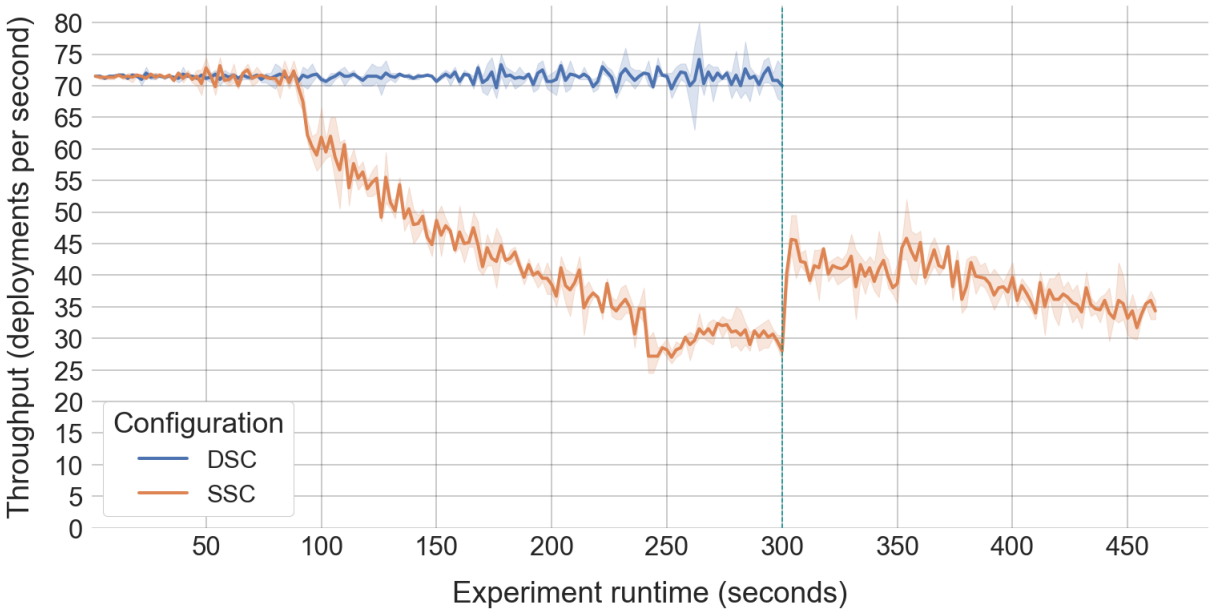


Figure 5.7: Creation burst mode, application deployment at 14ms interarrival time.

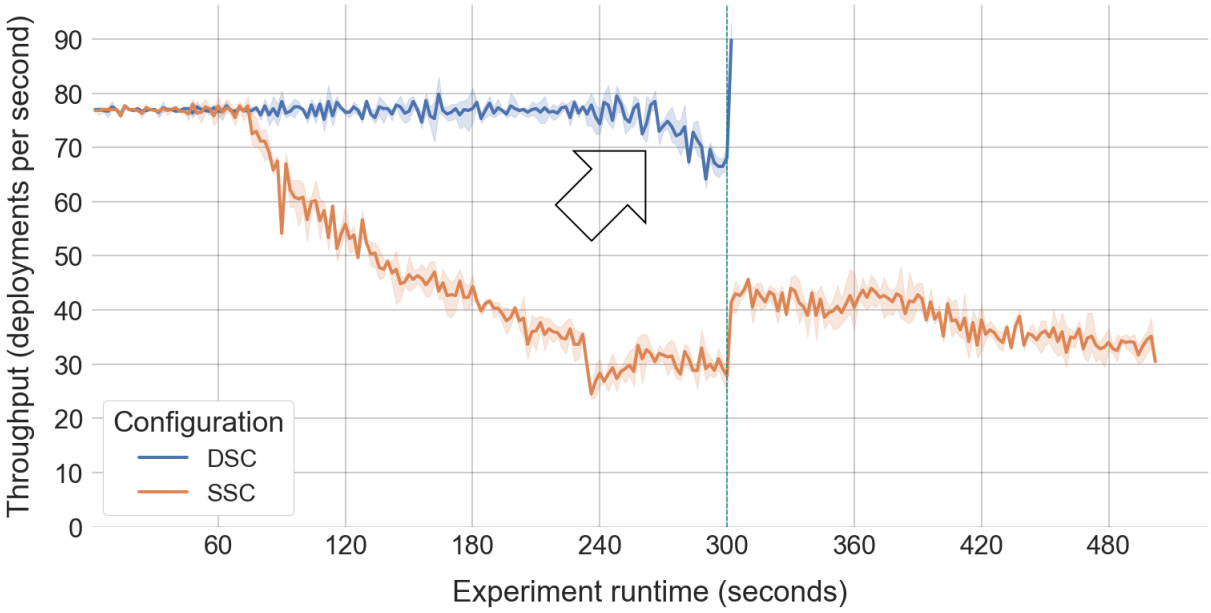


Figure 5.8: Creation burst mode, application deployment at 13ms interarrival time.

