UNIVERSITY OF CAMBRIDGE
Department of Computer Science and Technology

CARICO: Scheduling via Federated Workload Capacity Estimation

Luca Choteborsky

Selwyn College

June 2025

Submitted in partial fulfillment of the requirements for the Computer Science Tripos, Part III

Total page count: ??

Main chapters (excluding front-matter, references and appendix): 1 pages (pp ??–??)

Main chapters word count: 467

Methodology used to generate that word count:

[For example:

```
$ make wordcount
gs -q -dSAFER -sDEVICE=txtwrite -o - \
   -dFirstPage=6 -dLastPage=11 report-submission.pdf | \
egrep '[A-Za-z]{3}' | wc -w
467
```

]

Declaration

I, Luca Choteborsky of Selwyn College, being a candidate for Part III of the Computer Science Tripos, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this report, I adhered to the Department of Computer Science and Technology AI Policy. I am content for my report to be made available to the students and staff of the University.

Signed:

Date:

Abstract

Kubernetes is a widely-used open-source container orchestration system that automates the deployment, scaling and management of containerized applications. Efficient scheduling is a critical component of Kubernetes, dictating an optimal allocation of pods across nodes to maximise a set of goals. Kubernetes schedulers generally fall into two categories: pod-descriptive and telemetric-based. Pod-descriptive schedulers, like the default `kube-scheduler`, rely on accurately defined Pod resource requests for optimal performance. In contrast, telemetric-based schedulers leverage collected Node metrics to identify under or over-utilized Nodes. While these are often used to refine decisions made by existing pod-descriptive schedulers, this dissertation focuses on exploring the potential of a telemetric-only scheduler.

Scheduling in Kubernetes without explicit Pod resource requests remains largely unexplored due to its complexity. Traditional bin-packing approaches are insufficient; instead, a telemetric-only scheduler requires a novel strategy to derive optimal plans from collected telemetry.

PRONTO offers a promising direction. This federated, asynchronous, and memory-limited algorithm schedules tasks across hundreds of workers. It enables individual workers to build local workload models from telemetry, which are then aggregated to form a global system view. For a telemetric-only scheduler, maximizing information aggregation is crucial. However, in very large clusters, centralizing all this data quickly becomes a bottleneck. By distributing the knowledge federation workload across the cluster, we can significantly reduce the load on a central scheduler.

Unfortunately, PRONTO's direct application to Kubernetes proved unfeasible due to its communication latency assumptions and early empirical findings. Consequently, I propose CARICO (Italian for "load"), a novel scheduler that shares PRONTO's core properties but accounts for communication latency. CARICO enables Nodes to perform Federated Singular Value Decomposition (FSVD) on capacity-based metrics, modeling past resource usage to estimate future workload capacity.

To evaluate CARICO, I implemented a prototype within the Kubernetes ecosystem. This involved extensive investigation into various metrics and filters to generate a precise capacity signal. Finally, I benchmarked CARICO's overall performance against the default

kube-scheduler on a Kubernetes cluster with diverse workloads. While CARICO achieved comparable throughput, it significantly outperformed kube-scheduler as a QoS scheduler, demonstrating lower Pod Completion times and improved workload isolation.

Acknowledgements

This project would not have been possible without the wonderful support of . . . [optional]

Contents

Chapter 1

Introduction

Datacenters serve as the foundational infrastructure for modern computing, housing hundreds of thousands of machines that underpin diverse user applications and data-parallel computations. The escalating demand for compute, memory, storage, and network resources necessitates increasingly large and complex datacenters. Datacenter scheduling is a critical task: it involves allocating available resources to workloads to ensure performance objectives are met with any small deviations resulting in potential revenue losses of millions of dollars []. The difficulty of this problem is compounded by ever-increasing input workload rates, dynamically changing workload characteristics, and the heterogeneity of resources. Schedulers are expected to deliver short user response times, high resource utilization, and high scheduling throughput simultaneously, making it one of the most challenging aspects of datacenter operation.

Numerous approaches have been explored to tackle task scheduling at a datacenter-level. Multi-dimensional resource allocation attempt to reduce scheduling to a bin packing problem, with many schedulers [**?** **?**] exploiting user-provided job resource requests to influence allocations. However, the problem of inaccurate resource estimation has necessitated complex techniques to better estimate resource requirements []. Another approach has schedulers predict the availability of node resources. These schedulers initially consisted of those that probed a portion of the datacenter on-demand [], and those that performed offline machine learning on telemetry from the entire datacenter []. While existing schedulers had to trade responsiveness for a holistic model of the datacenter, PRONTO was proposed to achieve both. PRONTO is a novel federated, asynchronous, and memory-limited algorithm that exploits federated learning (FL) techniques to distribute knowledge aggregation and processing across all the compute nodes in the datacenter.

With the rise of containerized workloads in datacenters, Kubernetes has firmly as the leading platform for container orchestration. It has experienced widespread adoption across diverse industries and organizations of all scales [**?**] and serves as the infrastructure backbone for cloud computing services like Google Cloud [**?**] and Azure [**?**]. Its versatility, from supporting machine learning (ML) deployments [?], edge computing [?] end serverless functions [? ?], solidifies Kubernetes' position as a cornerstone of the cloud-native industry. For these large-scale and dynamic environments, efficient scheduling is paramount, as wasted resources can directly impact an organization's bottom line Misconfiguration in resource requests and limits is a primary cause of under-utilised resources or idle nodes, leading to higher billing [? ?].

Kubernetes' default scheduler [?] (`kube-scheduler`) performs bin-packing based on the resource requests included in Pod descriptions and heuristic-based capacities calculated from the Nodes' advertised hardware. Therefore, `kube-scheduler`'s performance relies on accurate estimates of Pod resource usage. While telemetry-based systems have been proposed to improve scheduling decisions, they often use expensive deep-ML techniques [? ?] or focuses on a per-Node level [?]. These telemetry-based systems typically augment existing resource requests or re-schedule already running workloads to less congested Nodes.

PRONTO [?] leveraging of telemetry across a all nodes offers a promising approach for a purely telemetry-driven Kubernetes scheduler. However, its direct application within a Kubernetes scheduling environment faces significant challenges. PRONTO's original design assumes zero communication latency and yields only a binary "Reject-Job" signal, which lacks the granularity for Node scoring and fails to account for significant Pod startup latencies observed in real-world Kubernetes clusters [?]. Furthermore, empirical investigations into Linux Pressure Stall Information (PSI) metrics, a potential replacement for the VMware vSphere CPU-Ready metric, revealed frequent transient spikes attributed to the container runtime. This would make it difficult to efficiently distinguish genuine resource contention from noise when collecting a small set of metrics. These limitations necessitate a more sophisticated signal that is both **comparable** for effective Node ranking and **reservable** to track the pending impact of in-flight Pods.

1.1 Dissertation Aims and Contributions

This dissertation aims to apply the theory behind PRONTO to scheduling in Kubernetes by proposing, implementing, and evaluating CARICO (Italian for "load"), a novel, purely telemetry-driven scheduling algorithm that maintains the federated, asynchronous, and memory-limited properties of PRONTO explicitly accounting for communication latency.

The key contributions of this project are:

- **Feasibility Analysis of Pronto for Kubernetes:** An investigation into the applicability of PRONTO's principles to Kubernetes scheduling, highlighting its flawed communication assumptions and presenting empirical evidence of the challenges of using raw telemetry metrics.

- **Proposal of CARICO:** A novel federated, asynchronous, and memory-limited scoring algorithm that explicitly accounts for communication latency using a reservation mechanism.
  - Novel application of Federated Singular Value Decomposition (FSVD) to build a local model of recent resource usage, providing new interpretations and a modified Subspace-Merge function.
  - Presentation of a new **comparable** Capacity signal to score a Node's predicted ability to accept new workloads, considering recent resource utilisation across the cluster.
  - Application of signal processing techniques to transform the generated capacity signal into a **reservable** metric, enabling schedulers to account for communication latencies.
- **Prototype Implementation and Evaluation of CARICO:**
  - Extensive investigation into different telemetry metrics and signal processing techniques within the setting of a Kubernetes cluster to generate an accurate and precise capacity signal.
  - Analysis of the correctness of the generated signal and exploration of approaches for estimating reservation quantities.
  - Comprehensive evaluation of CARICO's overall performance under various workloads on a real-world Kubernetes cluster, comparing it with `kube-scheduler` across multiple scheduling objectives.

1.2 Dissertation Overview

The remainder of this dissertation is structured as follows:

- **Chapter 2 (Background):** Provides foundational knowledge, detailing Kubernetes architecture and its scheduling mechanism. It introduces PRONTO, explaining its core mathematics and highlighting its accuracy in predicting performance degradation, while also critically examining its weaknesses that necessitate a more sophisticated signal. Finally, it reviews existing Kubernetes schedulers to contextualize this work within the current landscape.
- **Chapter 3 (Design):** Proposes the CARICO algorithm, providing detailed proofs and reasoning to explain the generation and properties of its capacity signal.
- **Chapter 4 (Implementation):** Outlines the structure of the CARICO system within Kubernetes, exploring the selection of numerous metrics and signal processing techniques employed to generate an accurate comparable and reservable signal.

- **Chapter 5 (Evaluation):** Presents a comprehensive evaluation of CARICO's performance under different workloads, collecting numerous performance metrics and comparing it against the standard `kube-scheduler`.
- **Chapter 6 (Conclusion and Future Work):** Summarizes the key findings of the dissertation and proposes potential directions for future research.

Chapter 2

Background

This chapter lays the groundwork for understanding the problem space addressed by this dissertation. First datacenters are introduced, highlighting the difficulty of scheduling in this setting and outlining where PRONTO fits in this landscape. The chapter then delves into PRONTO's core concepts, providing the motivation for applying PRONTO to Kubernetes and the reasons that necessitate a more complex signal. Following this, an explanation of Kubernetes' architecture and the scheduling process is give, with a review of related Kubernetes schedulers to help position this dissertation's work within the current state of the art.

2.1 Datacenter Scheduling

Modern datacenters are physical facilities that serve as the backbone for running diverse user applications and data-parallel computations. The increasing demand for computing resources like compute, memory, storage and network has culminated in hyperscale datacenters that house hundreds of thousands of machines [**?** **?**]. Datacenter scheduling is the fundamental task of allocating the available resources to workloads such that their performance objectives are satisfied and the overall datacenter utilisation is kept high. Small deviations from the desired objectives can have substantial detrimental effects with millions of dallars in revenue potentially lost [**?**]. This problem continues to grow in difficulty, with the appearance of ever-increasing input workload rate and ever-changing workload characteristics onto heterogeneous resources [**?** **?**]. Moreover, they are expected to schedule for short user response time and high resource utilisation while also delivering high scheduling throughput [**?** **?** **?** **?** **?**].

Multiple scheduler architectures have been proposed to tackle different settings:

- **Centralised Schedulers:** Centralised schedulers like Borg [], Firmament [], Kubernetes [], and Quincy [] consist of a single monolithic scheduler that executes the entire scheduling logic to place tasks on available resources []. This design allows for more sophisticated policies to be applied. To overcome the issue of a single point of failure (SOPF), centralised scheduelers are often configured in a highly available topology [? ? ?], where in the case of a failure, a leader election round is triggered.

Two-tiered scedulers [?]are a variant of central schedulers where one entity (master) manages resources of the datacenter and their allocation, and the other carries out task placement. This delegates resource allocation to schedulers with domain-specific knowledge, while still being able to handle node failures through the master.

- **Decentralised Schedulers:** As datacenters continue to grow, centralised schedulers become bottlenecks, facing increased overheads collecting resource information, while simultaneously handling high frequency job arrivals. To overcome this limitation, decentralised schedulers like Apollo [] and Omega [] employ several independent job schedulers which update a shared centralised state after each placement decision. Due to its concurrent nature, commits can result in conflicts [?] which must be handled carefully.

- **Distributed Schedulers:** Like decentralised schedulers, distributed schedulers have multiple indpendent scheduler instances. However, they require no coordination or shared state. Schedulers like Peacock [?] and Sparrow [?] aim to make scheduling decisions as fast as possible, without the knowledge of the entire datacenter. Although these schedulers are scalable and available owning to their distributed nature, having a limited view of the datacenter can result in sub-optimal placement choices.

- **Hybrid Schedulers:** Hybrid schedulers like Dice [?], Eagle [?], Hydra [?] and Mercury [?] combine the properties of centralised and distributed schedulers to achieve good placement at scale and for high workload arrival rates. They typically employ separate schedulers, each given different scheduling policies and tasked with placing a different kind of job into specially reserved sub-clusters [? ? ? ?]. Due to the requirement for workload-specific parameters, sub-clusters may become under- or over-utilised when workload characteristics change [?].

Much resarch has also gone into developing different algorithms for scheduling.
Multi-dimensional resource allocation transforms scheduling into a bin-packing problem. Schedulers like Quincy [?] and Jockey [?] allocate a uniform amount of resources to all tasks regardless of their needs, potentially hurting application performance. Other schedulers [? ? ? ?] support user-specified job resource requirements, but risk users overestimating [?] or not knowing howing much to request [?]. Several methods have been proposed to improve the accuracy of job resource requests. These range from profiling the job before actual execution [?] , decreasing or redistributing the initial resource allocation after an observation period [? ?], and analysing historical traces of jobs to build a performance model of the expected performance based on the job resource

allocation [? ? ?].

Constraint handling schedulers [] allows tasks to have placement constraints, such as, requiring specialist hardware like a Tensor Processing Units [?](TPUs). Kubernetes [] supports both hard (mandatory) and soft (not mandatory but preffered) constraints, as well as affinity/anti-affinity constraints.

Finally, schedulers like [? ? ? ? ? ? ?] rely on **predictions of node's future resource availablilty** to avoid saturation and efficently utilise resources across data center nodes. Approaches range from probing available nodes on an on-demand basis [? ? ?] to collectively generated telemetry across the datacenter and using offline machine learning prediction to tackle oversubscription [? ?]. While it has been shown that schedulers generate improved holistic models for efficient provision when given access to performance data from all data center nodes [? ? ? ? ? ?], they typically operate on a near offline fashion with a high network cost, making them slow to react in real-time to performance problems. PRONTO [?] pioneers a federated approach to analysing real-time telemetry from virtualised data center nodes for online task scheduling.

2.2 Kubernetes

2.2.1 Kubernetes Overview

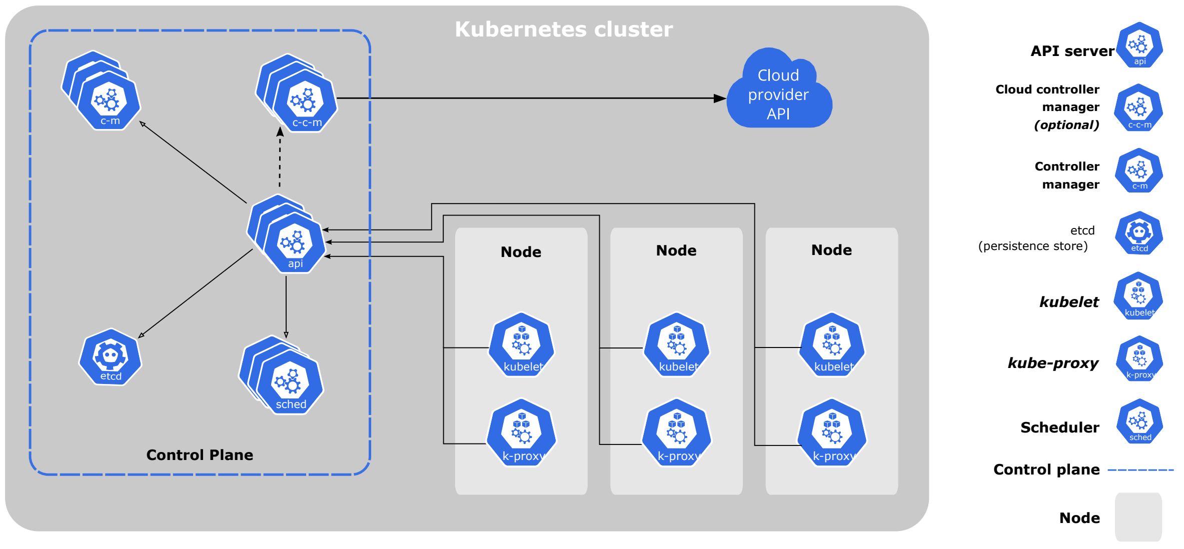


Figure 2.1: The components of a Kubernetes cluster [?]

A Kubernetes cluster consists of a control plane and one or more worker Nodes. Components in the control plane manage the overall state of the cluster, while each Node in the cluster contains a `kubelet` which manages Pods and ensures they and their containers are running via a container runtime. Kubernetes objects are persistent entities in the Kubernetes system. They act as "records of intent" and describe the cluster's desired state: once created, the Kubernetes system will constantly work to ensure that the objects exists. The `kube-apiserver` exposes the Kubernetes HTTP API, which is used to create, modify or delete these Kubernetes objects.

2.2.2 Scheduling in Kubernetes

In Kubernetes, Pods are the smallest deployable units of computing that you can create and manage. It represents a single instance of a running process in your cluster and typically contains one or more containers that are tightly coupled and share resources. While Pods can be individually created, the Kubernetes API also provides workload objects to manage multiple pods: objects that represent a higher abstraction level than a Pod, and the Kubernetes control plane uses the workload's specification to manage Pod objects on your behalf. This dissertation will use the following:

- **Deployment:** manages a set of Pods to run an application workload and typically doesn't maintain state [].
- **DaemonSet:** ensures that all (or some) Nodes run a copy of a Pod to provide Node-local facilities [].
- **Job:** creates one or more Pods and will continue to retry execution of the Pods until a specified number of them successfully terminate [].

Almost every Kubernetes object includes two fields: `spec` and `status`. `spec` is used on creation as a description of the Objects desired state. A Pod's `spec` can define affinities and QoS classes to influence scheduling decisions. Containers also contain a `spec` field which specifies `request` and `limits`. The `request` field behaves as a set of minimum requirements and is used when scheduling Pods. In contrast, the `limits` field is used by kernel of the Node to throttle a container's resource usage. In Kuberentes, the resource units for `resource` and `limits` are the following:

- **CPU:** The metric used is *cpu* units, where 1 cpu unit is equivalent to a 1 physical/virtual core. Fractional requests are permitted with the notation xm; 100m can be read as "one hundred millicpu".
- **Memory:** This is measured in bytes. Kubernetes also allows the use of quantity suffixes, such as, k, M, G, Ki, Mi, Gi.

`status` describes the current state of the object, supplied and updated by the Kubernetes system. These fields are core to scheduling in Kubernetes.

When a Pod is created, it initially exists in a "Pending" state: it has been declared but hasn't yet been allocated to a Node. Kubernetes schedulers watch for newly created but unassigned Pods, and based on a set of rules or algorithms, select the most suitable Node for that Pod. Once a Node is chosen, the scheduler "binds" the Pod to the Node, updating the Pod's definition in the Kubernetes API server by setting its `spec.nodeName` field to the name of the Node. Once this occurs, the Pod transitions from “Pending” to “Running”.

2.3 Pronto

This section first explains theory behind Pronto, which will also be eventually used by Carico. We outline the motivation for this dissertation, highlighting Pronto’s strengths and potential benefits to Kubernetes scheduling. Finally, we investigate its assumptions and empirical evidence to reveal Pronto’s limitations and outline requirements of a new algorithm that produces and uses a **comparable** and **reservable** signal to inform scheduling decisions.

2.3.1 Singular Value Decomposition

The SVD of a real matrix $\mathbf{A}$ with m rows and n columns where $m \geq n$ is defines as $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T$. Here, $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices of shape $m \times m$ and $n \times n$ containing the left and right singular vectors, respectively. Σ is a rectangular matrix of shape $m \times n$ with singular values σ_i along its diagonal [?].

$$\mathbf{A} = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix} = \begin{bmatrix} | & & | \\ u_1 & \dots & u_m \\ | & & | \end{bmatrix} \begin{bmatrix} \sigma_1 & & & | & & | \\ & \ddots & & 0 & \dots & 0 \\ & & \sigma_m & | & & | \end{bmatrix} \begin{bmatrix} — & v_1^T & — \\ & \vdots & \\ — & v_m^T & — \\ — & v_{m+1}^T & — \\ & \vdots & \\ — & v_n^T & — \end{bmatrix} \tag{2.1}$$

SVD can also be written compactly by discarding the elements which do not contribute to $\mathbf{A}$.

$$\mathbf{A} = \begin{bmatrix} | & & | \\ u_1 & \dots & u_m \\ | & & | \end{bmatrix} \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_m \end{bmatrix} \begin{bmatrix} — & v_1^T & — \\ & \vdots & \\ — & v_m^T & — \end{bmatrix} \tag{2.2}$$

There always exists the SVD for a real matrix, but the decomposition is not unique: if $\mathbf{A} = \mathbf{U}_1\Sigma\mathbf{V}_1^T = \mathbf{U}_2\Sigma\mathbf{V}_2^T$ then $\Sigma_1 = \Sigma_2$ but $\mathbf{U}_1 = \mathbf{U}_2\mathbf{B}_a$ and $\mathbf{V}_1 = \mathbf{V}_2\mathbf{B}_b$ for some block diagonal unitary matrices $\mathbf{B}_a, \mathbf{B}_b$ [? ?]. Each column in $\mathbf{U}$ and $\mathbf{V}$ is an eigenvector of $\mathbf{A}\mathbf{A}^T$ and $\mathbf{A}^T\mathbf{A}$.