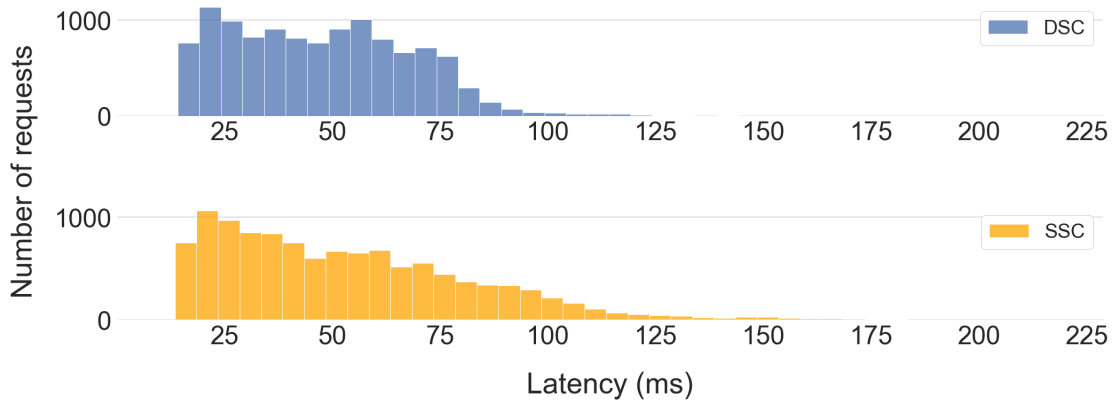


Figure 5.9: Creation burst mode, end-to-end latency, at 26ms interarrival time.

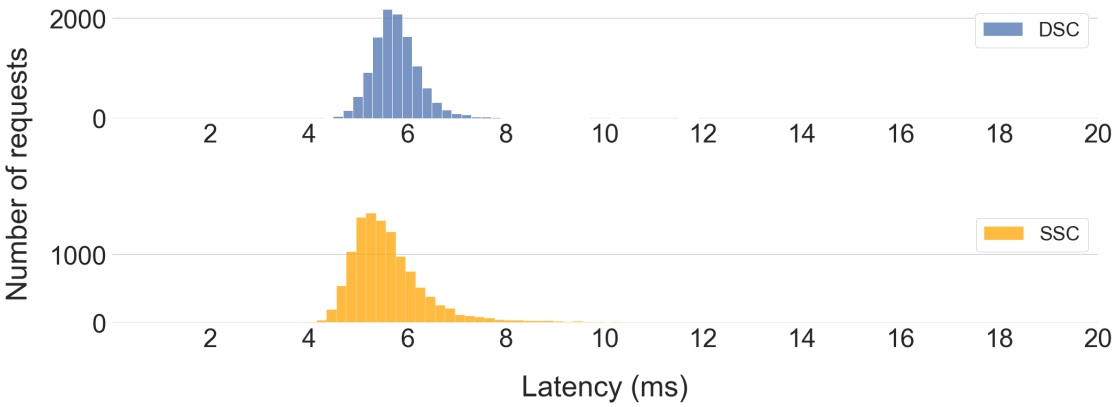


Figure 5.10: Creation burst mode, etcd latency, at 26ms interarrival time. X-axis is truncated as SSC latency data points reach into more than 100ms.

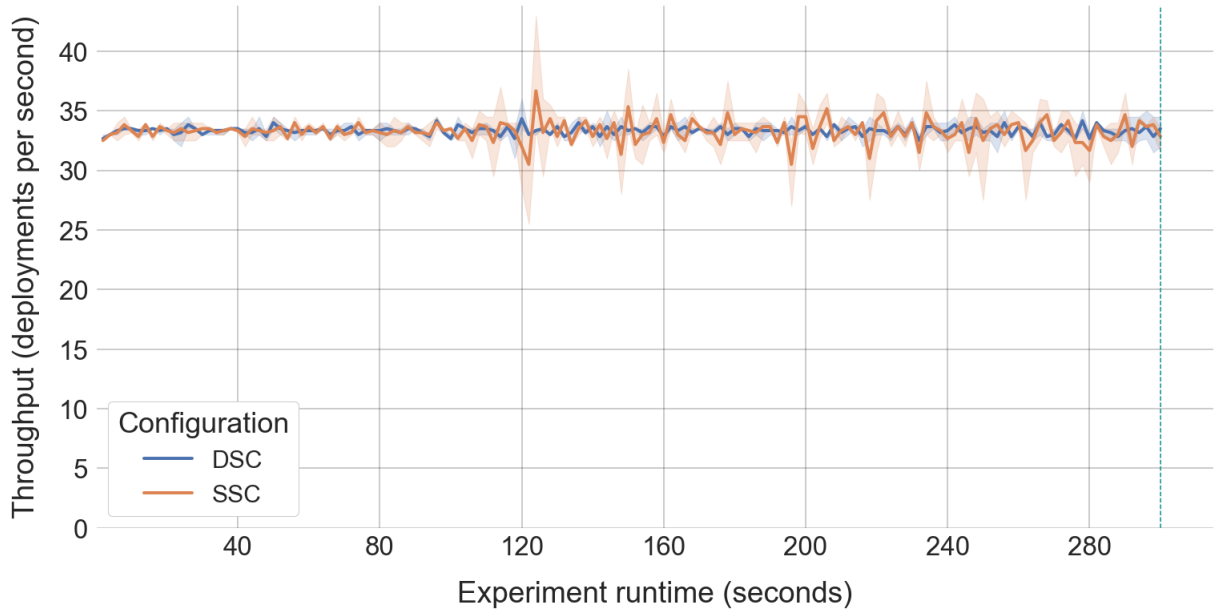


Figure 5.11: Steady-state mode, request at interarrival time of 15ms.