2.3.2 Principal Component Analysis

Principal Component Analysis [] is a staple of linear dimensionality-reduction techniques. The standard PCA procedure takes as input a matrix $\mathbf{B}$ representing n columns of data with m dimensions and mean-centers it: $\mathbf{A}_{ij} = (\mathbf{B}_{ij} - \mu_i)$ where μ_i is the mean of the row i. The output of PCA is a set of vectors that explain most of the variance within $\mathbf{B}$. Given the covariance of the mean-centered matrix $\mathbf{A}$ is defined as $\mathbf{A}\mathbf{A}^T$, the Principal Components (PCs) maximise the following equation:

$$\mathrm{Var}_i = \max_{\substack{x_i \in \mathbb{R}^m \setminus \{\mathbf{0}\} \\ \|x_i\|=1 \\ x_i \perp x_1 \dots x_{i-1}}} x_i^T \mathbf{A}\mathbf{A}^T x_i \tag{2.3}$$

As $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T$ from SVD, $\mathbf{A}\mathbf{A}^T = \mathbf{U}\Sigma\mathbf{V}^T\mathbf{V}\Sigma\mathbf{U}^T = \mathbf{U}\Sigma^2\mathbf{U}^T$. Therefore, it can be shown that the PCs $x_i = u_i$ from $\mathbf{U}$. The pair $\mathbf{U}, \Sigma$ will also be referred to as a subspace as they provide sufficient information to describe the original $\mathbf{B}$ matrix.

2.3.3 Subspace-Merge

Subspace-Merge is used to merge two subspaces together. Given two subspaces $(\mathbf{U}_1, \Sigma_1)$ and $(\mathbf{U}_2, \Sigma_2)$ from $\mathbf{Y}_1$ and $\mathbf{Y}_2$ respectively, the subspace of $\mathbf{Y} = [\mathbf{Y}_1, \mathbf{Y}_2]$ is:

$$\mathbf{U}\Sigma = \mathrm{SVD}([\mathbf{U}_1\Sigma_1, \mathbf{U}_2\Sigma_2]) \tag{2.4}$$

$[\mathbf{A}, \mathbf{B}]$ signifies the concatenation of two matrices with the same number of rows. This procedure doesn't use the right singular vectors, reducing the required computations and memory.

2.3.4 Incremental-SVD

Incremental-SVD allows Pronto to become a streaming algorithm with limited memory. It takes a stream of chunks $\mathbf{Y}_i$, such that $[\mathbf{Y}_1, \dots, \mathbf{Y}_l] = \mathbf{Y}$, and with each recieved chunk it performs Subspace-Merge to produce $\mathbf{U}_l, \Sigma_l$ where $\mathbf{Y} = \mathbf{U}_l\Sigma_l\mathbf{V}_l^T$

Algorithm 1 Incremental-SVD

```
Data: $\mathbf{Y} = [\mathbf{Y}_1, \dots, \mathbf{Y}_l]$
Result: $\mathbf{U}_l, \Sigma_l$ such that $\mathbf{Y} = \mathbf{U}\Sigma\mathbf{V}^T$
  $\mathbf{U}_1, \Sigma_1, \mathbf{V}_1^T = \mathrm{SVD}(\mathbf{Y}_1)$
  for $i = 2$ to $l$ do
      $\mathbf{U}_i, \Sigma_i, \mathbf{V}_i^T = \mathrm{SVD}([\mathbf{U}_{i-1}\Sigma_{i-1}, \mathbf{Y}_i])$
  end for
```

If the shape of the batches of data is $m \times b$, the space complexity of Incremental-SVD is $\mathcal{O}(m^2 + mb)$ as only the latest version of $\mathbf{U}_i, \Sigma_i$ and $\mathbf{Y}_i$ are needed for each iteration.

2.3.5 FPCA and FSVD

FPCA combines the relationship between standard SVD and PCA with the Subspace-Merge operation, to calculate the PCs of the data $[\mathbf{Y}_1, \ldots, \mathbf{Y}_m]$ from m nodes. Every node i performs perform SVD on their local data, to produce the subspace $\mathbf{U}_i, \Sigma_i$. These subspaces can be merged using at most $m-1$ Subspace-Merges to obtain the global subspace $\mathbf{U}'\Sigma'$, corresponding to the PCs of the aggregated data from all the nodes. Pronto turns this procedure into a streaming algorithm, by running Subspace-Merge on the subspaces produced by incremental-SVD. Finally, Pronto also introduces a forgetting factor γ in front of $\mathbf{U}_i\Sigma_i$ in Incremental-SVD that allows it to "forget" old data by gradually reducing the influence of previous subspaces. Like with standard PCA, FPCA can be considered as FSVD on mean-centered data.

2.3.6 Low-Rank Approximations

It can be shown that using the first r eignenvectors in the above algorithms approximate the result of using all m eignenvectors, i.e. if

$$\mathbf{Y} = \begin{bmatrix} | & & | \\ u_1 & \ldots & u_m \\ | & & | \end{bmatrix} \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_m \end{bmatrix} \begin{bmatrix} — & v_1^T & — \\ & \vdots & \\ — & v_m^T & — \end{bmatrix} \tag{2.5}$$

We can use $\mathbf{U}^r = \begin{bmatrix} | & & | \\ u_1 & \ldots & u_r \\ | & & | \end{bmatrix}$ and $\mathbf{\Sigma}^r = \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_r \end{bmatrix}$ where $r \leq m$ in Incremental-SVD and Subspace-Merge. This lets Pronto reduce the number of computations it performs and its memory usage.

2.3.7 Pronto System Overview

There are two types of nodes in Pronto (Figure **??**): compute node (C) and aggregator node (A). Compute nodes collect and center telemetry (i.e. CPU and Memory) and perform Incremental-SVD to obtain the low rank approximations of the local subspace $\mathbf{U}, \Sigma$. The aggregator nodes perform Subspace-Merge on incoming subspaces, with subspace produced by the root aggregator node being propagated back to the compute nodes.

2.3.8 Pronto Reject-Job Signal

Pronto performs peak detection to estimate performance degradation (Figure **??**) Each compute node projects their data onto the latest version of $\mathbf{U}$, and identifies all the spikes. If the weighted sum of these spikes, using the corresponding singular values in Σ, exceeds a threshold, a "Reject-Job" signal is raised to indicate that the node is potentially experiencing performance degradation and a task should not be scheduled on that node.

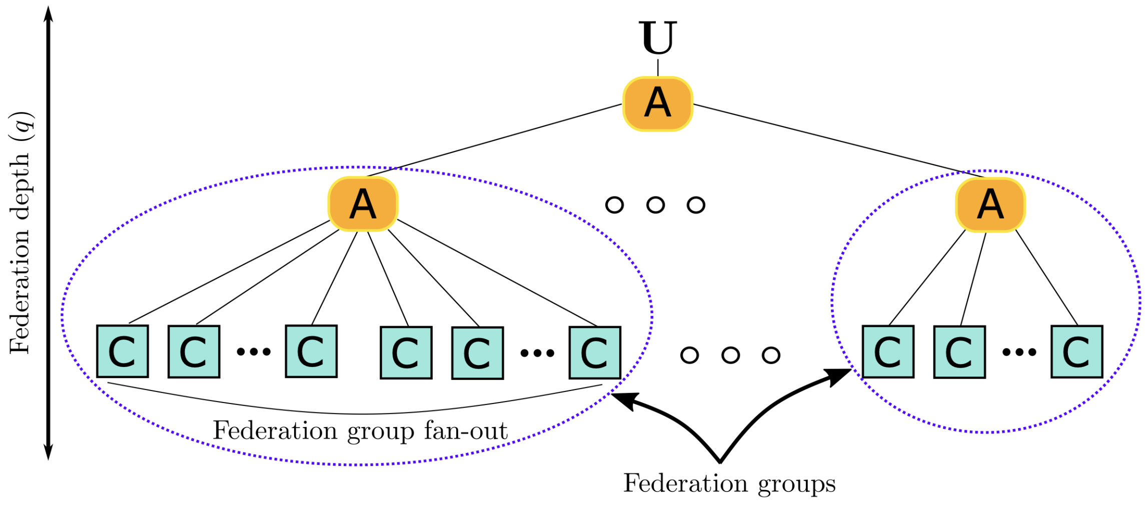


Figure 2.2: How local models are aggregated in PRONTO. Dedicated aggregator nodes propagate the updated subspaces until the root is reached [?].

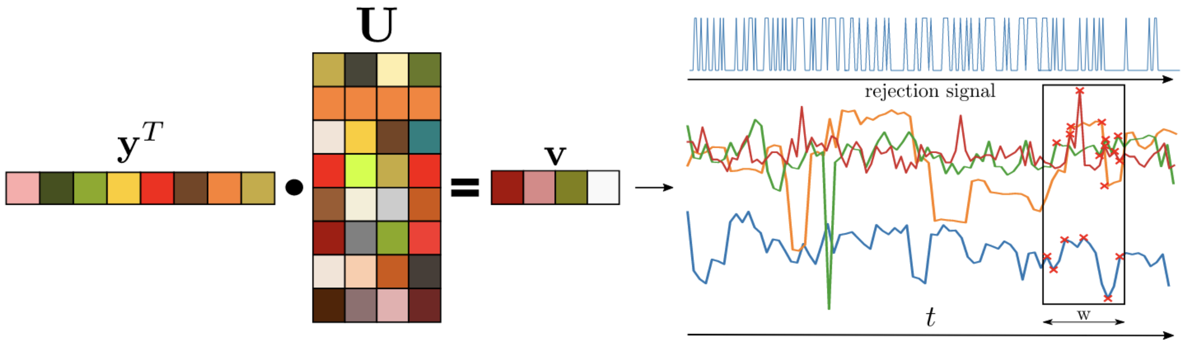


Figure 2.3: Projection of incoming $y \in \mathbb{R}^d$ onto embedding $U \in \mathbb{R}^{d \times r}$ producing R projections in $v \in \mathbb{R}^{1 \times r}$. Projections are tracked over time for detecting spikes which form the basis of the rejection signal. The sliding window for spike detection for each projection is of size w also shown in the figure.

2.3.9 Strengths

PRONTO's core strength is its ability to leverage global telemetry data to predict performance degradation with high accuracy. The orginal PRONTO paper [?], which compared its effectiveness against non-distributed dimensionality-reduction methods, demonstrated superior performance in predicting CPU-Ready spikes in real-world datacenter traces. This improved accuracy over the non-distributed strategies suggests that a federated approach allows for a more comprehensive understanding of system-wide resource contention, providing more accurate contention predictions than with only individual node data. These compelling results indicate a strong potential benefit from applying PRONTO underlying principles to Kubernetes where efficient resource management is paramount.

2.3.10 Weaknesses

While PRONTO offers a promising approach to performance prediction, its direct application within a Kubernetes environment faces significant challenges due to fundamental communication assumptions and practical limitations of peak-prediction.

Assumptions

Implementing PRONTO in a complex system like Kubernetes raises fundamental challenges:

1. **Zero-Latency Assumption:** A critical assumption in the PRONTO paper is the absence of communication latency, and implicitly binding latency. This implies that scheduled workloads are immediately reflected in a node's telemetry, and therefore, its Reject-Job signal. In such a scenario, a central scheduler could instantaneously stop assigning workloads once a Node signal potential degredation. However, this assumption does not hold in real-world Kubernetes clusters. The latency between a pod being bound to a Node and that Pod actually starting to run and consume resources has been shown to reach as high as 4 seconds [**?**]. Directly applying PRONTO's binary signal in this high-latency environment could lead to a "runaway train" schenario": Nodes might advertise willingness to accept new Pods while a large number of "inflight" Pods are still pending startup and will overload the node once active. This highlights the need for a reservation mechanism for a node's signal.

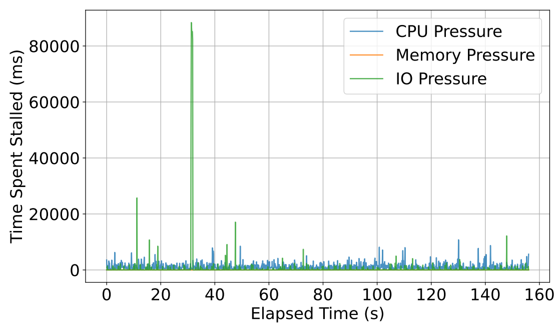
2. **Lack of Explicit Allocation Algorithm:** PRONTO allows individual compute nodes to reject or accept incoming jobs. However, it does not explicitly provide a system to optimally decide which compute nodes are considered for a task. While a simple approach would be to allow individual Nodes to request Pending Pods, it will struggle to scale. Having hundreds of Nodes sending requests to `kube-apiserver` would greatly degrade its performance or even cause it to crash. Instead, I decided to take a centralised approach, where Nodes periodically send their signal to a central scheduler to influence where Pods are allocated.

3. **Limited Scoring Capability:** Unlike typical Kubernetes schedulers that employ a scoring-function to rank Nodes and select the "optimal" fit [**?**], a binary signal offers no mechanism to differentiate between suitable Nodes. This could lead to suboptimal allocation decisions, potentially reducing overall cluster throughput and efficiency. Thus, Nodes would need to produce a comparable signal.

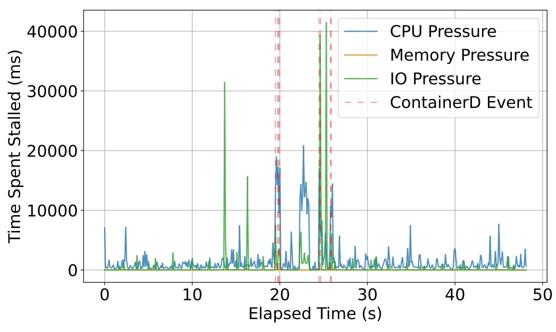
Peak-Prediction

PRONTO uses peak-prediction on collected telemetry to predict future performance degradation. For PRONTO to generate an accurate Reject-Job signal in Kubernetes, the chosen contention metrics should exhibit clear, distinguishable spikes during genuine high-resource contention. While increasing the number of collected metrics can help reduce the impact of erroneous spikes, collecting more telemetry and operating on larger matrices will incur additional overheads and reduce the available resources for Pods.

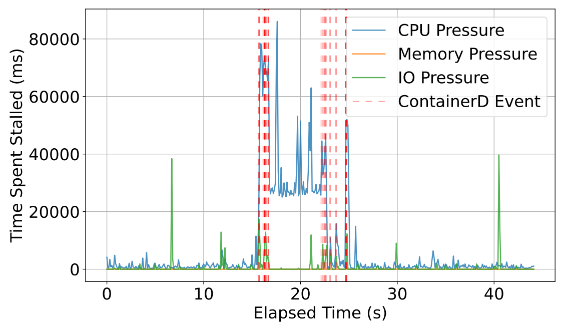
Pronto used metrics, like CPU-Ready, specific to virtual machines (VMs), limiting its application to Kubernetes clusters with only VMs. Linux-based systems offer alternative Pressure Stall Information (PSI) metrics, accessible via `/proc/pressure/<cpu|memory|io>`. These pseudo-files track the time tasks are stalled waiting for resources. To investigate the feasibility of using PSI for peak prediction within a Kubernetes Node, I polled the polled the `/proc/pressure` files under different workloads.



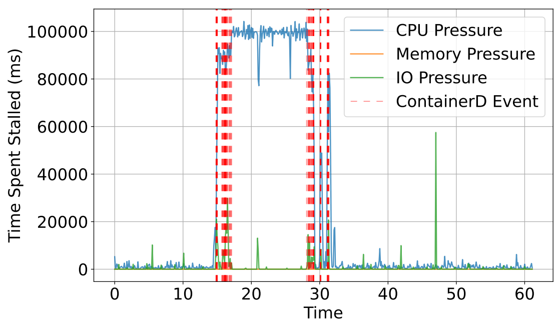
(a) PSI when no Pods are running



(b) PSI when running a single Pod



(c) PSI when running 5 Pods

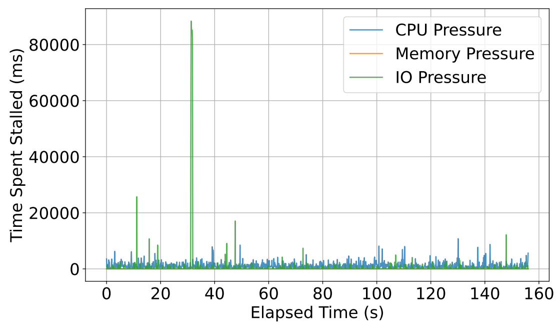


(d) PSI when running 10 Pods

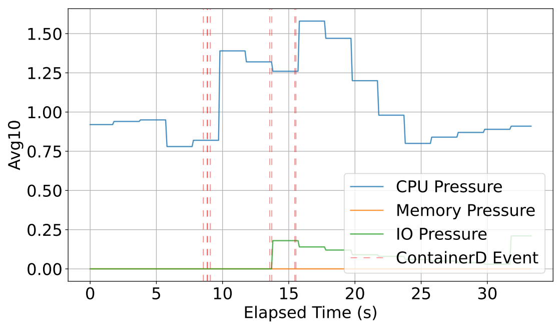
Figure 2.4: Measurements of `total` from `/proc/pressure/` when running different sized Jobs with Pods executing `bpi(2000)`. Spikes are observed across all workloads.

Figure **??** shows how lightweight workloads can still cause the PSI metrics to experience significant spikes. These transient spikes can be attributed to the container runtime (e.g. Containerd) consuming resources during the creation or deleteion of containers. This "noise" would be difficult to distinguish from genuine indicators of resource contention , compromising the accuracy of Pronto's peak-detection mechanism.

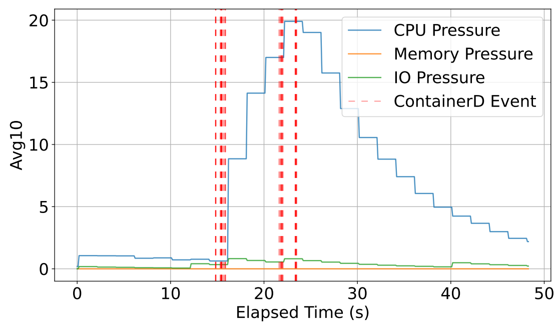
The PSI metrics also expose an average over a 10-second window. Figure **??** illustrates that averaging the PSI metrics can indeed reduce the impact of these container runtime-induced spikes. However, this smoothing comes at the cost of responsiveness. In the experiments running 10 Pods, the averaged metrics failed to converge on the true value observed in Figure **??** before the Pods had finished running. This delay could still result



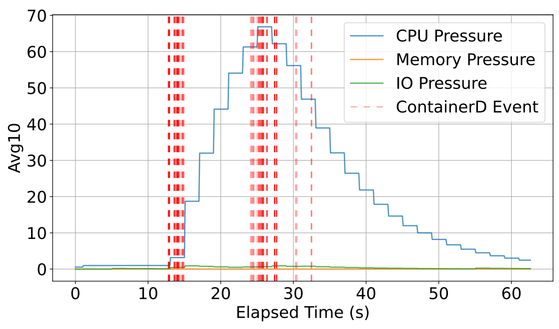
(a) PSI when no Pods are running



(b) PSI when running a single Pod



(c) PSI when running 5 Pods



(d) PSI when running 10 Pods

Figure 2.5: Measurements of `avg10` from `/proc/pressure/` when running different sized Jobs with Pods executing `bpi(2000)`. The average fails to respond quickly to large contention.

in the same “runaway train” situation described earlier.

Based on this early empirical investigation, I conclude that peak-prediction on sub-second polling of raw PSI metrics is not feasible within the noisy Kubernetes environment.

2.4 Next Steps

These weaknesses highlight the requirements for a more sophisticated federated scheduler:

1. **Comparable and Reservable Signal:** Node’s will generate a signal that: 1) provides enough information to score and rank Nodes effectively, and 2) allows the scheduler to track the pending impact of previous scheduling decisions until Pods have begin running.
2. **Centralised Architecture:** A central scheduler will use Nodes’ scores to determine Pod allocation perform the “bind” operation.
3. **Avoids Peak Prediction:** Telemetry generated by Node is too noisy to perform low-dimensionality peak-prediction.

2.5 Summary

This chapter provided the foundational knowledge for the dissertation, introducing an overview of the Kubernetes architecture and its scheduling mechnaism. It then introduced PRONTO, a novel approach to scheduling, explaining the core mathematics behind its FPCA, and highlighting its accuracy in predicting performance degradation through global telemetry. Finally, we identified significant limitations of PRONTO, and produced the requirements for a more sophisticated federated algorithm for Kubernetes scheduling.