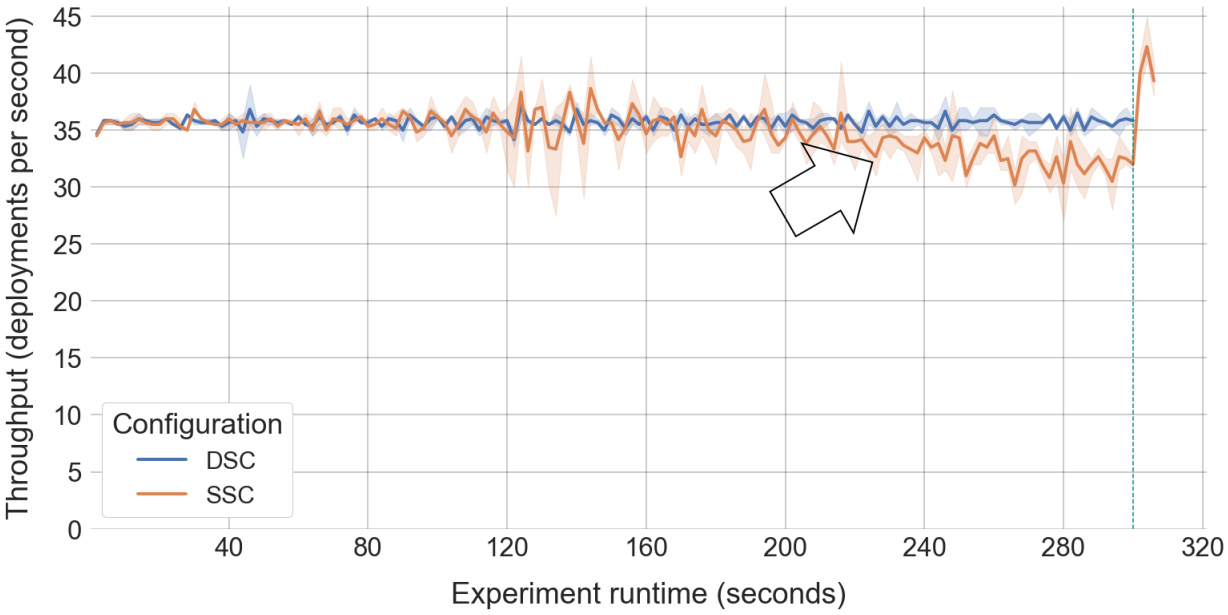


Figure 5.12: Steady-state mode, request at interarrival time of 14ms.

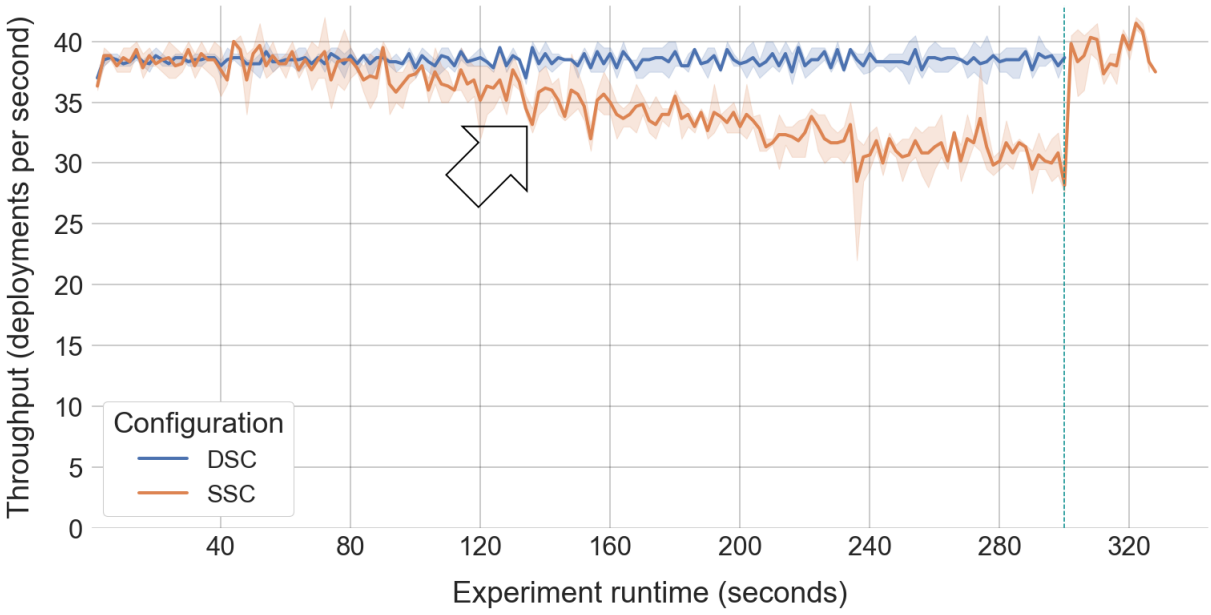


Figure 5.13: Steady-state mode, request at interarrival time of 13ms.