Chapter 3

Carico

This chapter details CARICO, a federated, asynchronous, memory-limited algorithm, building upon the principles of PRONTO. CARICO differentiates itself from PRONTO for the following reasons:

1. Rather than performing standard FPCA, CARICO applies FSVD on observed [0,1]-normalised resource usage. Subsequently, CARICO uses new interpretations for the results of SVD and modifies the Subspace-Merge to preserve the interpretations.
2. CARICO presents a novel continuous and **comparable** capacity signal function that combines the new model interpretations with a Node's current resource usage to calculate its estimated workload capacity.
3. CARICO makes two assumptions to translate the calculated capacity signal into a signal that is both **reservable** and uses the number of Pods as its unit of measure.

As this algorithm produces a signal that measures "capacity", I will refer to it as CARICO - "load" in Italian.

3.1 Capacity Signal

In Section **??**, we identified critical limitations of PRONTO's binary "Reject-Job" signal within the Kubernetes ecosystem: its lack of comparable Node scoring and inability to handle Pod startup latencies. While one could conceivably compare the number of detected spikes, it's difficult to quantitatively assign or reserve "peak detections" for incoming Pods. Furthermore, as shown in Figure **??**, contention metrics often do not change proportionally to the number of tasks assigned, making them difficult to reserve accurately.

These limitations demand a signal reflecting varying levels of contention and offering a predictable, scalar relationship with task assignments. `kube-schedulers`'s reservation of a Node's resource highlights the potential use of a capacity metric. FPCA's interpretations focus on the variability within telemetry data rather than its absolute value, and therefore, is not suitable for calculating measures of capacity. The subsequent challenge was to find a means of transforming PRONTO's mathematical framework to produce values that could be interpreted as the direction and magnitude of recent workload's resource usage. Only once we had this interpretation, could we then develop a continuous, comparable, and reservable signal that could effectively and accurately guide scheduling decisions.

3.2 Local Model

CARICO assumes a batch of telemetric data $\mathbf{A}$ is an $m \times n$ matrix, where $\mathbf{A}$ contains n samples of m-dimensional columns of data. Each dimension in the vector represents a different resource, where the value 0 indicates that full capacity is available for that resource and 1 indicates that the resource is being fully used ([0,1]-normalised). As we are no longer mean-centering the dataset before applying SVD, **PCA's interpretations no longer apply to the resulting $\mathbf{U}$ and Σ matrices**.

Instead, CARICO is built on top of a different set of interpretations. Using the Lemma proved in Appendix **??**, we can interpret the first left singular vector u_1 in $\mathbf{U}$ as a pseudo-weighted average direction of the datapoints in the matrix: "larger" or more aligned columns contribute more significantly to the sum defining u_1, and thus to its final direction. Thus u_1 behaves as a primary indicator of the direction of the current workload's resource usage.

While knowing the proportion of resource usage is useful, it is also important to be able to differentiate between lightweight workloads and more intense workloads. For this, CARICO uses the first singular value σ_1 in Σ. Given $\sigma_1(\mathbf{A})$ and u_1 correspond to the first singular value and first left singular vector of a batch of telemetry $\mathbf{A}$:

$$\sigma_1(\mathbf{A})^2 = u_1^T \mathbf{A}\mathbf{A}^T u_1 \tag{3.1}$$

$$= (u_1^T \mathbf{A})^2 \tag{3.2}$$

Given a_j is the j-th column of $\mathbf{A}$, $\sigma_1(\mathbf{A})^2$ can be interpreted as the sum of squared scalar projections of columns in $\mathbf{A}$: $\sigma_1(\mathbf{A})^2 = \sum_{j=1}^{n}(u^T a_j)^2$.

Furthermore, the first singular value can be shown to scale with resource usage. Given two batches of telemetry $\mathbf{A}$ and $\mathbf{B}$ where batch $\mathbf{B}$ experienced more resource usage ($0 \leq a_{ij} \leq b_{ij} \leq 1$ for all i, j), it is shown in Appendix **??** that the first singular value of $\mathbf{B}$ will be greater than or equal to that of $\mathbf{A}$ ($\sigma_1(\mathbf{A}) \leq \sigma_1(\mathbf{B})$). Subsequently, the first singular value makes a good indicator of measured resource usage.

3.3 Subspace Merging

While we have shown that the results from performing SVD on [0,1]-normalised telemetry data can have useful interpretations, the resulting σ_1 can grow with each Incremental-SVD (Appendix **??**):
Given two non-negative matrices $\mathbf{A} \in \mathbb{R}^{m\times n_1}$ and $\mathbf{B} \in \mathbb{R}^{m\times n_2}$, the first singular value of the concatenated matrix $\mathbf{C} = [\mathbf{A}, \mathbf{B}]$ is greater than or equal to the maximum of the first singular values of $\mathbf{A}$ and $\mathbf{B}$:

$$\sigma_1([\mathbf{A}, \mathbf{B}]) \geq \max(\sigma_1(\mathbf{A}), \sigma_1(\mathbf{B}))$$

This property is important, as we established earlier that CARICO uses σ_1 as an indicator of the magnitude of resource usage. If σ_1 can increase while the measured telemetry reports a stable magnitude of resource usage, its interpretation no longer holds.

To solve this, CARICO scales the concatenated matrices using non-negative scalar weights $\gamma_{\mathbf{A}} = \sqrt{w_{\mathbf{A}}}$ and $\gamma_{\mathbf{B}} = \sqrt{w_{\mathbf{B}}}$ such that $w_{\mathbf{A}} + w_{\mathbf{B}} = 1$. This method still preserves the earlier interpretations:

- **First Left Singular Vector:**
  The resulting first singular vector u_1 of the concatenated matrix $\mathbf{C} = [\gamma_{\mathbf{A}}\mathbf{A}, \gamma_{\mathbf{B}}\mathbf{B}]$ still behaves as a pseudo-weighted average, but with the contributions of the input telemetry further weighted according to the scalar weights. Futhermore, the weights can be used as a forget factor similar to γ in PRONTO.

- **First Singular Value:**
  Appendix **??** proves that the resulting first singular value $\sigma_1(\mathbf{C})$ and the first left singular vector u_1 of the concatenated matrix $\mathbf{C} = [\gamma_{\mathbf{A}}\mathbf{A}, \gamma_{\mathbf{B}}\mathbf{B}]$ have the following property:
  $$\min(P_{\mathbf{A}}(u_1), P_{\mathbf{B}}(u_1)) \leq \sigma_1(\mathbf{C})^2 \leq \max(P_{\mathbf{A}}(u_1), P_{\mathbf{B}}(u_1))$$
  where $P_{\mathbf{M}}(u)$ is the sum of squared scalar projections of the columns in $\mathbf{M}$ onto u. As $\sigma_1(\mathbf{C})^2$ is a convex combination of $P_{\mathbf{A}}(u_1)$ and $P_{\mathbf{B}}(u_1)$, it is bound and acts as a weighted average of two projected sums in the new workload direction.

3.4 Capacity Signal Function

Using the established interpretations:

- u_1: the pseudo-weighted average direction of resource usage of recent workloads
- σ_1: the magnitude of the resource usage of recent workloads

we can derive a new capacity signal that considers both the estimated workload resource usage and its current resource usage. Given the the current [0,1]-normalised resource usage vector for the node y, the first singular value and left singular vector σ_1, u_1 from the latest subspace (U, Σ), a Node's estimated "Capacity" signal k is given by:

$$y_{\text{predict}} = y + k * \sigma_1 u_1 \tag{3.3}$$

$$\max_k \forall i : y_{\text{predict}} < 1 \tag{3.4}$$

The resulting Capacity signal k represents how many "units" of the learned "average" workload $(\sigma_1 u_1)$ can be added to the current workload (y) before any single resource dimension hits its normalised capacity of 1.

3.4.1 Example Scenarios

To better understand how this signal works, I explore how the Capacity signal changes under different different workloads. These scenarios will also be used to verify the Capacity signal implementation in the prototype.

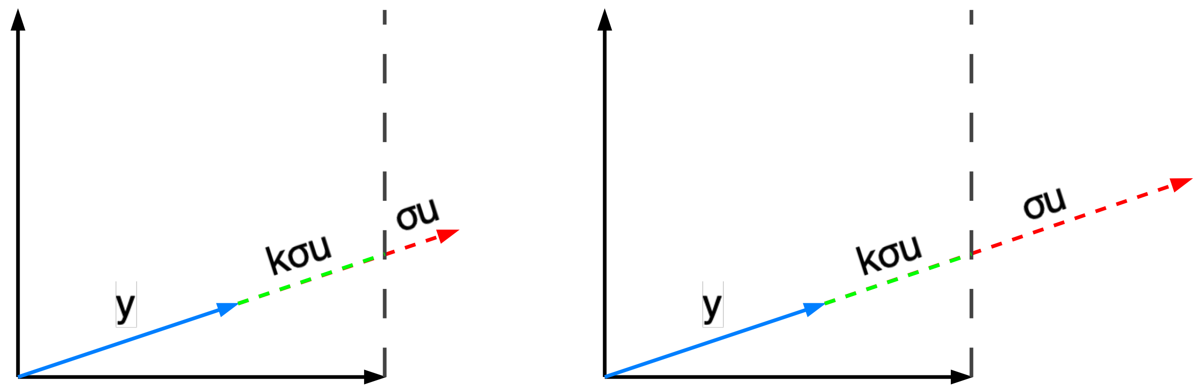


Figure 3.1: Visualisations of a Node's resource usage y and expected resource usage $\sigma_1 u_1$ when learning of conflicting resource utilisation.

Figure **??** presents the scenario where a Node updates its local model, learning that the experienced workload has increased in resource usage (σ_1 has increased). This means that a smaller constant k is needed before the combined vectors cross a resource boundary. The Node, thus, advertises a smaller capacity signal as it has less capacity for the expected resource usage.

Figure **??** presents the scenario where a Node updates its local model, learning that the learned workload is using more resources but in a different direction. This could be described as a complementary workload as the the new expected resource utilisation requires a larger constant k to cross a resource boundary. Therefore, the Node advertises a higher capacity signal as it has more capacity for the new expected resource utilisation.

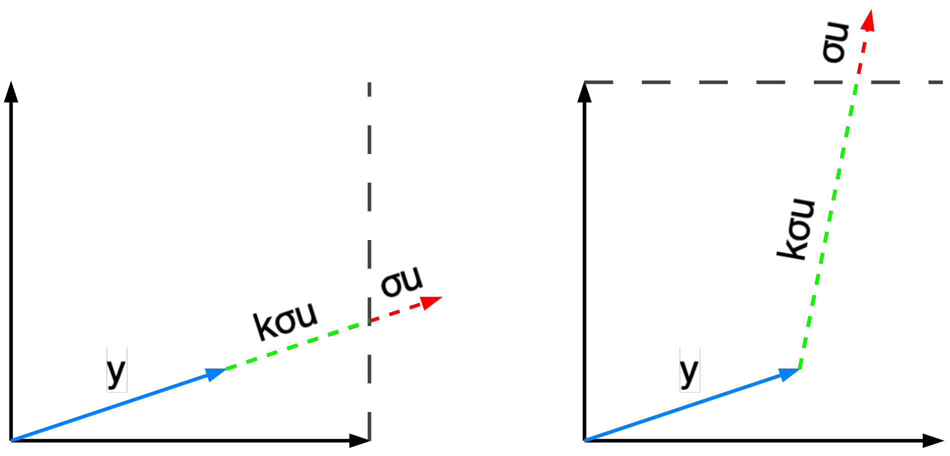


Figure 3.2: Visualisations of a Node's resource usage y and expected resource usage $\sigma_1 u_1$ when learning of complementary resource utilisation.

3.5 Reserve Cost and Capacity

Like Pronto, Carico's signal uses its current resource usage, and therefore, only reflects Pods that have been scheduled and are running on the Node. For Carico to account for Pod startup latency, the central scheduler must be able to predict the a Pod's effect on a Node's signal. Consequently, Carico assumes that Nodes know the number of currently running Pods and have the ability to estimate their:

1. **Baseline Capacity Signal:** the Capacity signal when no Pods are running
2. **Per-Pod-Cost:** the estimated drop in Capacity signal will drop when a single Pod starts running.

Section **??** implements and compares different prediction methods. With this information, a Node can calculate its Pod-Capacity (Capacity given in units of Pods) from:

$$\text{Pod-Capacity} = \frac{\text{Current Capacity Signal}}{\text{Per-Pod-Cost}} \tag{3.5}$$

$$\text{Pod-Capacity} = \frac{\text{Baseline Capacity Signal}}{\text{Per-Pod-Cost}} - \text{Current Pod Count} \tag{3.6}$$

This metric has two useful properties:

- **Dual-Mode:** Pod-Capacity can be calculated using two equations. This is especially useful in Kubernetes as container creation and deletion can cause significant resource spikes, even with filtering in Figure **??**, impacting immediate predictions. To combat this noise, Nodes can switch to predicting Pod-Capacity from previous Baseline and Per-Pod-Cost estimates. This reduces fluctuations in a Node's advertised Pod-Capacity and improves scheduling decisions
- **Unit of Measure:** As its unit of measure is in terms the number of Pods, it simplifies the central scheduler's logic, removing the need to keep track of each

Node's Per-Pod-Cost.

3.6 Central Scheduler

Each Node n will broadcast its Pod-Capacity$_n$ to a central scheduler. This scheduler also tracks each Node's reserved amount as # Pods Reserved$_n$. For each Pending Pod, the scheduler performs the following operations:

- **Filter:** Filters out all Nodes n with Pod-Capacity$_n$ − # Pods Reserved$_n$ < 1. Thus, the scheduler only considers Nodes with enough resources for another Pod. While reducing the threshold could pack more Pods on a Node, overallocation could result in OOM kills if there isn't enough memory available on a Node for all the Pods.
- **Score:** Score Nodes n by Pod-Capacity$_n$ − # Pods Reserved$_n$. This ensures we allocate to Nodes which can fit more Pods.
- **Reserve:** Once a Node n has been chosen, we increment # Pods Reserved$_n$ by 1. Once a scheduled Pod is no longer in the Pending state, the central scheduler decrements # Pods Reserved$_n$ by 1 for the Node n the Pod was assigned to.

3.7 Properties

Pronto is designed to be *federated, streaming* and *unsupervised.* Carico exhibits identical properties while also considering the existance of communication and Pod startup latency.

Federated: While Carico uses a central scheduler to score Nodes and perform the final Bind operation, it can still be considered federated because of its use of FSVD []: individual Nodes have a unified view of the global workload while maintaining their individual autonomy to set their own score.

Streaming: Like Pronto, Carico only requires a single pass over the incoming data in order to update its estimates. In addition, Incremental-SVD only requires memory linear to the number of features considered; given batches of dimension $d \times b$, the required memory is proportional to $\mathcal{O}(d)$. This memory footprint can be further reduced with low-rank approximation.

Unsupervised: Carico uses FSVD and assumptions about the collected telemetry to generate values with capacity-based interpretations. While this greatly differs from Pronto and its use of FPCA, Carico still exploits the resulting subspace estimate along with the incoming data to reveal patterns in recent resource-usage.

Comparable: Pronto is a binary signal, which makes it difficult to score Nodes. Carico's Pod-Capacity $\in \mathbb{R}$, allowing Nodes to be filtered and scored against each other.

Latency Resilient: Unlike PRONTO which assumes no communication latency, CARICO was designed with Kubernetes in mind, and thus must consider possible latency in communication and Pod startup. CARICO's Pod-Capacity's unit of measure allows the central scheduler to easily track the estimated cost of Pods in-flight, ensuring subsequent scheduling decisions do not mistakenly overload a Node with a high Pod-Capacity.

3.8 Related Work

This section briefly identifies how CARICO fits into the current landscape of Kubernetes schedulers.

3.8.1 Federated and Distributed Scheduling

PRONTO is described as a federated algorithm that executes plans in a decentralised fashion. Each computing node makes independent decisions for task assignments without needing global synchronisation. Custom Kubernetes schedulers have been designed to extend the Kubernetes architectures with modular, two-level, or distributed architectures [? ? ? ?]. However, the new federated scheduler returns to a centralised architecture, mirroring the default Kubernetes scheduler (`kube-scheduler`) with a single dispatcher on the master node and has access to a global view of the Nodes.

3.8.2 Performance-Aware and Predictive Scheduling

As a performance-aware scheduler, CARICO employs dimensionality-reduction technique to efficiently process the vast amount of telemetry data produced by each Node, with the goal of predicting a Node's capacity for incoming workloads. While, CPU and RAM utilisation are common metrics for performance-aware Kubernetes schedulers [? ? ? ? ?], CARICO distinguishes itself with its ability to work with any metric that can be [0,1]-normalised such that 0 indicates full capacity available and 1 indicates no capacity remaining.

Furthermore, the dynamic nature of Kubernetes workloads strongly suggest that predictive techniques could improve scheduling efficiency and resource utilisation, and still remains an open research topic [?].

3.8.3 Machine Learning-Based Schedulers

Machine Learning (ML) algorithms are increasingly adopted in scheduling to learn from data and improving decision quality. These range from Deep-Reinforcement learning-based schedulers [? ? ? ?] that use reward functions to continuously improve scheduling decisions, to domain-specifc to application or resource utilisation predictions [? ? ?

]. Instead, Carico exploits properties of simple SVD to learn characteristics of recent workloads, giving it broad applications with minimal complexity.

3.8.4 Summary of Related Work

In summary, Carico distinguishes itself within the Kubernetes scheduling landscape through its combined federated and centralised nature, and use of FSVD to predict a Node's capacity for incoming workloads.

Chapter 4

Implementation

4.1 System Architecture

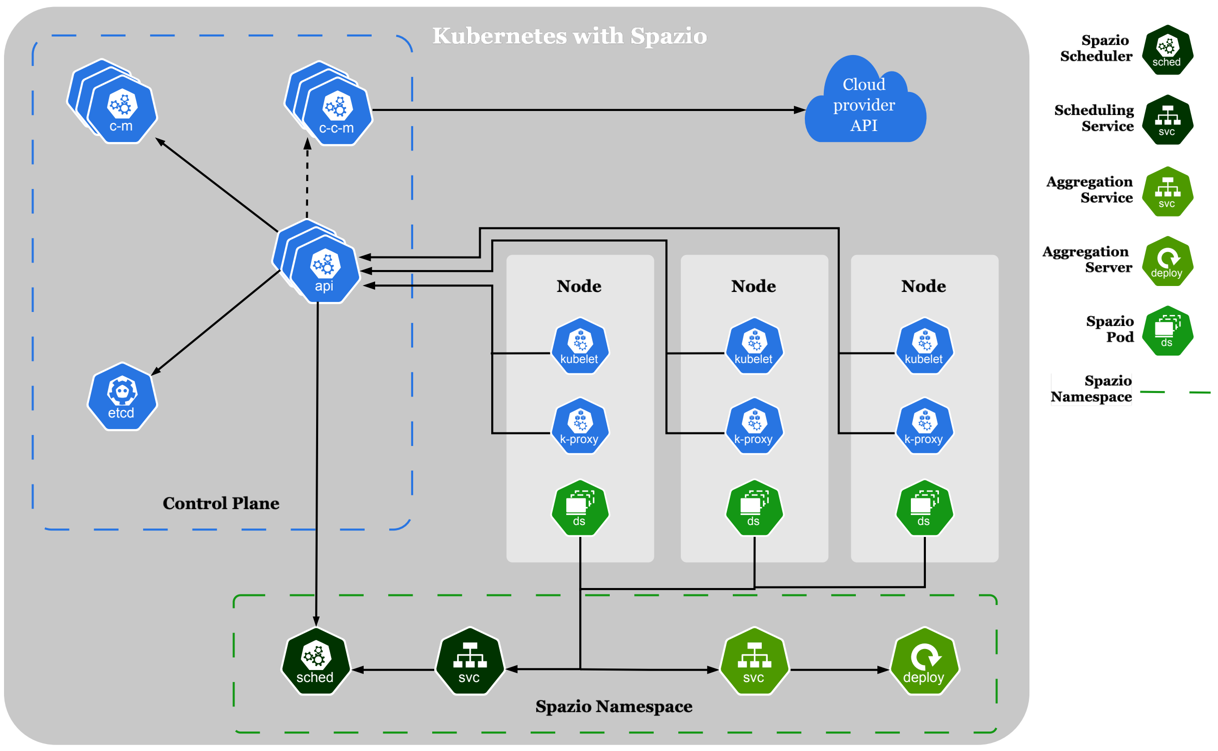


Figure 4.1: The components within the CARICO system

The CARICO system consists of three core components (Figure ??):

- CARICO DaemonSet: This Pod collectes telemetry from a Node and generates its subspace and Pod-Capacity signal. This signal is periodically sent to the Scheduling Service. When the CARICO Pod deems its subspace outdated, it requests the latest aggregated subspace from the Aggregation service.
- Aggregation Server: This Pod performs Subspace-Merge on incoming subspaces, and returns the latest aggregated subspace.

- Scheduler: The scheduler filters and scores Nodes based on their latest Pod-Capacity.

4.2 Carico Pod

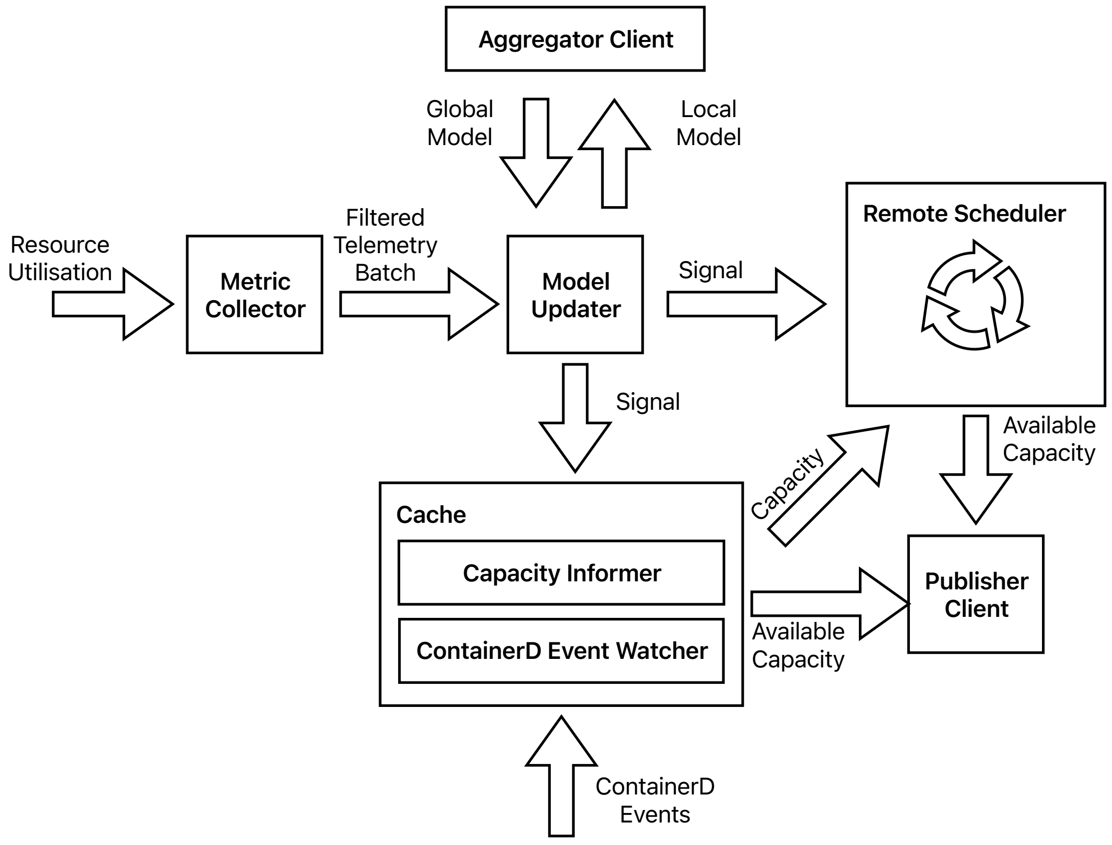


Figure 4.2: Core components within the Carico Pod

The goal of Carico Pods is to accurately generate its Node's Pod-Capacity. This necessitates high quality telemetry for subspace creation and effective signal processing techniques to estimate Baseline and Per-Pod-Cost. Kubernetes dynamically changing workloads also required Carico Pods to frequently update its subspace and Pod-Capacity signal. However, updating too frequently could result in large overheads and reduce the available resources. Investigations using the prototype showed that an update frequency of 1Hz achieved the best results.

4.2.1 Metric Collection

To achieve an update frequency of 1Hz, Carico Pods must collect batches of telemetry data every second. To construct a batch of telemetry, Carico Pods must periodically poll resource metrics and [0,1]-normalise them. This polling frequency is also limited by the overhead it incurrs. I explored two primary sources of telemetry data:

- Metrics Server: a cluster add-on that acts as a centralised source of container reosurce metrics.

- `/proc/`: a pseudo-filesystem within Linux that exposes real-time information about running processes and system's hardware.

Metrics Server uses a scraper to periodically (default every 15 seconds) collect resource metrics from Kubelets and exposes them from its `metrics.k8s.io/v1` APIService. While simple to use, it offers limited metrics (CPU and RAM utilisation only) and introduces additional latency. Moreover, its default scraping interval is too infrequent, potentially missing short-lived Pods entirely.

`/proc/`, conversely, offers low latency access to an up-to-date view of the current state of the system. Furthermore, `/proc/` contains various files and subdirectories, each providing specific information about different types of resources. Furthermore, these sources are not generated periodically, but rather on-the-fly. This guarantees that the information you see is as current as the system's internal state and allows for polling of resource metrics at higher frequencies. CARICO Pods use a polling frequency of 10Hz, providing enough samples to accurately construct the subspace (10 samples) without incurring too large of an overhead.

The next challenge was to select resource metrics, with which to build the subspace, that could be [0,1]-normalised to indicate full or no capacity. The subsequent subsections detail the resource metrics I considered and the rationale behind their use.

Utilisation Metrics

Standard utilisation metrics (e.g., CPU and memory percentage usage) are widely used in industry [? ?].

|/proc/stat— file reports the cumulative count of "jiffies" (typically hundredths of a second) each CPU spent in a specific mode [?]. The [0,1]-normalised CPU utilisation can then be calculated using:

$$\text{CPU Usage\%} = 1 - \frac{\Delta\text{idle} + \Delta\text{iowait}}{\Delta\text{across all fields}}$$

In addition, `/proc/meminfo` shows a snapshot of the memory usage in kilobytes. The [0,1]-normalised memory usage can then be calculated from the given field:

$$\text{Memory Used\%} = 1 - \frac{\text{MemFree} + \text{Buffers} + \text{Cached}}{\text{MemTotal}}$$

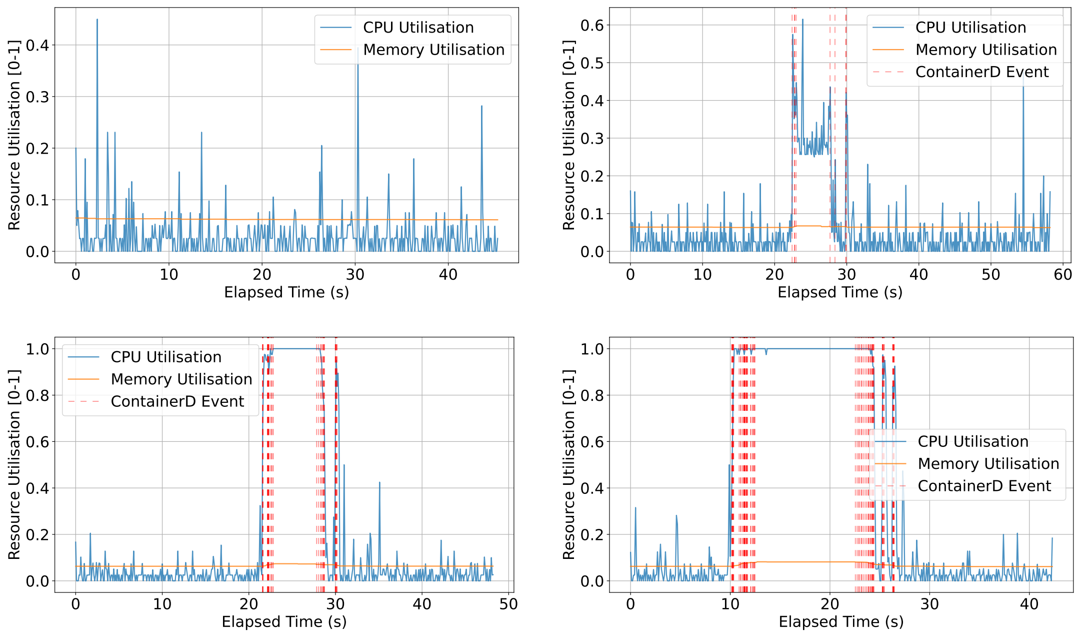


Figure 4.3: In this figure we sample CPU and memory utilisation from values of `/proc/stat`, `/proc/meminfo` at 10Hz during various Kubernetes workloads.

Figure ?? demonstrates how the [0,1]-normalised metrics can quickly respond to the resource usage workloads of different sizes.

Issues of using CPU Utilisation

Early prototypes using only utilisation metrics showed poor throughput compared to the default `kube-scheduler`. When Pods requested 100 milliCPU the `kube-scheduler`, Nodes would have ≈ 45 Pods running on them at once. In contrast, Carico, using only utilisation telemetry, would allocate at most 5 Pods concurrently per Node. Although both approaches achieved 100% CPU utilisation, `kube-scheduler` achieved a short Job Completion time while having a long-tailed distribution of individual Pod Completion times.

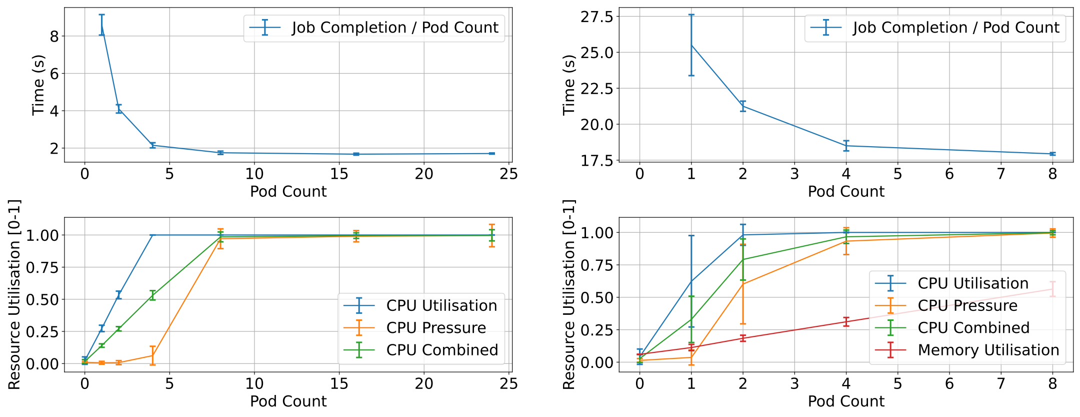


Figure 4.4: Collected resource metrics during various Kubernetes workload.

Further investigations, shown in Figure ??, revealed that even after a Node achieves 100% CPU utilisation, the contribution of each Pod to the Job Completion time decreases: running more Pods on a Node resulted in higher throughput. As the cluster runs on virtual machines (VMs), I hypothesise that the hypervisor inadvertantly masks contention effects like cache and CPU thrashing when providing a hardware abstraction. Consequently, high CPU utilisation does not correlate with performance degradation, and thus, is not an accurate measure of capacity.

Combining CPU Utilisation and CPU Pressure

[0,1]-normalised `/proc/pressure` total metrics,

$$\text{CPU Pressure} = \Delta\text{total} \times \text{polling frequency}$$

aren't sufficient as they do not initially exhibit a linear relation with Pod count (Figure ??). This would lead to unreasonably low intial Per-Pod-Cost estimations and an inflated advertised Node Pod-Capacity. However, by combining CPU utilisation and CPU pressure,

$$CPU = \frac{\text{CPU Utilisation} + \text{CPU Pressure}}{2}$$

the resulting metric exhibits a pseudo-linear relation with Pod count and doesn't saturate to quickly (Figure ??): it ouputs 1 when `/proc/pressure` indicates persistant CPU demand (at least one thread always waiting)

4.2.2 Filtering Metrics

While Carico does not perform peak detection, its Capacity signal is still susceptible to short-lived spikes. As previously noted, Pod creation and deletion incur visible resource usage spikes. These spikes introduce noise into both the Node's subspace and its Capacity signal. As a noisy signal would result in inaccurate baseline Capacity and Per-Pod-Cost estimation, a low-overhead signal filter was required.

Dynamic Exponential Moving Average (EMA) automatically adjusts its smoothing factor α, and thus its responsiveness, based on predefined conditions. As first-hand investigations showed that container events resource spikes last ≈200 milliseconds, Dynamic EMA can be used to suppress container-event caused spikes (typically ≈ 200ms) using a small α_{slow}, while sustained changes (spikes > 300ms) causes it to switch to a larger α_{fast} for rapid convergence.

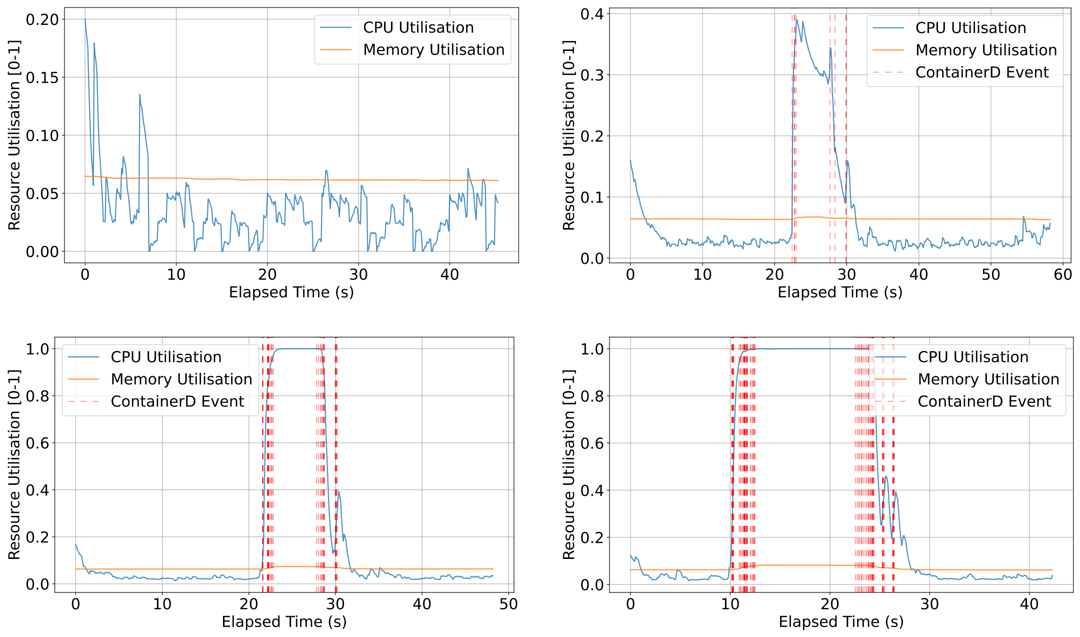


Figure 4.5: This figure shows the smoothed metrics under different workloads.

Comparing Figure ?? and Figure ?? reveals the impact of Dynamic EMA on the collected telemetry.

4.2.3 Signal Generation

To verify Capacity signal's implementation, I test a Carico Pods under the scenarios described in Section ??.

Figure ?? shows the Capacity signal of a Node when running a light CPU-focused workload (`ng-stress --cpu=8 --cpu-load=25`), while the remaining Nodes in the cluster execute a CPU-intense workload (`bpi(2000)`). The measured Node's capacity signal drops once the CPU-intense workload is scheduled on its peers. This is expected as the subspaces of surrounding Nodes, and thus the aggregated subspace, will reflect a more costly CPU-focused workload (larger σ_1). After aggregating its subspace, combinings its unchanged resource usage with the new increased expected workload (reflecting in $\sigma_1 u_1$) results in a smaller k (fewer units of this workload can be added), and thus a lower Capacity signal.

Figure ?? demonstrates the Capacity signal of a Node when running a light Memory-focused workload (`ng-stress --vm=4 --vm-bytes=4G`) with the remaining Nodes in the cluster execute the same CPU-intense workload as in the earlier scenario. Like before, the global aggregated subspace reflects a heavy CPU-focused workload. However, when the measured Node combines this new expected workload with its current memory-focused resource usage, a larger k is required to reach a resource limit.

We can conclude that the Carico Pods behave correctly.

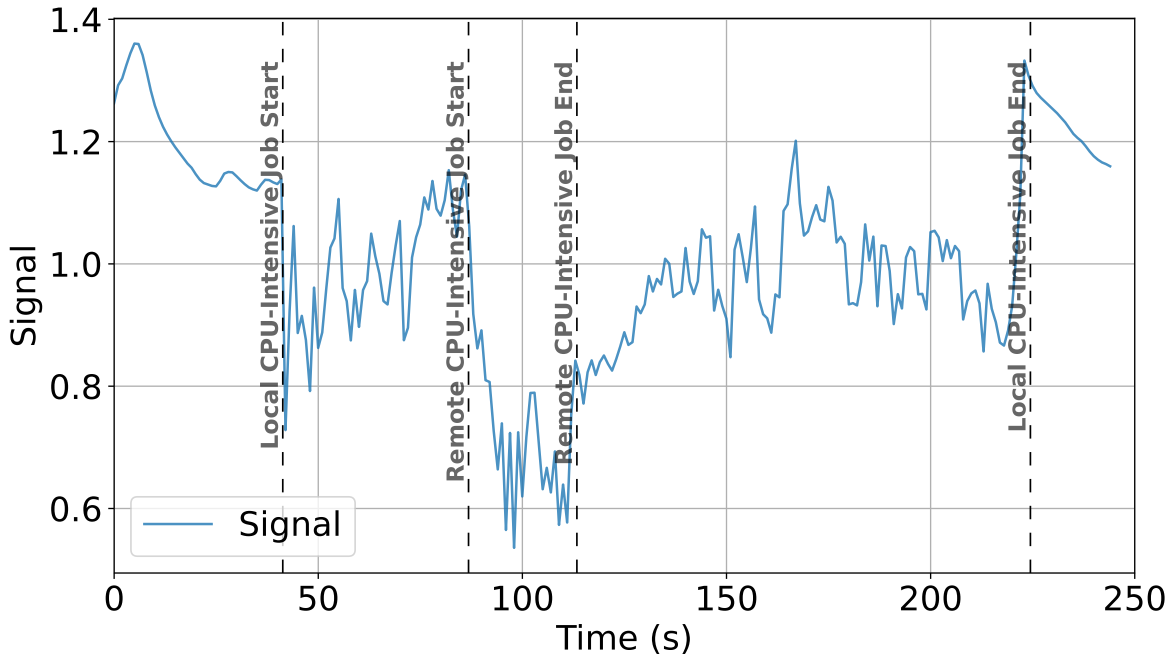


Figure 4.6: The calculated capacity signal of a Node running `ng-stress --cpu=8 --cpu-load=25`. While running this workload, 1000 Pods running `bpi(2000)` were scheduled across surrounding Nodes.

4.2.4 Calculating Cost and Capacity

To implement a reservation mechanism, Carico assumes the ability to measure Pod count and estimate the Baseline Capacity Signal and Per-Pod-Cost. Due to the resource constraints of the Carico Pods, the chosen estimation technique must be lightweight and operate on a single pass of telemetry. Furthermore, Kubernetes fast-paced and noisy environment, means the techniques must handle handle dynamic workloads and noise from container churn.

Detecting Pod Events

The quality Capacity signal is crucial to the accuracy of estimations. Dynamic EMA can't fully eliminate the noise from the container runtime (shown in Figure **??**). Therefore, some noise will bleed into the Capacity signal and reduce the accrucy of estimates. To combat this, early detection of container events can halt estimations during these bursts. Thus, the requirements for an event listener are:

- Detect the creation and deletion of Pods to establish a Pod count
- Provide warning for potential container-caused churn

I investigated two event detection approaches: watching the Kubernetes API vs ContainerD events. Figures **??** and **??** show that communication latency from the Kubernetes API causes its listener to miss container runtime resource spikes, while initial container

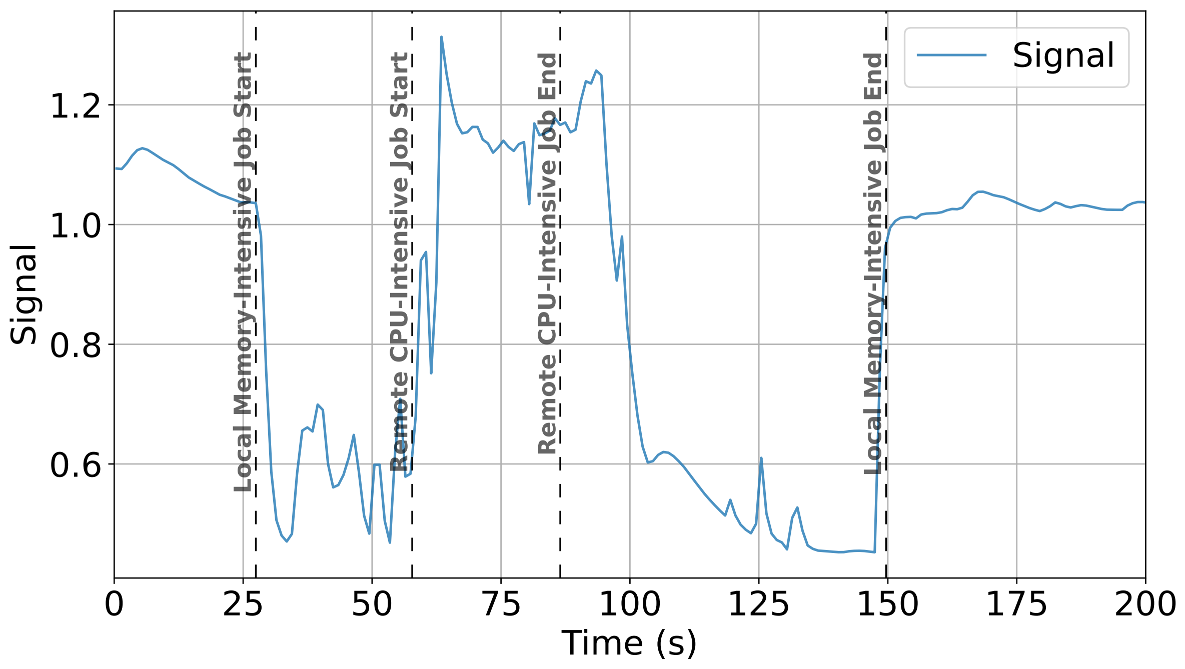


Figure 4.7: The calculated capacity signal of a Node running `ng-stress --vm=4 --vm-bytes=4G`. While running this workload, 1000 Pods running `bpi(2000)` was scheduled across surrounding Nodes.

events precede the spikes. Though more complex, handling ContainerD events directly provides sufficient warning for container churn.