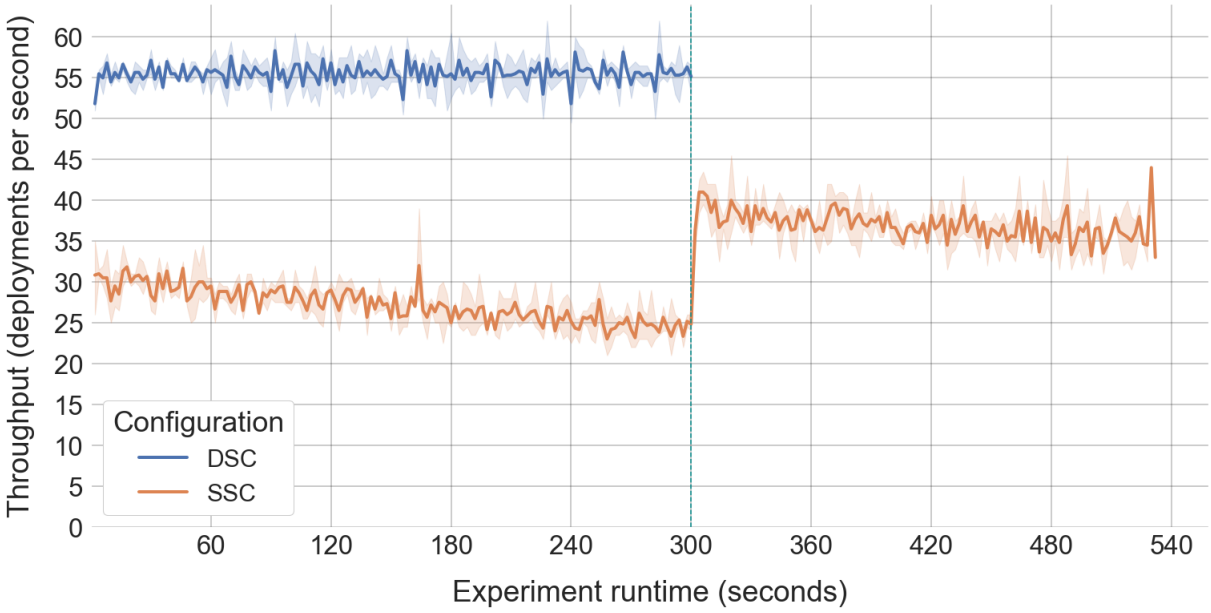


Figure 5.14: Steady-state mode, request at interarrival time of 9ms.

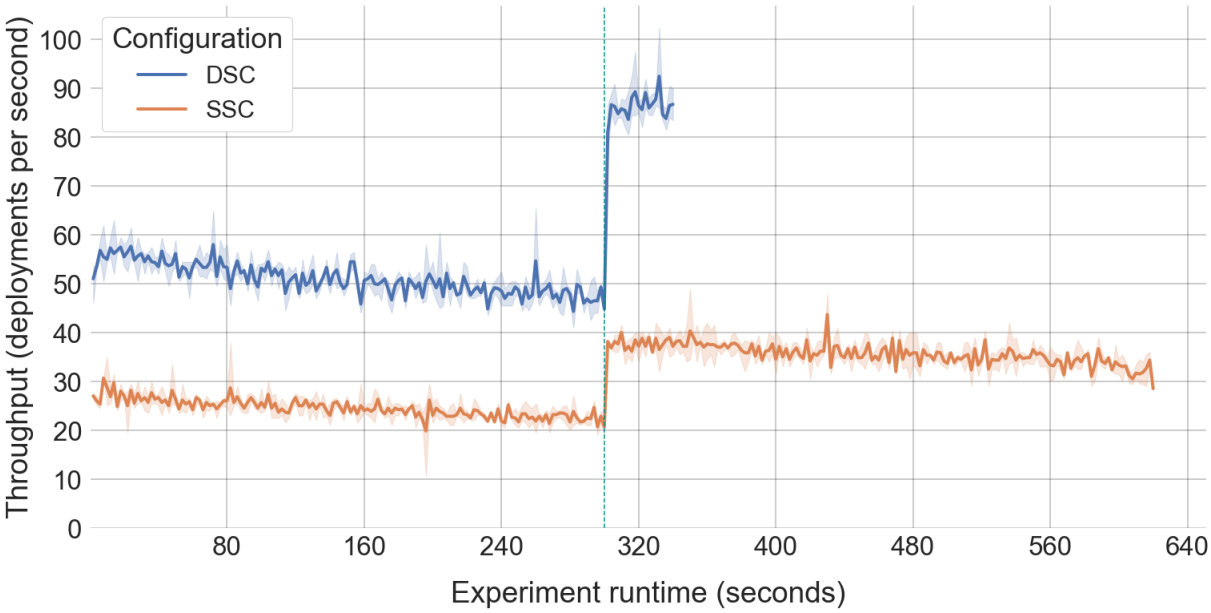


Figure 5.15: Steady-state mode, request at interarrival time of 8ms.