Estimating Pod-Cost

A Kalman filter [] is a powerful lightweight streaming algorithm used for estimating the true state of a dynamic from a series of noisy and uncertain measurements. It's widely applied in fields like navigation (GPS), robotics, signal processing and control systems. Its ability to estimate a system's state from noisy measurements makes it suitable for this dynamic environment.

I devised three Kalman filter-based approaches to estimating reservation costs.

- 1D Kalman Filter predicting reservation cost based on the function:

$$\Delta\text{Capacity signal} = \Delta\#\text{ Running Pods} \times \text{Per-Pod-Cost}$$

- 2D Kalman Filter to predict the signal based on the function:

$$\text{Capacity signal} = \text{Baseline Capacity signal} - \text{Per-Pod-Cost} \times \#\text{ Running Pods}$$

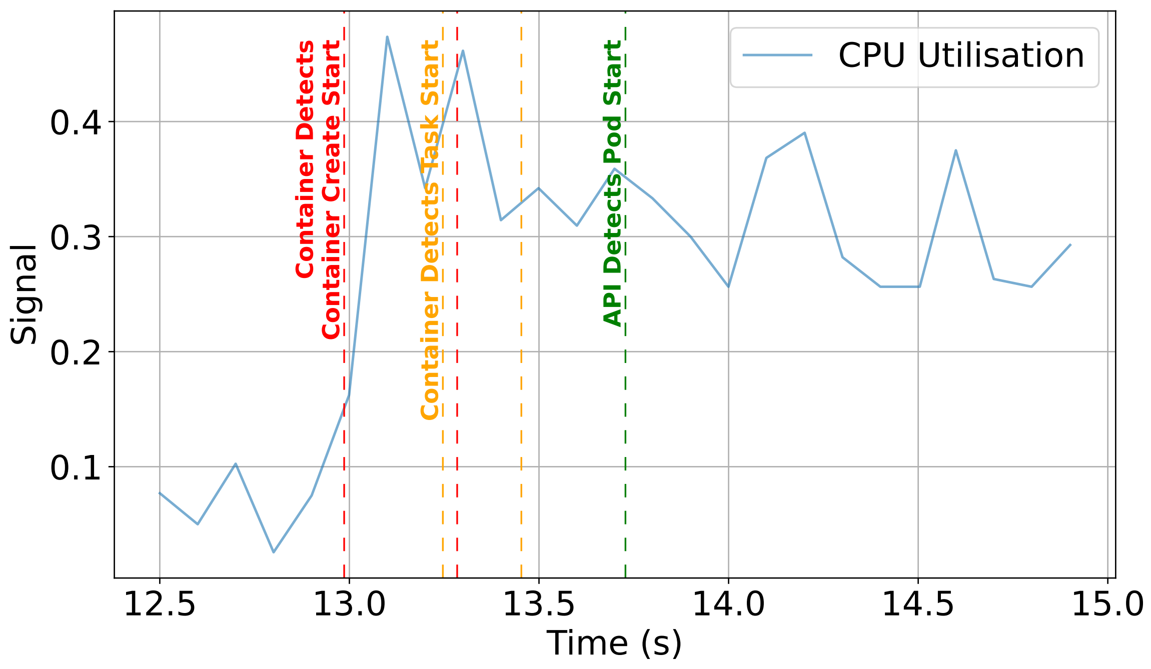


Figure 4.8: When different event listeners detected the creation of a Pod.

- Two separate 1D Kalman Filters predicting the equation:

  $$\text{Capacity signal} = \text{Baseline Capacity signal} - \text{Per-Pod-Cost} \times \# \text{ Running Pods}$$

  Here, each filter learns a separate variable. This separation prevents non-zero covariance entries in the state covariance matrix ($\mathbf{P}$), mitigating oscillations observed with a single 2D Kalman filter.

To ensure accurate estimates, estimations are halted whenever the Capacity signal reaches zero. When this occurs, it indicates that at least one resource is fully utilised, and additional Pod won't further decrease the observed Capacity signal. Filters may account for this behaviour by inaccurately decrease the Per-Pod-Cost, leading to an inflated advertised Pod-Capacity.

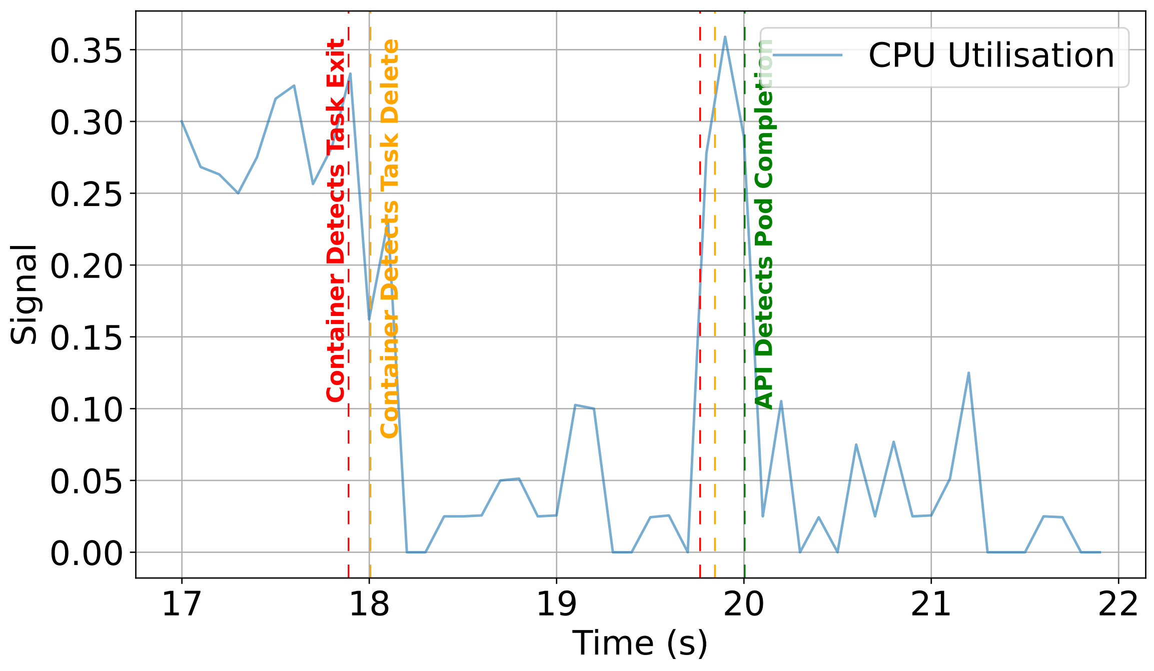


Figure 4.9: When different event listeners detected the completion of a Pod.

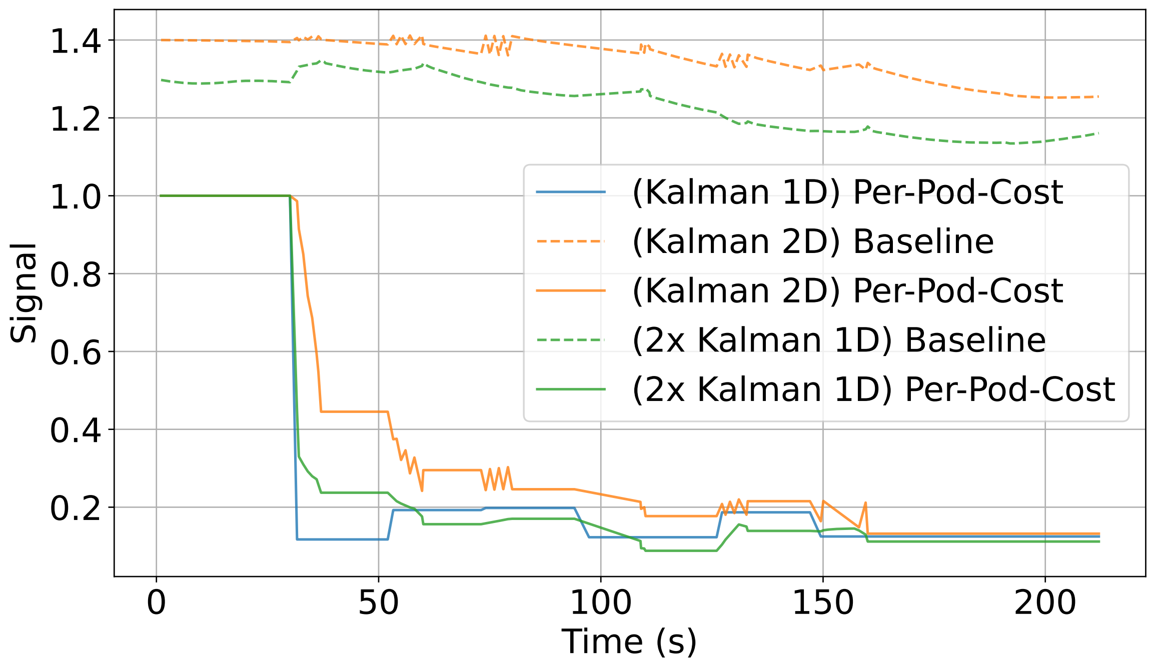


Figure 4.10: The estimates of the Kalman filters when Node experiences variable-sized bursts of `bpi(2000)` Pods.

To identify the optimal approach, I observed each methods' when under the same workloads (shown in Figure **??**). The Δ-based Kalman filter accurately estimated Per-Pod-Cost, but being one-dimensional, could not estimate the Node's baseline Capacity signal. The 2D Kalman filter approach provides a simple method for estimating both the Node's capacity and its per-Pod cost. Faster convergence for the 2D filter was attempted using large process noise covariance (**Q**) values. This, however, led to large oscillations, as the filter adjusted both baseline capacity and cost variables to correct errors. The dual 1D Kalman filter was inspired by the stability of the 1D Kalman filter. This filter converged quickly and accurately without exhibiting the oscillations that plagued the 2D Kalman filter. Therefore, this final approach was used in the CARICO Pod implementation.

4.3 Aggregation Server

The purpose of the Aggregation Server is to perform Subspace-Merge on incoming local subspaces and return the global aggregated subspace. PRONTO's approach resembles the distributed agglomerative summary model (DASM) []. Subspaces are aggregated in a "bottom-up" approach following a tree-structure depicted in Figure **??**. While this approach reportedly requires minimal synchonisation, it demands multiple dedicated Pods, and Kubernetes' inherent communication latency could significantly delay global model updates after workload changes.

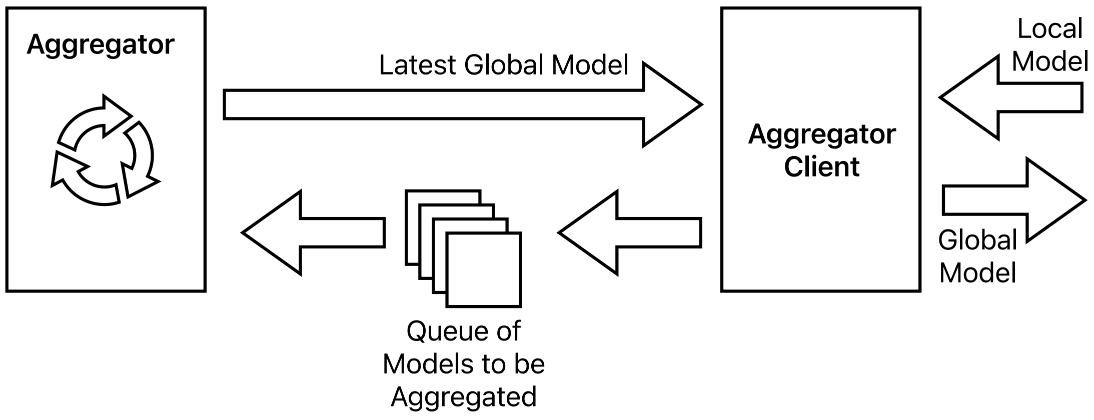


Figure 4.11: The components within the Aggregator Server.

Instead CARICO opts for a "flat" on-the-fly aggregation approach (shown in Figure **??**). CARICO Pods request the latest aggregate subspace with a gRPC (go Remote Procedure Call) and their local subspace. The Aggregation Server corresponding gRPC function enqueues the input subspace to be aggregated and returns its latest view of the global model, which the CARICO Pods then aggregate locally with their subspace. Decoupling model aggregation from the gRPC call's critical path reduces server load. Furthermore, as the Aggregate Server does not wait for all the pending subspaces to have been aggregated, it trades consistency for latency.

The Aggregation Server also executes a long-running thread which waits on a queue of subspaces to aggregate. When the queue is non-empty, the thread pops a subspace at a time and performs a Subspace Merge operation (as defined in ??) with its latest aggregated subspace. To balance the influence of each subspace, weights $\gamma_{\mathbf{A}} = \#$ of Nodes $- 1$ and $\gamma_{\mathbf{B}} = 1$ are used in the Subspace-Merge (as per Section ??).

While not implemented in this prototype, CARICO could scale to larger clusters with multiple Aggregation Servers connected to a load balancer. These aggregation Servers could then periodically merge their aggregate subspaces to ensure consistency across Servers.

4.4 Scheduler Pod

The purpose of this component is to allocate Pods to Nodes, based on the Nodes' latest Pod-Capacity scores. Furthermore, it keeps track of all Pod reservations.

4.4.1 Kubernetes Scheduler Plugin

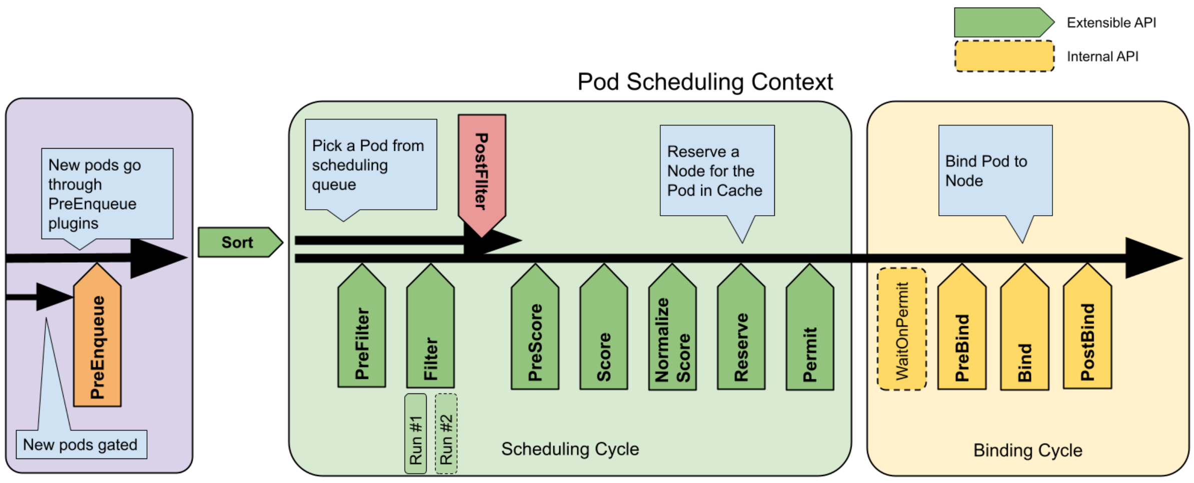


Figure 4.12: The available scheduling framework extension points [?].

The scheduling framework (depicted in Figure ??) is a pluggable architecture for the Kubernetes scheduler. It defines extension points at which scheduler plugins register to be invoked. A scheduler plugin can register to multiple extension points, with each extension point defining an interface: a set of functions the scheduler plugin has to implement.

Developing a CARICO-based scheduler plugin, rather than a standalone scheduelr, simplified implemention and offered many performance benefits. First, CARICO's **filter**, **score**, and **reserve** operations (given in Section ??) have one-to-one mapping to the framework's extension points. Second, the framework provides access to efficient, pre-existing data structures and algorithms. Finally, plugins allow customisation through selective enabling/disabling of default plugins and ordering of custom ones. This means that the CARICO plugin could be used in tandem with other plugins to improve scheduling decisions.

To track Pod-Capacity reservations, the Carico scheduler uses a `map` (Pod ID to Node ID) and a Kubernetes API Pod Event Listener. During the Reserve phase, the scheduler increments the target Node's reserve count and records Pod-Node assignment. When the API listener detects a Pod transitioning from the 'Pending' status, if the Pod is in its reservation `map`, the scheduler decrements the corresponding Node's reserved count.

Chapter 5

Evaluation

This section is focused on the empirical evaluation of CARICO as a Kubernetes scheduler, comparing it against the default industry-standard `kube-schedulers`. I was unable to compare CARICO against any similar telemetric-only schedulers due to their limited number and time constraints. For the evaluation, I use a set of common scheduling objectives used by datacenter providers [**?**]:

Job Completion - Section ??:
Schedulers often aim to reduce the time it takes for a job or a set of jobs to complete. In Kubernetes, Jobs are specified by a template of the Pods to be executed, the total number of Pods to complete (`completions`) and the maximum number of Pods running at once (`parallelism`). Job completion is defined as the time between a Job object being published to the Kubernetes API and the time when the last Pod of the Job completes. During the experiments, I set `parallelism` to `completions` so that only `kube-scheduler`'s decisions impact Job completion.

Pod Completion - Section ??:
The goal of many datacenter schedulers is to complete jobs as fast as possible so that resources are freed for subsequent jobs. For this section, I investigate the distribution of Pod Completion, the time it takes from when a Pod starts running to when it completes. In addition, I explain my findings using traces of the number of concurrent Pods on each Node during job execution.

Resource Utilisation - Section ??:
Since resources are expensive, datacenters aim to ensure that resources are well utilised with efficient placements. Over-utilisation can result in large amounts of resource thrashing, which wastes resources and hurts a datacenters proffitability. As CARICO currently only uses CPU and memory metrics for scheduling, I only measured the utilisation of those resources during the exection of Jobs.

Workload Heterogeneity - Section ??:

As outlined in ??, datacenters must handle workloads with different characteristics, such as, resource usage and running times. In this section, I evaluate how well Carico is able to handle mutliple Jobs with different characteristics.

Workload Isolation - Section ??:
Datacenters have to deal with jobs with different QoS requirements. One of these requirements is workload isolation: the execution of one Job will not interfere with the execution of another. With this property, users can be guaranteed a minimum level of quality.

Overhead - Section ??:
Due to the limited number of resources that need to be shared between users, datacenter providers aim to mitigate the overhead from scheduling. A lower overhead means more resources are available for users and providers can achieve higher profits margins. For this experiment, I measure the overhead incurred from running the DaemonSet of Carico Pods.

To ensure a fair comparison between Carico (telemetric-only scheduler) and `kube-scheduler` (resource description-based scheduler), I use a range of resource requests with `kube-scheduler` to highlight how much the performance can vary with different resource request.

5.1 Evaluation Setup

These experiments ran on a Kubernetes cluster containing 20 virtual machines (VMs) running on the Xen hypervisor. One of the machines is used as the master Node, and the rest are worker Nodes. The master Node contains all the Pods in the control plane, and during the evaluation of Carico, it contains the Carico Scheduler and Aggregation Server. Each VM features four Intel Xeon Gold 6142 CPUs 2.60Ghz with 8 GB of RAM running Ubuntu 24.04.2 LTS. Each CPU has a single core with hyperthreading disabled. When running `kubectl describe nodes`, each Node advertises 4000 milli-CPU seconds and 8GB of RAM.

During the evaluation, I use a Prometheus deployment [] to collect various statistics, such as running Pod count, resource utilisation and Kubernetes object completions.

5.2 Experimental Workloads

During the experiments I used very short-lived workloads; workloads that take less than a minute to complete. While datacenters can expect to receive longer-running workloads, using them in the evaluation was not feasilble due to limited time and the need to run experiments multiple times. Short-lived tasks can still provide valuable insights into the performance of a scheduler. Due to their short completion time, the cost of poor scheduling decisions becomes more significant and easier to observe.