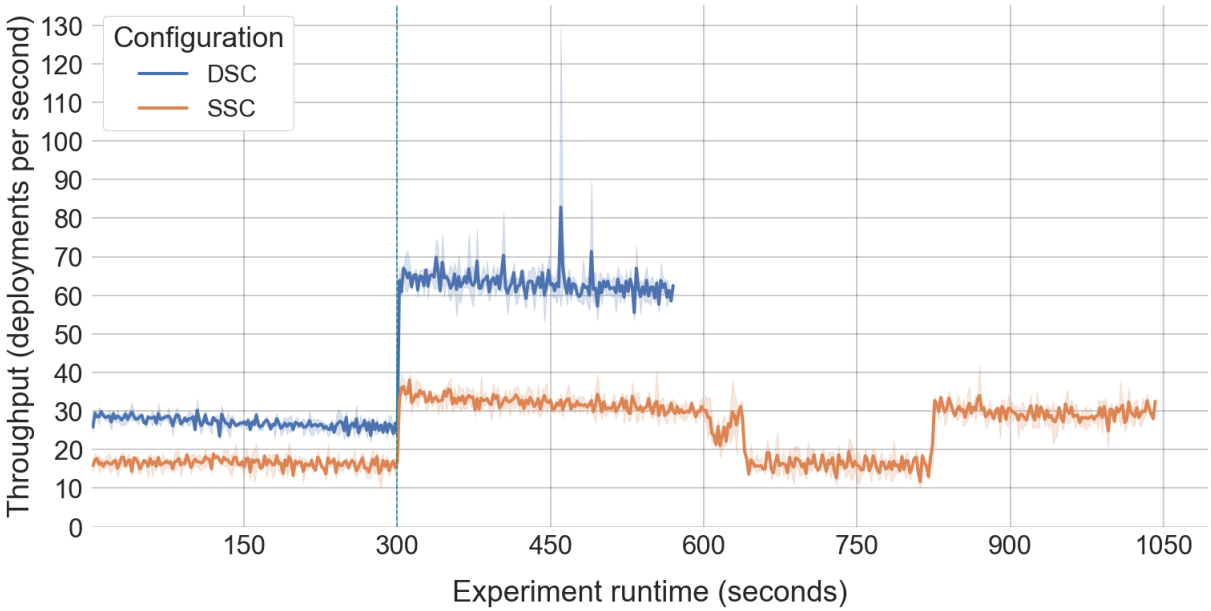


Figure 5.16: Steady-state mode, request at interarrival time of 6ms.

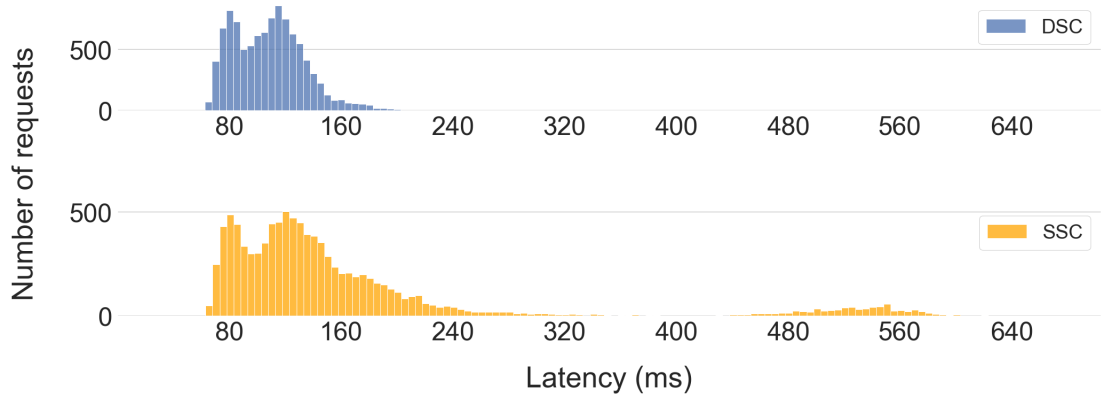


Figure 5.17: Steady-state mode, application deployment, end-to-end latency for an inter-arrival time of 15ms.

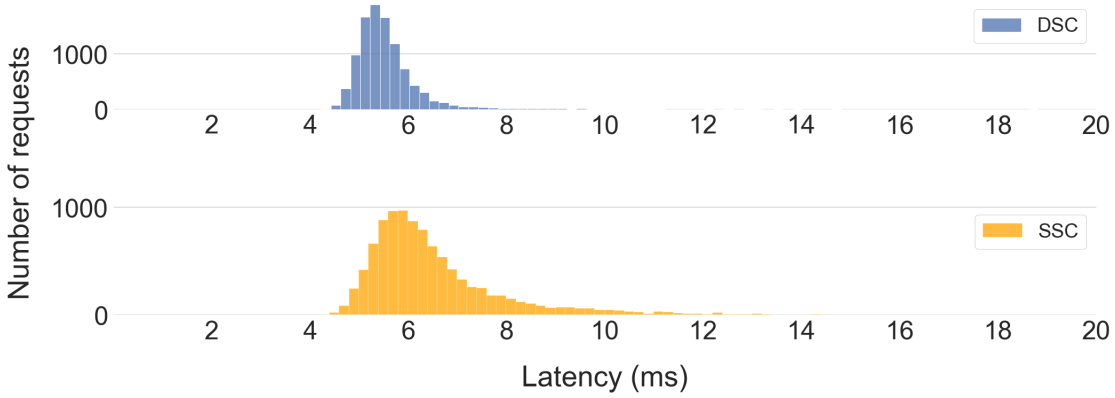


Figure 5.18: Steady-state mode, application deployment, etcd latency for an interarrival time of 15ms. X-axis truncated for clarity, SSC data points continue past 20ms.