During this evaluation, I used two types of workloads:

1. `pi-2000`: A short-lived CPU-focused workload where a Pod computes the value of π to 2000 decimal points.

2. `sklearn`: A longer-running workload with a larger memory footprint. This Pod executes a script which uses `sklearn` which trains a small neural network classifier (512 features, 16 classes, 8 hidden layers each containing 1024 neurons) on 5000 randomly generated samples before running inference on another set of 5000 randomly generated samples.

Investigating metrics when running `pi-2000` and `sklearn` Jobs separately demonstrates how Carico handles opposing resource usage characteristics (CPU-focused vs memory-focused). Furthermore, when executing both these Jobs concurrently, the difference in runtime helps investigate how Carico also handles workloads with differnt running times.

In this experiment, I use a Job that specifies Pods that performed a small ML workload. This workload uses a significant amount of memory, which unlike CPU, must be carefully handled. If we increase the number of processes on a fully-utilised CPU, it only results in each process having a smaller portion of CPU time and degrading its performance. On the other hand, memory is less forgiving as once memory runs out, the kernel begins OOM killing processes. This be detrimental to Job Completion, as terminated Pods results in wasted computations.

5.3 Job Completion

Figure **??** demonstrates the Job completion of `pi-2000` with different Pod counts. We can observe that Carico is able to consistently achieve comparable Job completions (within $\approx 10\%$ of the optimal run with `kube-scheduler`) without any prior knowledge of a Pod's resource usage. On the other hand, incorrect Pod requests can result in $\approx 80\%$ increase in Job completion when using `kube-scheduler`.

Figure **??** presents the Job completion of `sklearn` Jobs. Like in Figure **??**, Carico is able to achieves comparable Job completion times using only telemetric data. Similarly, we can also observe the effect of inaccurate resource requests. More importantly, underestimating memory usage can result in Nodes running out of memory and their kernel OOM killing actively running Pods. This often results in Jobs unable to complete and is the reason for why I do not present Job completions for runs requesting less than 750Mi bytes of memory.

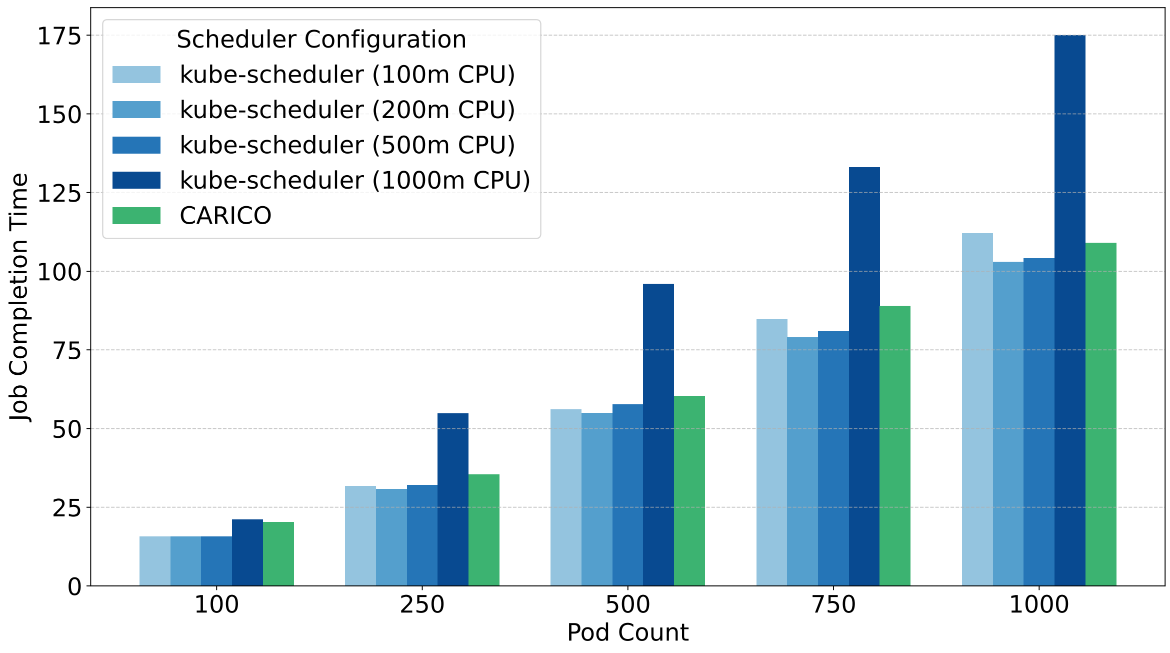


Figure 5.1: The Job completion of `pi-2000` Jobs with different Pod counts. The requested resources are given when using `kube-scheduler`

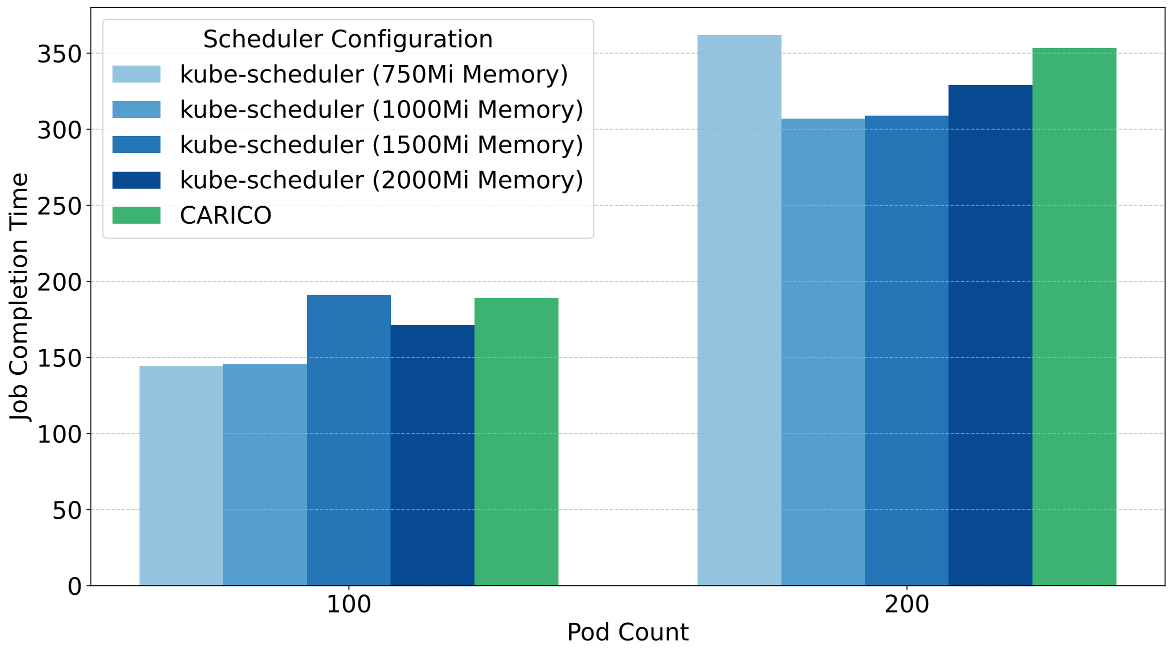


Figure 5.2: The Job completion of `sklearn` Jobs with different Pod counts. The requested resources are given when using `kube-scheduler`

5.4 Pod Completion

In this section, I compare the individual Pod Completion times from the traces of different Job executions.

Table **??** presents the Pod completion distribution of the traces from running a 1000 Pod

Scheduler	CPU Request	Mean	Std.	Min.	25%	Median	75%	Max.
Default	100m	47.43	14.59	7.00	39.00	52.00	57.00	73.00
Default	500m	9.55	1.49	5.00	9.00	10.00	10.00	14.00
Carico		7.69	0.99	5.00	7.00	8.00	8.00	11.00

Table 5.1: Pod Completion of a trace when executing a `pi-2000` Job with 1000 Pods under different schedulers.

`pi-2000` Job under different schedulers. Carico is able to consistently achieve a tight Pod completion distribution with 75% of Pods completing in less than 8 seconds. In contrast, when underestimating resource requests and using 100 milliCPU, the distribution of Pod completion times becomes tail-heavy with 50% of Pods taking longer than 52 seconds to complete and a maximum completion time of 73 seconds. However, even when using the optimal and more conservative resource request of 500 milliCPU, its distribution has a 25% higher mean with 50% more variance.

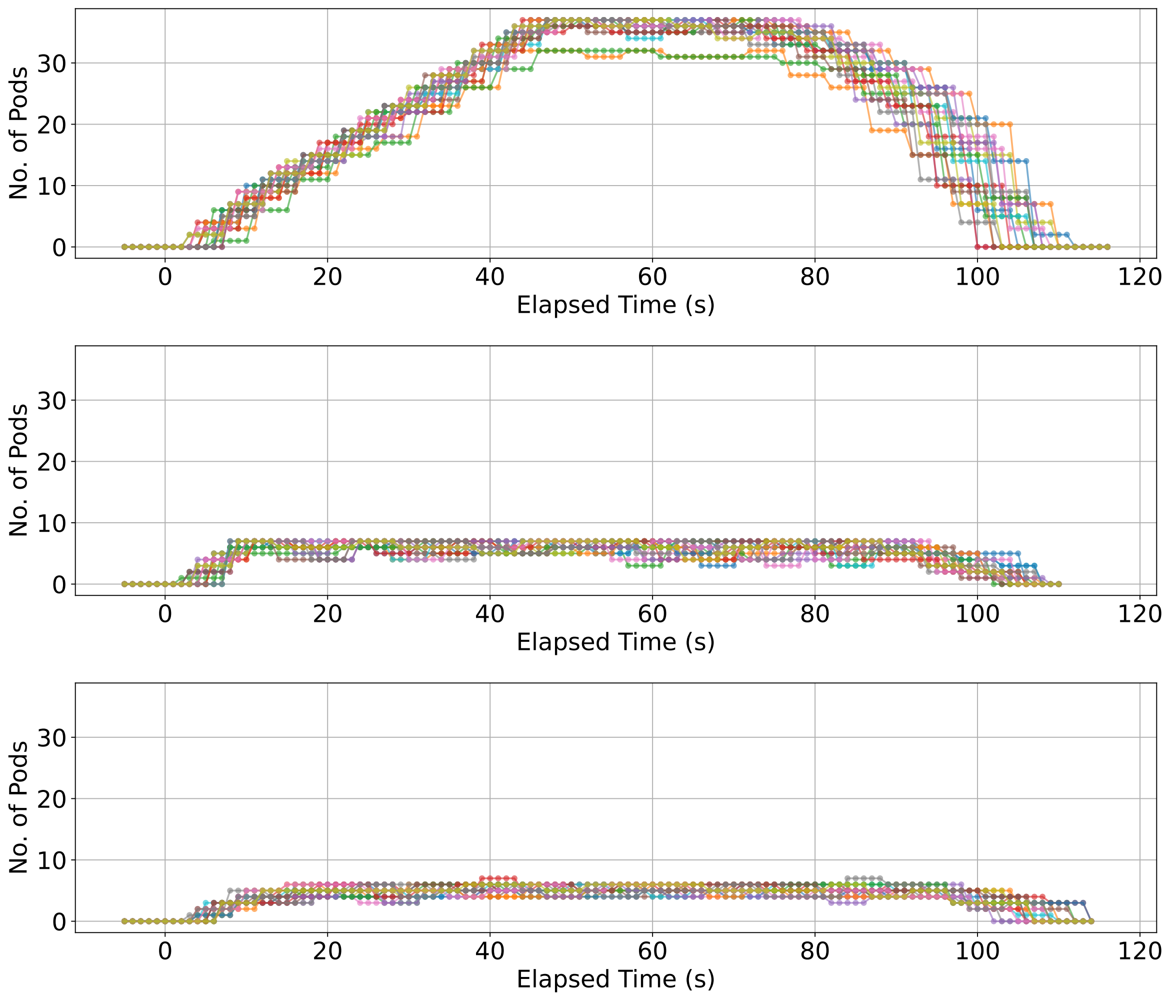


Figure 5.3: The number of `pi-2000` Pods running on a Node during the execution of a Job with 1000 Pods.

To further justify the results seen in Table ??, I plot the number of concurrently running Pods on each Node during the Job traces in Figure ??. When underestimating the resource request, `kube-scheduler` is able to pack more Pods on a Node at once. This results in higher resource contention as more Pods have to share their time on the CPU. With less time on the CPU per second, Pods take longer to finish their computation. On the other hand, the more conservative requests means the `kube-scheduler` is able to pack fewer Pods, meaning each Pod gets a reasonable share of time on the CPU and therefore can complete faster. With Carico, Nodes detect the CPU is at capacity once there is always at least one process waiting for a CPU core. This therefore limits the number of Pods a Node can take and ensures all Pods receive an adequate amount of CPU resources.

Scheduler	**# Pods**	**Mean**	**Std.**	**Min.**	**25%**	**Median**	**75%**	**Max.**
Default	200	225.19	36.19	45.00	231.00	236.00	239.00	244.00
Carico	200	64.46	14.33	23.00	57.75	67.00	73.25	89.00

Table 5.2: Pod Completion of a trace when executing a `sklearn` Job with 200 Pods under different schedulers.

Table **??** presents the Pod completion distribution of the traces from running a 200 Pod `sklearn` Job under different schedulers. Carico is still able to consistently achieve a tight Pod completion distribution with a mean completion time of 64.46 seconds and a variance of 14.33 seconds. On the other hand, when using the lowest overestimate of memory utilisation, 750Mi bytes of memory, `kube-scheduler` achieves a 3.5× higher mean Pod completion time with more than twice the variance.

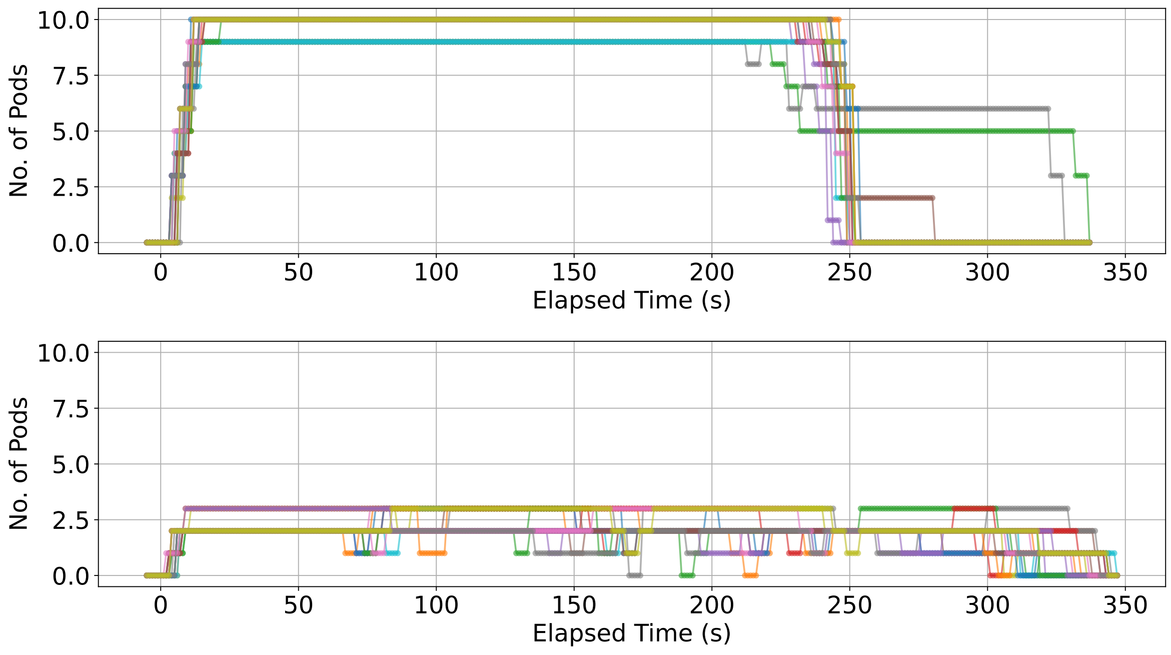


Figure 5.4: The number of `sklearn` Pods running on a Node during the execution of a 200 Pods Job.

Like with `pi-2000`, `kube-scheduler` allocates more Pods onto a Node at once, resulting in higher resource contention, and thus longer Pod completion times. This graph also explains the longer Job completion time. Due to the memory request (750 Mi) and each

Node's capacity (8GB), `kube-scheduler` is unable to allocate all the Pods in one sweep. However, once the Pods on the faster machines (imbalance caused by the hypervisor) complete, `kube-scheduler`'s greedy nature causes it to allocate all the remaining Pods to those machines. The long running time of the `sklearn` Pods combined with the resource contention from allocating the remaining Pods to a small number of machines means the Job must wait longer for the last Pod to complete. On the other hand, Carico ensures a consistent number of Pods are running on a Node.

5.5 Resource Utilisation

In this section, I investigate the resource utilisation during the Job traces explored in Section **??**.

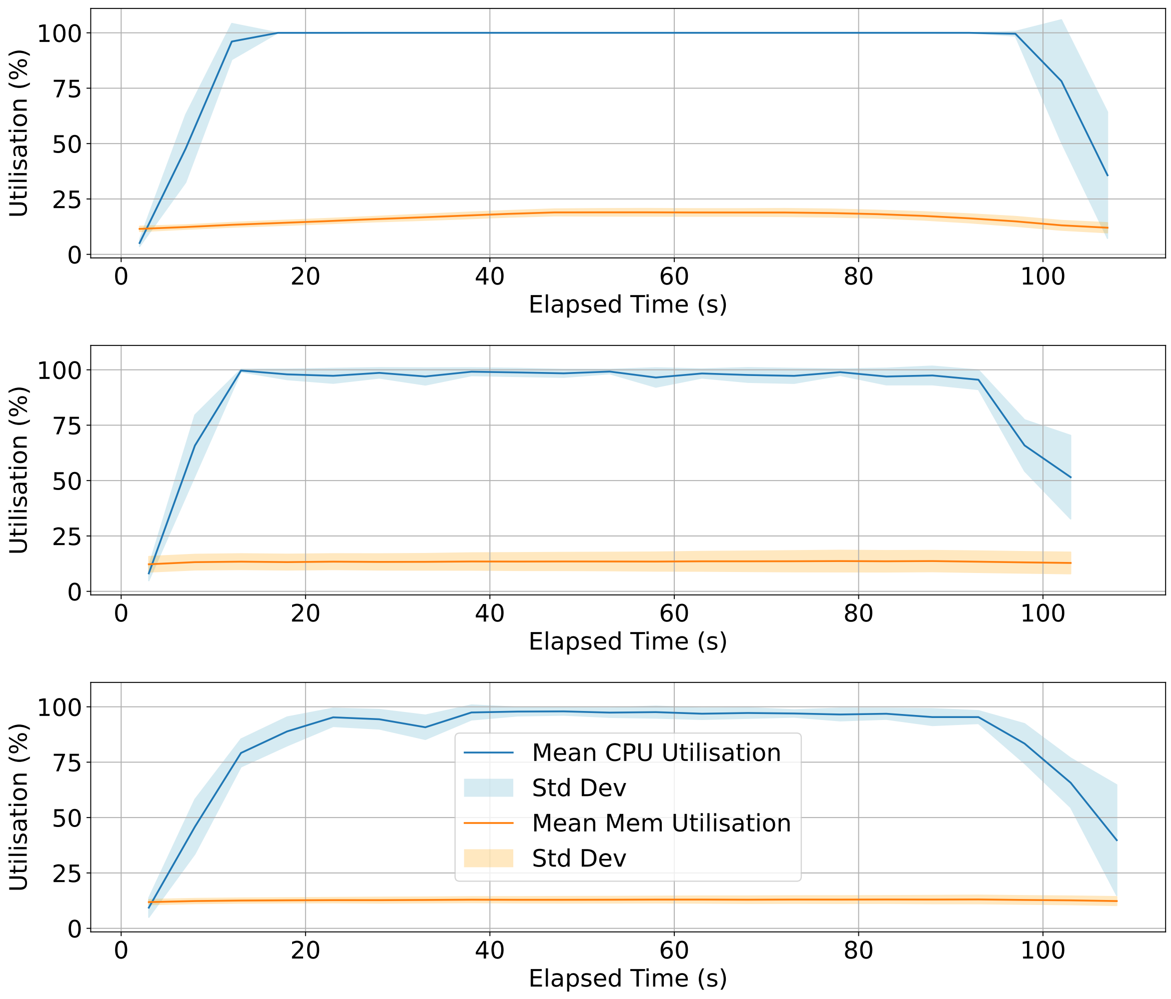


Figure 5.5: Resource utilisation during a trace of executing a `pi-2000` Job with 1000 Pods under different schedulers.

Figure **??** illustrates the resource utilisation during the execution of a 1000 `pi-2000` (CPU-focused) Pod Job. When under-estimating the resource request, `kube-scheduler` is able to easily maximise the CPU utilisation, with all Nodes reading 100% CPU utilisation.

The more conservative 500 milliCPU request is still able to achieve close to 100% CPU utilisation. Although, Carico takes a more than twice as long to reach $\geq 95\%$, it achieves consistently high CPU utilisation.