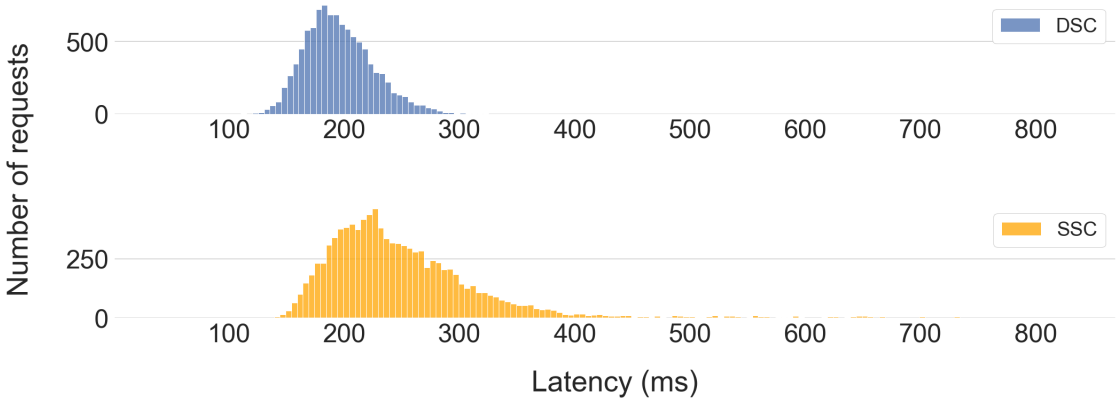


Figure 5.19: Steady-state mode with a 8ms synthetic disk latency, application deployment, end-to-end latency for an interarrival time of 15ms. The SSC latency tail end reaches 833ms while the DSC tail end latency reaches 322ms.

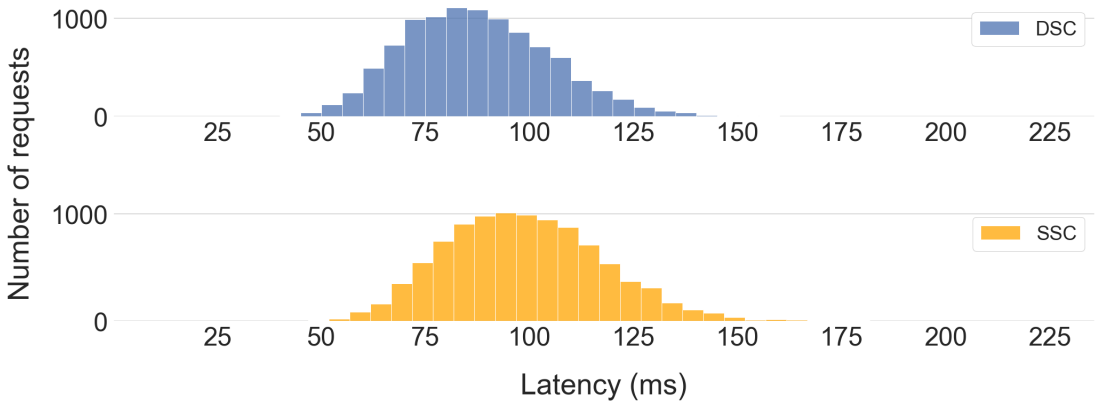


Figure 5.20: Steady-state mode with a 8ms synthetic disk latency, application deployment, etcd latency for an interarrival time of 15ms. The SSC latency tail end reaches 225ms while the DSC tail end latency reaches 165ms.