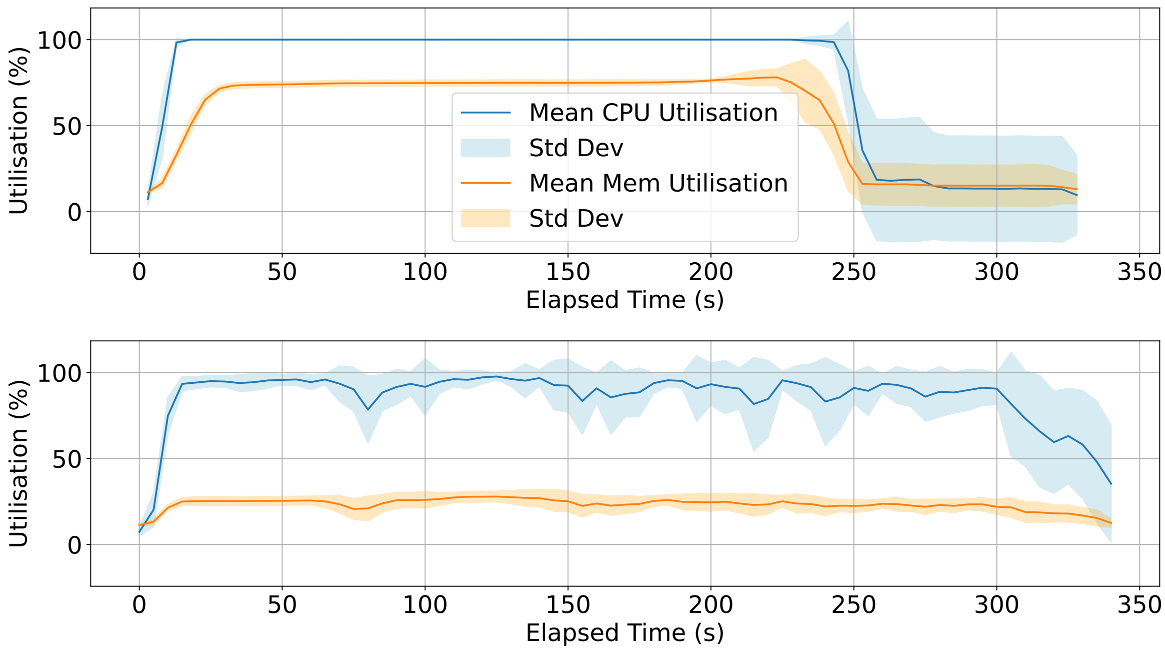


Figure 5.6: Resource utilisation during a trace of executing a `sklearn` Job with 200 Pods under different schedulers.

Figure ?? presents the resource utilisation during the execution of a 200 `sklearn` (memory-focused) Pod Job. While Carico again achieves high CPU utilisation, it fails to fully utilise memory. This indicates that even with `sklearn`'s significantly large memory footprint, CPU still became the bottleneck resource. As a result, Nodes would limit capacity before fully utilising memory. Contrastingly, as kube-scheduler does not use live-telemetry, it is able to pack enough `sklearn` pods to achieve a higher memory utilisation. However, this trace also highlights problems caused by `kube-scheduler`'s greedy bin-packing. Figure ?? from Section ?? painted the scenario in which `kube-scheduler` results in a mostly idle cluster waiting for a small number of machines to finish executing the remaining Pods. We can observe this impact on resource utilisation, as during the last minute of the Job's execution, the clusters average CPU and memory utilisation is around $\approx 10\%$.

5.6 Workload Heterogeneity

To investigate how Carico handles workloads with different characteristics, I executed both the `pi-2000` and `sklearn` Jobs. `pi-2000` Pods requests 100 milliCPUs, while `sklearn` Pods requested 200 milliCPUs and 750Mi of memory.

Throughput

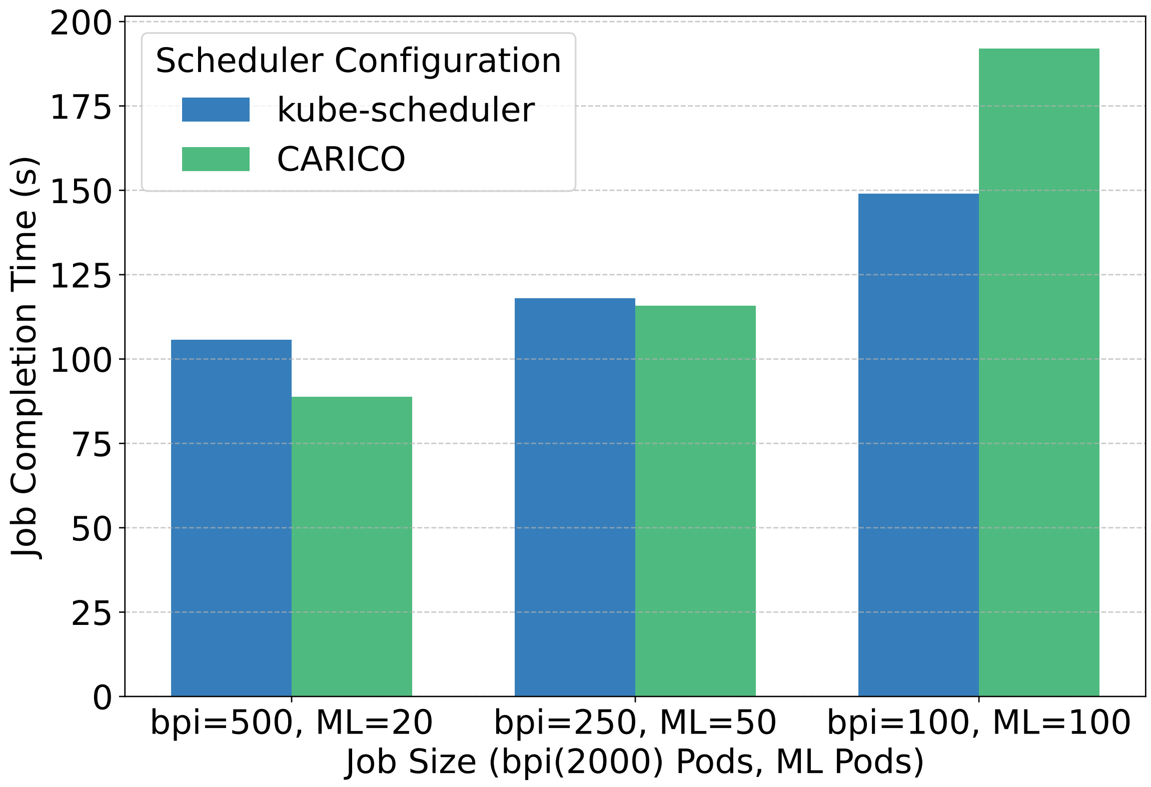


Figure 5.7: Job Completion of multi-resource Job deployments with different Pod counts. For the default scheduler, `bpi(2000)` Pods requested 100 milliseconds of CPU and ML Pods requested 200 milliseconds CPU and 750Mi of memory

Figure ?? presents the Job Completion time when running `pi-2000` and `sklearn` Jobs with varying distributions. Carico is able to achieve lower Job completion times when executing a larger portion of `pi-2000`. However, once a significant portion of the workloads are long-running `sklearn` Pods, `kube-scheduler` edges ahead. This is likely because the interference from `pi-2000` Pods are small enough where the performance resembles that of running only `sklearn` Pods.

Pod Completions

Scheduler	**Job**	**Mean**	**Std.**	**Min.**	**25%**	**Median**	**75%**	**Max.**
Default	`pi-2000`	28.78	7.52	7.00	27.00	30.00	33.00	45.00
Default	`sklearn`	65.25	10.53	49.00	58.25	64.00	75.25	82.00
Carico	`pi-2000`	7.40	1.14	5.00	7.00	7.00	8.00	11.0
Carico	`sklearn`	34.50	7.39	22.00	27.50	36.00	40.25	44.00

Table 5.3: Pod Completion of a trace when concurrently executing a 500 Pod `pi-2000` Job and a 20 Pod `sklearn` Job.

I also investigated how Pod Completion time changes when running diverse workloads.

Table ?? presents the distributions of the Pod Completion for both `pi-2000` and `sklearn` Jobs. From it we can see how Carico continues to achieve significantly lower mean Pod Completion times with both Jobs than `kube-scheduler`. Furthermore, the achieved Pod Completion times vary significantly less with Carico ($\approx 7\times$ lower) than those of `kube-scheduler`.

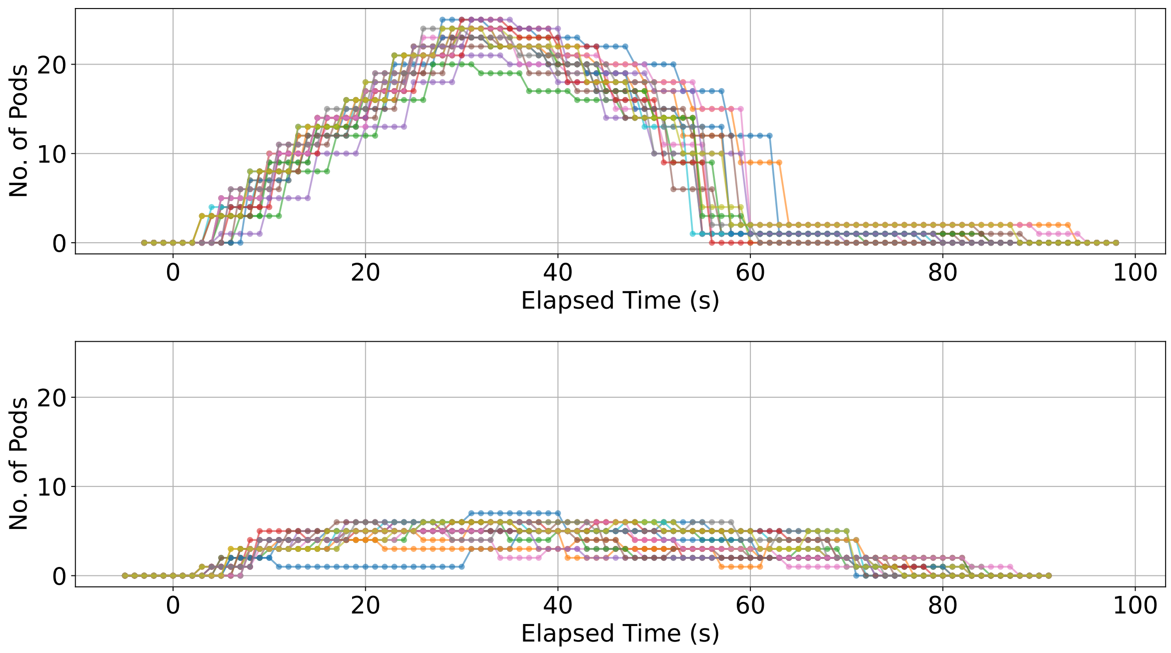


Figure 5.8: The number of Pods running on a Node when executing concurrently a 500 Pod `pi-2000` Job and a 20 Pod `sklearn` Job.

Further investigation into the number of Pods running on each Node, depicted in Figure ??, shows how Carico achieves a consistent number of running Pods throughout the job, while with `kube-scheduler` the number of running Pods varied from 25 Pods to less than 3 Pods.

Resource Utilisation

Figure ?? shows the resource utilisation when executing a 500 Pod `pi-2000` Job and a 20 Pod sklearn Job concurrently. We can see the impact of `kube-scheduler`'s varying number of Pods running on each Node. When Nodes have more than 20 Pods running on them, they advertise 100% CPU utilisation, which is indicative of high resource contention. Then, when only a few `sklearn` Pods remain on a Node, the overall mean CPU utilisation of the clusterdrops to less than 75%. In constrast, Carico achieves a high CPU utilisation $\geq$ 95% without experiencing extreme contention. Furthermore, its balanced allocation ensures a high CPU utilisation throughout the entire execution time of the Jobs.

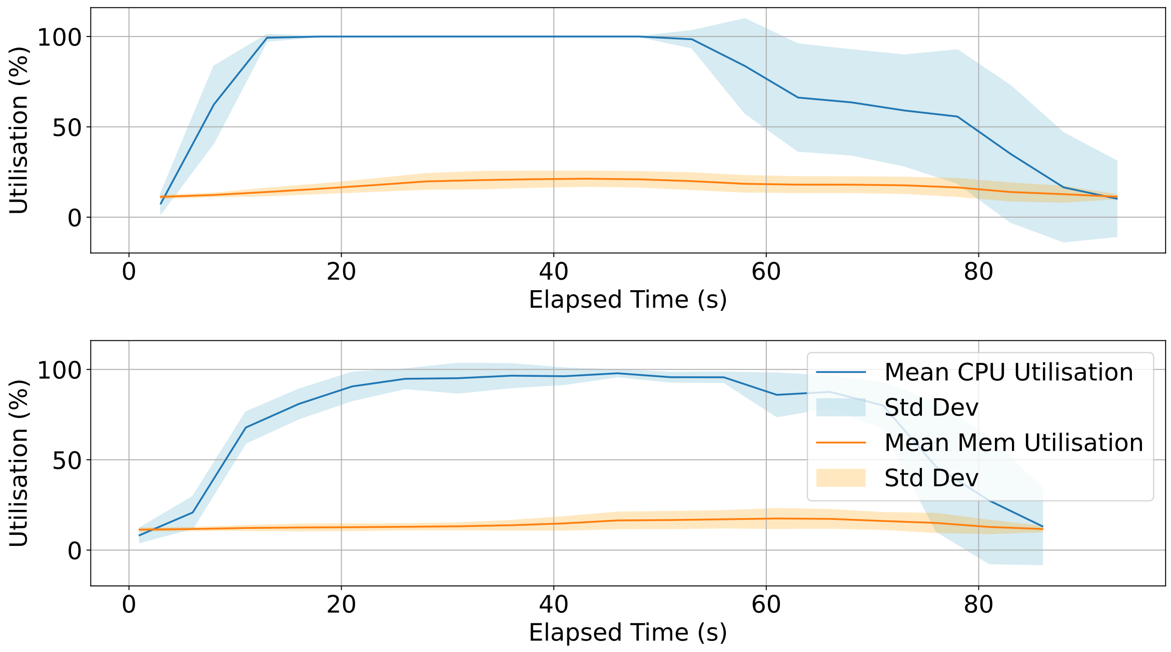


Figure 5.9: Resource utilisation during a trace of executing concurrently a 500 Pod `pi-2000` Job and a 20 Pod `sklearn` Job.

5.7 Workload Isolation

To evaluate Carico's QoS, I investigated how its scheduling decisions impact the performance of already running Pods. For this experiment, I had a Pod running on a worker Node, while another Pod periodically sent HTTP GET requests. This polling Pod would then measure latency of the response. I then scheduled a Job of 1000 Pods executing `bpi(2000)` across the cluster and measured how the response latency changed. In the default schedulers case, each Pod requested 100 milliseconds of CPU time.

Scenario	**Response Latency (ms)**					
	Min	**Med**	**P90**	**P95**	**P99**	**Max**
Baseline	0.99	3.04	3.78	4.00	4.47	8.32
Default Scheduler	1.07	10.06	18.61	22.28	28.82	54.49
Carico	1.00	2.48	6.09	7.82	10.53	17.39

Table 5.4: The distribution of a servers response latency when different schedulers attempt to allocate a 1000 Pod Job across the server.

Table **??** contains the measured distribution of the response latency from the server. It shows how scheduling with the default scheduler using 100m CPU requests, resulting a significant shift in latency distribution. The median more than doubles and the distribution greatly shifted towards the tail: the maximum latency was $\approx \times 7$ larger. On the other hand, when scheduling with Carico, the median latency actually decreased. Furthermore, while the tail distribution did increase, the max latency was only $\approx \times 2$ bigger.

5.8 Carico Overhead

To measure the overhead incurred from running the Carico pods on the Nodes, I compared the completion time of Jobs when running on Nodes with and without the Carico deployment. I considered easuring resource utilisation on a Node with just the Carico Pod running, but much of the behaviour of the Carico Pod occurs during container events. As a result, the measured overhead would not represent the entire impact of the Carico Pod. Instead, by measuring the overhead over Jobs, we aggregate the impact of Carico across multiple container events, providing a more holistic view of its impact.

Number of Pods	**% Overhead with Carico**
100	-2.33 ± 3.29
250	-1.19 ± 2.44
500	4.84 ± 1.25
750	1.69 ± 0.42
1000	2.27 ± 0.66

Table 5.5: The overhead incurred when running Carico Pods on Nodes during the executing of Jobs with varying Pod counts. Each Pod executed `bpi(2000)` and requested 200 milliseconds of CPU time.

To measure Carico's overhead, I used Pod's executing `bpi(2000)`. Table **??** presents the relative change in Job Completion time with Carico Pods running on the Nodes. We can see that the Job Completion time of smaller Jobs are more noisy, therefore, resulting in an observed decrease in Job Completion times. However, with larger and more stable Jobs, the overhead from Carico is more visible. From these observations we can conclude that Carico has $\approx 2\%$ overhead.

5.9 Limitations

Firstly, Carico is a telemetry-based scheduler and thus relies on accurate telemetry that is able to accurately capture a Node's true limit, after which over-contention results in degraded performance. During this dissertation, we have observed how this is not guaranteed. VMs, like those I used in my evaluation, only have access to a simulated hardware provided by the underlying hypervisor. As a result, symptoms of over-contention, such as the overhead from CPU or cache thrashing, are not observed when a VM reports high CPU utilisation. This supported by Figure **??**: the Job Completion time per Pod when executing a Job on a single Node, continued to decrease even after the Node reported full CPU pressure. Consequently, Carico allocates fewer Pods to Nodes reducing Job completion.

Secondly, Carico can only learn about Pod workloads once they have already been scheduled and observed. This can limit performance, as Carico is unable to discern new Pods with potentially complementary directions of resource usage from those it has already observed. Conversely, a high-intensity workload may be submitted after a low-intensity one, and Carico's recent low Per-Pod-Cost estimate could result in a over-eager Nodes accepting too many high-intensity workloads before Carico has time to detect the new resource usage. This situation could then cause performance degradation or even the termination of Pods. Carico attempts to mitigate this by having the Per-Pod-Cost slowly increase when there are no Pods running, however, this has a limited impact for workloads that arrive in quick succession.

Chapter 6

Conclusion

6.1 Summary

This dissertation set out to explore the application of Pronto's underlying theory to a Kubernetes scheduler. Pronto's ability to efficiently aggregate and process telemetry produced across a datacenter addressed the limitations of telemtry-focused schedulers which aimed to reduce the reliance on user-provided resource requests. However, the dynamic nature of Kubernetes necessitated a novel scorable and reservable signal.

The project successfully achieved its objectives by proposing Carico, a federated, asynchronous, and memory-limited scoring algorithm that explicitly accounts for communication latency, a key challenge in real-world Kubernetes environments. A significant contribution was the novel application of Federated Singular Value Decomposition (FSVD) on non-mean-centered data to build a local model of recent resource usage, providing a unique interpretation of resource utilisation direction and magnitude. This led to the development of a continuous, comparable, and reservable capacity signal, expressed in terms of Pod units, which allows for robust reservation mechanisms and dynamic adaptation to changing workloads.

The comprehensive evaluation of the Carico prototype within a Kubernetes cluster yielded several key findings. Carico demonstrated comparable Job Completion times to the default kube-scheduler, crucially without requiring explicit Pod resource requests. More significantly, it consistently achieved substantially lower Pod Completion times and improved workload isolation, highlighting its effectiveness as a Quality of Service (QoS) scheduler. The evaluation also confirmed CARICO's low overhead, estimated at approximately 2%, making it a practical solution for real-world deployments. These results, combined with Carico portable nature, highlight its potential to significantly enhance scheduling efficiency and resource utilisation in Kubernetes and other orchestration systems by providing a more adaptive and performance-aware approach.

6.2 Future Work

The insights gained from Carico's development and evaluation open several promising avenues for future research. One significant direction involves exploring Carico's applicability in other scheduling environments where explicit resource requests are either impractical or unavailable. The core design principles of Carico, particularly its federated, asynchronous, and memory-limited nature, coupled with its ability to derive meaningful capacity signals from telemetry, are not intrinsically tied to Kubernetes. This inherent simplicity suggests that Carico could be adapted for diverse scheduling challenges beyond container orchestration.

Further research could also focus on expanding the range of metrics Carico utilizes for its capacity signal generation. Currently, Carico primarily uses CPU and memory metrics. Incorporating additional telemetry data, such as network I/O, disk I/O, or GPU utilization, could potentially provide a more holistic and accurate representation of a Node's workload capacity, leading to even more optimized scheduling decisions.

Finally, an important area of exploration is to evaluate Carico's performance on physical machines rather than virtualized environments. As noted in the evaluation, hypervisors can mask certain performance degradation effects, making CPU utilization alone an unreliable indicator of true contention. Testing Carico on bare-metal servers could reveal how its telemetric-only approach performs when telemetry data is more directly representative of underlying hardware contention and actual performance degradation. This would provide valuable insights into Carico's robustness and effectiveness in diverse infrastructure settings.