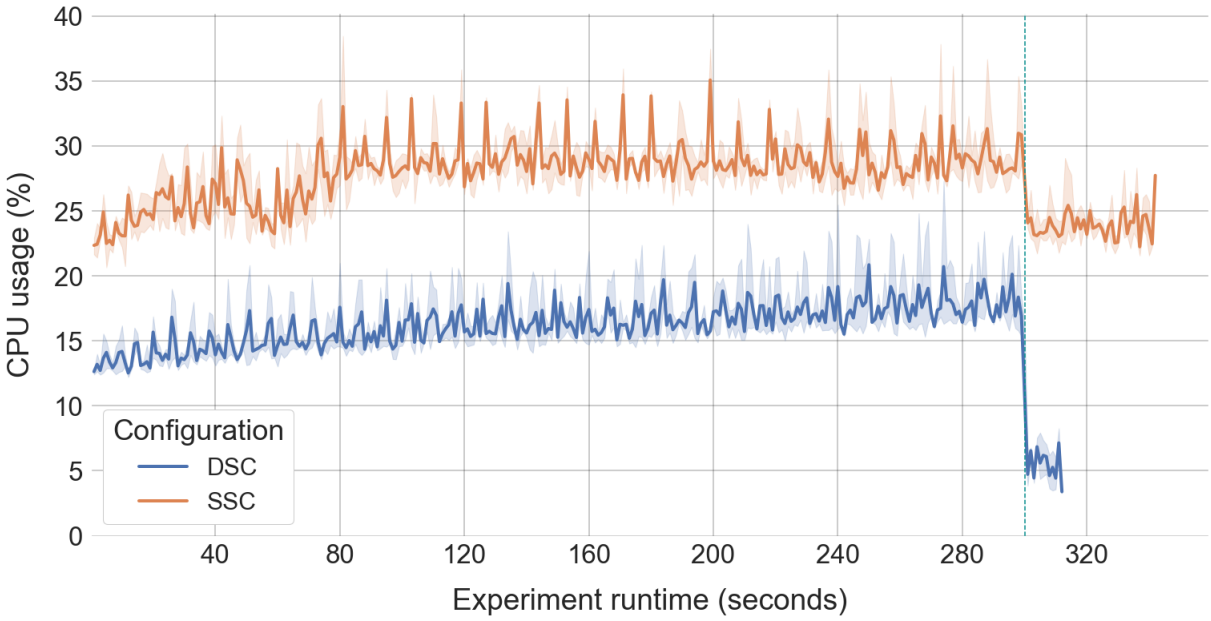


Figure 5.21: Steady-state mode, CPU usage, 13ms interarrival time.

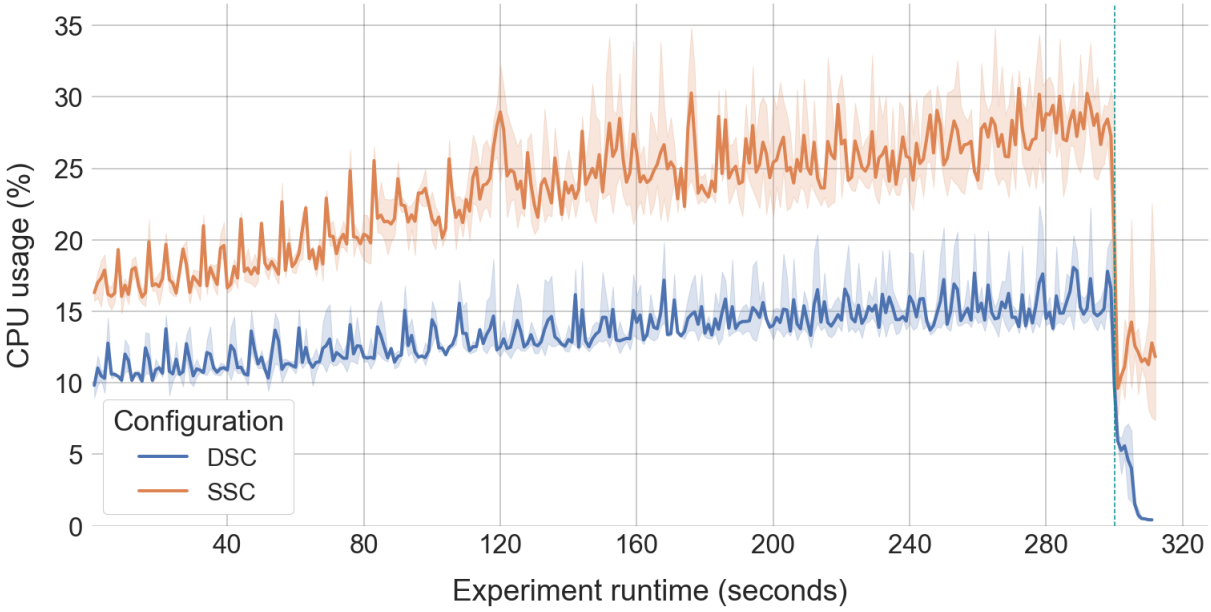


Figure 5.22: Steady-state mode, CPU usage, 15ms interarrival time.

# Chapter 6

# Conclusion

The MEC handoff process requires an orchestration system that is able to sustain a high throughput of application deployment and deletion. Kubernetes is such an orchestrator and its control plane is a major component that needs to handle application churn. The Kubernetes control plane, with its stateless and decoupled controllers, uses etcd as the primary dual-purpose mechanism to store information, by performing etcd writes, and to pass information between controllers using an event-based pattern that relies on on etcd notifications. Because any application deployment depends on a chain of object creations, a series of writes to the cluster state is required to both store the latest object in the chain and notify the controller responsible for the next object in the chain. As such, both the cluster write process and notification process are crucial to a high deployment and deletion throughput.

This work presents a modification to the Kubernetes architecture, namely identifying that cluster writes can be categorized as essential and non-essential, the latter resembling more system logs and updates for the cluster operator to understand the state of the deployments. By introducing a secondary store for non-essential writes, the primary store as well as the notification pipeline can be dedicated to the essential writes that are critical for the application deployment and deletion process. These two changes, accomplished by a modest code change, has increased the throughput capacity of the control plane by 67% in steady-state experiments where applications are deployed and deleted, and by 86% in creation burst mode where applications are only deployed. This change has reduced the end-to-end request latency and improved the etcd-specific latency characteristics. Finally, this thesis shows that end-to-end request latencies are significantly affected by the notification pipeline.

# References


[1] 3gpp specification set: 5g. https://www.3gpp.org/dynareport/SpecList.htm?release=Rel-15&tech=4.

[2] Imt 5g spider chart comaprison. https://www.etsi.org/images/articles/IMT-advanced-spider-chart.png.

[3] Saad Z Asif. *5G mobile communications : concepts and technologies*. Boca Raton, FL : CRC Press/Taylor and Francis Group, 2019.

[4] Aws global infrastructure. https://aws.amazon.com/about-aws/global-infrastructure/.

[5] Conductor: Why we migrated from kubernetes to nomad. https://thenewstack.io/conductor-why-we-migrated-from-kubernetes-to-nomad/.

[6] Akraino. https://wiki.akraino.org/.

[7] Intel smart edge. https://www.intel.com/content/www/us/en/edge-computing/smart-edge.html.

[8] Intel smart edge open. https://www.intel.com/content/www/us/en/developer/tools/smart-edge-open/overview.html.

[9] K3s. https://k3s.io/.

[10] Kubeedge. https://github.com/kubeedge/kubeedge.

[11] Wan Lei et al. *5G System Design: An End to End Perspective*. Springer, 2019.

[12] Luiz André Barroso et co. *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*. 2018.

[13] Github etcd-io/etcd benchmark. https://github.com/etcd-io/etcd/tree/v3.3.3/tools/benchmark.

[14] Etcd kv api guarantees. https://etcd.io/docs/v3.3/learning/api_guarantees/.

[15] Getting started with kubernetes — etcd performance optimization practices. https://www.alibabacloud.com/blog/getting-started-with-kubernetes-%7C-etcd-performance-optimization-practices_596294.

[16] Multi-access edge computing (mec). https://www.etsi.org/technologies/multi-access-edge-computing.

[17] Etsi standards: 5g. https://www.etsi.org/standards#page=1&search=5g&title=1&etsiNumber=1&content=1&version=0&onApproval=1&published=1&historical=1&startDate=1988-01-15&endDate=2021-12-20&harmonized=0&keyword=&TB=&stdType=&frequency=&mandate=&collection=&sort=1.

[18] Gcp global infrastructure. https://cloud.google.com/about/locations.

[19] Google cloud platform (gcp) cloud locations. https://cloud.google.com/about/locations.

[20] Ieee software defined networks – cord: Central office re-architected as a datacenter. https://sdn.ieee.org/newsletter/november-2015/cord-central-office-re-architected-as-a-datacenter.

[21] Comparison of networking solutions for kubernetes. https://machinezone.github.io/research/networking-solutions-for-kubernetes/.

[22] Andrew Jeffery, Heidi Howard, and Richard Mortier. Rearchitecting kubernetes for the edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '21, page 7–12, New York, NY, USA, 2021. Association for Computing Machinery.

[23] Kubernetes: Considerations for large clusters. https://kubernetes.io/docs/setup/best-practices/cluster-large/.

[24] Subtleties: Debugging an intermittent connection reset. https://kubernetes.io/blog/2019/03/29/kube-proxy-subtleties-debugging-an-intermittent-connection-reset/.

[25] Kubernetes components. https://kubernetes.io/docs/concepts/overview/components/.

[26] Itsio. https://istio.io/.

[27] Kubernetes control plane-node communication. https://kubernetes.io/docs/concepts/architecture/control-plane-node-communication/.

[28] Kubernetes. https://kubernetes.io/.

[29] Kubernetes slo. https://github.com/kubernetes/community/blob/master/sig-scalability/slos/slos.md.

[30] Kubernetes authorization overview. https://kubernetes.io/docs/reference/access-authn-authz/authorization/#determine-the-request-verb.

[31] Lars Larsson, William Tärneberg, Cristian Klein, Erik Elmroth, and Maria Kihl. Impact of etcd deployment on kubernetes, istio, and application performance. *Software: Practice and Experience*, 50(10):1986–2007, 2020.

[32] Kubernetes: Deployments. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.

[33] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B. Letaief. A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys Tutorials*, 19(4):2322–2358, 2017.

[34] Víctor Medel, Omer Rana, José ángel Bañares, and Unai Arronategui. Modelling performance and resource management in kubernetes. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, UCC '16, page 257–262, New York, NY, USA, 2016. Association for Computing Machinery.

[35] Víctor Medel, Rafael Tolosana-Calasanz, José Ángel Bañares, Unai Arronategui, and Omer F. Rana. Characterising resource management performance in kubernetes. *Computers and Electrical Engineering*, 68:286–297, 2018.

[36] Wahida Nasrin and Jiang Xie. A joint handoff and offloading decision algorithm for mobile edge computing (mec). In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2019.

[37] Nomad by hashicorp. https://www.nomadproject.io/.

[38] Openness. https://www.openness.org.

[39] Scaling kubernetes to over 4k nodes and 200k pods. https://medium.com/paypal-tech/scaling-kubernetes-to-over-4k-nodes-and-200k-pods-29988fad6ed.

[40] Evaluating container platforms at scale. https://medium.com/on-docker/evaluating-container-platforms-at-scale-5e7b44d93f2c.

[41] Docker swarm. https://docs.docker.com/engine/swarm/.

[42] Amazon wavelength and 5g. https://aws.amazon.com/wavelength/.

[43] Aws wavelength. https://techblog.comsoc.org/tag/aws-wavelength/.

[44] Amazon wavelength locations. https://aws.amazon.com/wavelength/locations/.

[45] Uber Yunpeng Liu. Online presentation – only slightly bent: Uber's kubernetes migration journey for microservices - yunpeng liu, uber. https://www.youtube.com/watch?v=91c3iUI2K7M&t=594s.