Appendix A

Required Lemmas for Model Interpretation

A.1 Proof: The First Left Singular Vector as a Pseudo Weighted Average

Let $\mathbf{A}$ be an $m \times n$ matrix with non-negative elements $0 \leq a_{ij} \leq 1$. Let u_1 be the first left singular vector and σ_1 the corresponding singular value of $\mathbf{A}$.

A.1.1 Fundamental Representation of u_1

The vector u_1 is an eigenvector of $\mathbf{A}\mathbf{A}^T$ associated with the largest eigenvalue σ_1^2: $\mathbf{A}\mathbf{A}^T = \sigma_1^2 u_1$. Let a_j denote the j-th column of $\mathbf{A}$. Then $\mathbf{A}\mathbf{A}^T = \sum_{j=1}^n a_j a_j^T$. Substituting this into the eigenvalue equation and assuming $\sigma_1 \neq 0$: $\left(\sum_{j=1}^n a_j a_j^T\right) u_1 = \sigma_1^2 u_1 \implies \sum_{j=1}^n a_j (a_j^T u_1) = \sigma_1^2 u_1$. Thus u_1 can be expressed as a linear combination of the columns of $\mathbf{A}$:

$$u_1 = \sum_{j=1}^n \left(\frac{a_j^T u_1}{\sigma_1^2}\right) a_j = \sum_{j=1}^n w_j a_j, \text{ where } w_j = \frac{a_j^T u_1}{\sigma_1^2}$$

A.1.2 Contribution of Columns to u_1

The magnitude of each weight w_j is given by:

$$|w_j| = \frac{|a_j^T u_1|}{\sigma_1^2} = \frac{\parallel a_j \parallel |\cos(\theta_j)|}{\sigma_1^2}$$

where θ_j is the angle between column a_j and u_1. Columns a_j with larger norms $\parallel a_j \parallel$ or those more parallel to u_1 (i.e. $|\cos(\theta_j)| \approx 1$) will have weights w_j of greater magnitude. Consequently, these "larger" or more aligned columns contribute more significantly to the sum defining u_1 and thus to its direction

A.1.3 Non-Uniqueness of u_1 and Relation to Perron-Frobenius Theorem

The singular vectors in SVD are not unique: if $\mathbf{A} = \mathbf{U}_1\Sigma\mathbf{V}_1^T = \mathbf{U}_2\Sigma\mathbf{V}_2^T$ then $\Sigma_1 = \Sigma_2$ but $\mathbf{U}_1 = \mathbf{U}_2\mathbf{B}_a$ and $\mathbf{V}_1 = \mathbf{V}_2\mathbf{B}_b$ for some block diagonal unitary matrices $\mathbf{B}_a, \mathbf{B}_b$ [**?**].

Since $\mathbf{A}$ has non-negative entries ($a_{ij} \geq 0$), $\mathbf{AA}^T$ is a non-negative matrix. By the Perron-Frobenius theorem, there exists an eigenvector u_1^+ corresponding to the eigenvalue σ_1^2 whose components u_{1i}^+ are all non-negative ($u_{1i}^+ \geq 0$). The first left singular vector u_1 obtained from SVD must then be $u_1 = bu_1^+$ where $b = \pm 1$

A.1.4 Interpretation as a Pseudo Weighted Average

- **Case 1:** $b = 1 \implies u_1 = u_1^+$.
  In this case, $a_j^T u_1 = a_j^T u_1^+ = \sum_i a_{ij}u_{1i}^+ \geq 0$ (as $a_{ij} \geq 0, u_{1i}^+ \geq 0$).
  The weights $w_j = (a_j^T u_1)/\sigma_1^2$ are therefore non-negative ($w_j \geq 0$).
  $u_1 = \sum w_j a_j$ becomes a conic combination of the non-negative column vectors a_j. This u_1 (itself non-negative) acts as a pseudo weighted average, representing a principal direction within the cone spanned by the a_j. Columns contributing larger non-negative w_j pull u_1 more strongly in their direction.

- **Case 2:** $b = -1 \implies u_1 = -u_1^+$.
  Here, $a_j^T u_1 = a_j^T(-u_1^+) = -a_j^T(u_1^+) \leq 0$.
  The weights $w_j = (a_j^T u_1)/\sigma_1^2$ are non-positive ($w_j \leq 0$).
  While $u_1 = \sum w_j a_j$ now involves non-positive weights for non-negative vectors a_j, the magnitudes $|w_j| = (a_j^T u_1)/\sigma_1^2$ remain the same as in Case 1. Thus, columns a_j that are "larger" or more aligned with u_1^+ still contribute with greater magnitude to the sum, determining the orientation of the line spanned by u.

A.1.5 Conclusion

The vector u_1 is a linear combination of the columns of $\mathbf{A}$, $u_1 = \sum w_j a_j$. The magnitude of the coefficient w_j for each column a_j is proportional to the projection of a_j onto u_1 (scaled by $1/\sigma_1^2$). This means the columns with larger norms or those more aligned with the principal direction (the line spanned by u_1) contribute more significantly to defining this direction.

Irrespective of the sign b (determined by the SVD algorithm), the line along which u_1 lies is shaped by this weighted aggregation.

A.2 Proof: Monotonicity of the First Singular Value for Non-Negative Matrices

Let $\mathbf{A}$ and $\mathbf{B}$ be $m \times n$ matrices with real entries such that $0 \leq a_{ij} \leq b_{ij} \leq 1$ for all i, j. We want to show that $\sigma_1(\mathbf{A}) \leq \sigma_1(\mathbf{B})$, where $\sigma_1(\mathbf{M})$ denotes the first (largest) singular value of a matrix $\mathbf{M}$.

A.2.1 Relating Singular Values to $\mathbf{M}^T\mathbf{M}$

The square of the first singular value, $\sigma_1(\mathbf{M})^2$, is the largest eigenvalue of the matrix $\mathbf{M}^T\mathbf{M}$. Since $\mathbf{M}^T\mathbf{M}$ is a symmetric positive semi-definite matrix, its largest eigenvalue is also its spectral radius, $\rho(\mathbf{M}^T\mathbf{M})$. Thus, $\rho(\mathbf{A})^2 = \rho(\mathbf{A}^T\mathbf{A})$ and $\rho(\mathbf{B})^2 = \rho(\mathbf{B}^T\mathbf{B})$.

A.2.2 Comparing $\mathbf{A}^T\mathbf{A}$ and $\mathbf{B}^T\mathbf{B}$

Let $\mathbf{M_A} = \mathbf{A}^T\mathbf{A}$ and $\mathbf{M_B} = \mathbf{B}^T\mathbf{B}$. The (j, k)-th element of thess matrices are:

$$(\mathbf{M_A})_{jk} = \sum_{i=1}^{m} a_{ij}a_{ik} \tag{A.1}$$

$$(\mathbf{M_B})_{jk} = \sum_{i=1}^{m} b_{ij}b_{ik} \tag{A.2}$$

Since $0 \leq a_{ij} \leq b_{ij}$ for all i, j:

- All a_{ij} and b_{ij} are non-negative
- Therefore, $a_{ij}a_{ik} \leq b_{ij}b_{ik}$ for all i, j, k. Summing over i:

  $$\sum_{i=1}^{m} a_{ij}a_{ik} \leq \sum_{i=1}^{m} b_{ij}b_{ik}$$

  This implies $(\mathbf{M_A})_{jk} \leq (\mathbf{M_B})_{jk}$ for all j, k. Thus, $0 \leq \mathbf{M_A} \leq \mathbf{M_B}$ element-wise. Both $\mathbf{M_A}$ and $\mathbf{M_B}$ are matrices with non-negative entries.

A.2.3 Monotonicity of Spectral Radius

A standard result from Perron-Frobenius theory states that if $\mathbf{X}$ and $\mathbf{Y}$ are non-negative matrices such that $\mathbf{X} \leq \mathbf{Y}$ element-wise, then their spectral radii satisfy $\rho(\mathbf{X}) \leq \rho(\mathbf{Y})$. Applying this result to $\mathbf{M_A} = \mathbf{A}^T\mathbf{A}$ and $\mathbf{M_B} = \mathbf{B}^T\mathbf{B}$:

$$\rho(\mathbf{A}^T\mathbf{A}) \leq \rho(\mathbf{B}^T\mathbf{B})$$

A.2.4 Conclusion

From Section **??** and **??**:

$$\sigma_1(\mathbf{A})^2 \leq \sigma_1(\mathbf{B})^2$$

Since the singular values are by definition non-negative ($\sigma_1(\mathbf{M}) \geq 0$):

$$\sigma_1(\mathbf{A}) \leq \sigma_1(\mathbf{B})$$

This shows that if the values of a non-negative matrix **A** are increased (while remaining non-negative, e.g., elements between 0 and 1), the first singular value of the resulting matrix **B** wil greater than or equal to that of **A**.

A.3 Proof: Singular Value of Concatenated Matrix

This section proves the following statement:
Given two non-negative matrices $\mathbf{A} \in \mathbb{R}^{m \times n_1}$ and $\mathbf{B} \in \mathbb{R}^{m \times n_2}$, the first singular value of the concatenated matrix $\mathbf{C} = [\mathbf{A}, \mathbf{B}]$ is greater than or equal to the maximum of the first singular values of **A** and **B**. That is,

$$\sigma_1([\mathbf{A}, \mathbf{B}]) \geq \max(\sigma_1(\mathbf{A}), \sigma_1(\mathbf{B}))$$

A.3.1 Setup

Let **A** be an $m \times n_1$ matrix and **B** be an $m \times n_2$ matrix. The concatenated matrix $\mathbf{C} = [\mathbf{A}, \mathbf{B}]$ is an $m \times (n_1 + n_2)$ matrix. The first singular value of any matrix **M**, denoted $\sigma_1(\mathbf{M})$, is defined as its spectral norm:

$$\sigma_1(\mathbf{M}) = \| \mathbf{M} \|_2 = \max_{\|x\|_2 = 1} \| \mathbf{M}x \|_2$$

where x is a unit vector.

A.3.2 Relating $\sigma_1(\mathbf{C})$ to $\sigma_1(\mathbf{A})$

Let $v^*_{\mathbf{A}}$ be a unit vector in $\mathbb{R}^{n_1}$ such that $\sigma_1(\mathbf{A}) = \| \mathbf{A}v^*_{\mathbf{A}} \|_2$. Such a vector $v^*_{\mathbf{A}}$ is the right singular vector corresponding to $\sigma_1(\mathbf{A})$.

Consider a vector $x_0 \in \mathbb{R}^{n_1+n_2}$ constructs as $x_0 = \begin{bmatrix} v^*_{\mathbf{A}} \\ 0_{n_2 \times 1} \end{bmatrix}$, where $0_{n_2 \times 1}$ is the zero vector of dimensions n_2.
The squared norm of x_0 is $\| x_0 \|_2^2 = \| v^*_{\mathbf{A}} \|_2^2 + \| 0_{n_2 \times 1} \|_2^2 = 1^2 + 0 = 1$. Thus, x_0 is a unit vector.

By definitions of $\sigma_1(\mathbf{C})$:

$$\sigma_1(\mathbf{C}) = \max_{\|x\|_2=1} \| \mathbf{C}x \|_2$$

Since x_0 is a specific unit vector, we have:

$$\sigma_1(\mathbf{C}) \geq \| \mathbf{C}x_0 \|_2$$

Computing $\mathbf{C}x_0$ gives:

$$\mathbf{C}x_0 = [\mathbf{A}, \mathbf{B}] \begin{bmatrix} v^*_{\mathbf{A}} \\ 0_{n_2 \times 1} \end{bmatrix} = \mathbf{A}v^*_{\mathbf{A}} + \mathbf{B}\dot{0}_{n_2\times 1} = \mathbf{A}v^*_{\mathbf{A}}$$

Therefore,

$$\sigma_1(\mathbf{C}) \geq \| \mathbf{A}v^*_{\mathbf{A}} \|_2 = \sigma_1(\mathbf{A})$$

A.3.3 Relating $\sigma_1(\mathbf{C})$ to $\sigma_1(\mathbf{B})$

Similarly, let $v^*_{\mathbf{B}}$ be a unit vector in $\mathbb{R}^{n_2}$ such that $\sigma_1(\mathbf{B}) = \| \mathbf{B}v^*_{\mathbf{B}} \|_2$. Such

Consider a vector $x_1 \in \mathbb{R}^{n_1+n_2}$ constructs as $x_1 = \begin{bmatrix} 0_{n_1\times 1} \\ v^*_{\mathbf{B}} \end{bmatrix}$.
The squared norm of x_1 is $\| x_1 \|_2^2 = \| 0_{n_1\times 1} \|_2^2 + \| v^*_{\mathbf{B}} \|_2^2 = 0 + 1^2 = 1$. Thus, x_1 is a unit vector.

Again, by definitions of $\sigma_1(\mathbf{C})$:

$$\sigma_1(\mathbf{C}) \geq \| \mathbf{C}x_1 \|_2$$

Computing $\mathbf{C}x_1$ gives:

$$\mathbf{C}x_1 = [\mathbf{A}, \mathbf{B}] \begin{bmatrix} 0_{n_1 \times 1} \\ v^*_{\mathbf{B}} \end{bmatrix} = \mathbf{A}\dot{0}_{n_1\times 1} + \mathbf{B}v^*_{\mathbf{B}} = \mathbf{B}v^*_{\mathbf{B}}$$

Therefore,

$$\sigma_1(\mathbf{C}) \geq \| \mathbf{B}v^*_{\mathbf{B}} \|_2 = \sigma_1(\mathbf{B})$$

A.3.4 Conclusion

From Sections **??** and **??**, we have shown that $\sigma_1(\mathbf{C}) \geq \sigma_1(\mathbf{A})$ and $\sigma_1(\mathbf{C}) \geq \sigma_1(\mathbf{B})$. This implies that $\sigma_1(\mathbf{C})$ must be greater than or equal to the maximum of these two values:

$$\sigma_1([\mathbf{A}, \mathbf{B}]) \geq \max(\sigma_1(\mathbf{A}), \sigma_1(\mathbf{B}))$$

A.4 σ_1 of Scaled Concatenation

This section proves the following statement: Given two non-negative matrices $\mathbf{A} \in \mathbb{R}^{m\times n_1}$ and $\mathbf{B} \in \mathbb{R}^{m\times n_2}$ and non-negative scalar weights $\gamma_\mathbf{A} = \sqrt{w_\mathbf{A}}$ and $\gamma_\mathbf{B} = \sqrt{w_\mathbf{B}}$ such that $w_\mathbf{A} + w_\mathbf{B} = 1$, the resulting first singular value and the first left singular vector of the concatenated matrix $\mathbf{C} = [\gamma_\mathbf{A}\mathbf{A}, \gamma_\mathbf{B}\mathbf{B}]$ has the following property:

$$\min(P_\mathbf{A}(u_\mathbf{C}), P_\mathbf{B}(u_\mathbf{C})) \leq \sigma_\mathbf{C}^2 \leq \max(P_\mathbf{A}(u_\mathbf{C}), P_\mathbf{B}(u_\mathbf{C}))$$

where $\sigma_\mathbf{C}$ and $u_\mathbf{C}$ correspond to the first singular value and first left singular vector of $\mathbf{C}$, $P_\mathbf{M}(u)$ is the sum of squared scalar porjections of the columns in $\mathbf{M}$ onto u (let m_j be the j-th column of $\mathbf{M}$ $P_\mathbf{M}(u) = \sum_{j=1}^{n}(u^T m_j)^2$).

A.4.1 Expanding $\sigma_1(\mathbf{C})$

By definition, $S_\mathbf{C} = \sigma_1(\mathbf{C})$ and $u_\mathbf{C}$ is the corresponding first singular left vector. Thus, they satisfy the eigenvalue equation for $\mathbf{C}\mathbf{C}^T$:

$$\mathbf{C}\mathbf{C}^T u_\mathbf{C} = S_\mathbf{C}^2 u_\mathbf{C}$$

Pre-multiplying by $u_\mathbf{C}^T$:

$$u_\mathbf{C}^T(\mathbf{C}\mathbf{C}^T u_\mathbf{C}) = u_\mathbf{C}^T(S_\mathbf{C}^2 u_\mathbf{C}) \tag{A.3}$$

$$u_\mathbf{C}^T\mathbf{C}\mathbf{C}^T u_\mathbf{C} = S_\mathbf{C}^2(u_\mathbf{C}^T u_\mathbf{C}) \tag{A.4}$$

Since $u_\mathbf{C}$ is a unit vector, $u_\mathbf{C}^T u_\mathbf{C} = \| u_\mathbf{C} \|_2^2 = 1$. So,

$$S_\mathbf{C}^2 = u_\mathbf{C}^T\mathbf{C}\mathbf{C}^T u_\mathbf{C}$$

Now, lets express $\mathbf{C}\mathbf{C}^T$ in terms of $\mathbf{A}$ and $\mathbf{B}$:
The matrix $\mathbf{C} = [\gamma_\mathbf{A}\mathbf{A}, \gamma_\mathbf{B}\mathbf{B}]$.
Then $\mathbf{C}^T = \begin{bmatrix} \gamma_\mathbf{A}\mathbf{A}^T \\ \gamma_\mathbf{B}\mathbf{B}^T \end{bmatrix}$.
So, $\mathbf{C}\mathbf{C}^T = [\gamma_\mathbf{A}\mathbf{A}, \gamma_\mathbf{B}\mathbf{B}] \begin{bmatrix} \gamma_\mathbf{A}\mathbf{A}^T \\ \gamma_\mathbf{B}\mathbf{B}^T \end{bmatrix} = (\gamma_\mathbf{A}\mathbf{A})(\gamma_\mathbf{A}\mathbf{A}^T) + (\gamma_\mathbf{B}\mathbf{B})(\gamma_\mathbf{B}\mathbf{B}^T)$

$$\mathbf{C}\mathbf{C}^T = \gamma_\mathbf{A}^2\mathbf{A}\mathbf{A}^T + \gamma_\mathbf{B}^2\mathbf{B}\mathbf{B}^T$$

Substitute this expression for $\mathbf{C}\mathbf{C}^T$ into the equation for $S_\mathbf{C}^2$:

$$S_\mathbf{C}^2 = u_\mathbf{C}^T(\gamma_\mathbf{A}^2\mathbf{A}\mathbf{A}^T + \gamma_\mathbf{B}^2\mathbf{B}\mathbf{B}^T)u_\mathbf{C} \tag{A.5}$$

$$S_\mathbf{C}^2 = \gamma_\mathbf{A}^2(u_\mathbf{C}^T\mathbf{A}\mathbf{A}^T u_\mathbf{C}) + \gamma_\mathbf{B}^2(u_\mathbf{C}^T\mathbf{A}\mathbf{A}^T u_\mathbf{C}) \tag{A.6}$$

Using the definitions for $P_{\mathbf{A}}(u_{\mathbf{C}})$ and $P_{\mathbf{B}}(u_{\mathbf{C}})$:

- $P_{\mathbf{A}}(u_{\mathbf{C}}) = u_{\mathbf{C}}^T \mathbf{A}\mathbf{A}^T u_{\mathbf{C}}$ (sum of squared projections of columns of $\mathbf{A}$ onto $u_{\mathbf{C}}$
- $P_{\mathbf{B}}(u_{\mathbf{C}}) = u_{\mathbf{C}}^T \mathbf{B}\mathbf{B}^T u_{\mathbf{C}}$ (sum of squared projections of columns of $\mathbf{B}$ onto $u_{\mathbf{C}}$

This means the equation becomes:

$$S_{\mathbf{C}}^2 = \gamma_{\mathbf{A}}^2 P_{\mathbf{A}}(u_{\mathbf{C}}) + \gamma_{\mathbf{B}}^2 P_{\mathbf{B}}(u_{\mathbf{C}})$$

We are given weights $\gamma_{\mathbf{A}}, \gamma_{\mathbf{B}}$ such that $\gamma_{\mathbf{A}}^2 \geq 0, \gamma_{\mathbf{B}}^2 \geq 0$, and $\gamma_{\mathbf{A}}^2 + \gamma_{\mathbf{B}}^2 = 1$. This means that $S_{\mathbf{C}}^2$ is a convex combination of the two values $P_{\mathbf{A}}(u_{\mathbf{C}})$ and $P_{\mathbf{B}}(u_{\mathbf{C}})$. A fundamental property of convex combinations is that they always lie between the minimum and maximum of the values being combined.
Let $x = P_{\mathbf{A}}(u_{\mathbf{C}})$ and $y = P_{\mathbf{B}}(u_{\mathbf{C}})$. Then $S_{\mathbf{C}}^2 = \gamma_{\mathbf{A}}^2 x + \gamma_{\mathbf{B}}^2 y$.
If $x \leq y$:

- $\min(x, y) = x = (\gamma_{\mathbf{A}}^2 + \gamma_{\mathbf{B}}^2)x = \gamma_{\mathbf{A}}^2 x + \gamma_{\mathbf{B}}^2 x \leq \gamma_{\mathbf{A}}^2 x + \gamma_{\mathbf{B}}^2 y = S_{\mathbf{C}}^2$ (since $\gamma_{\mathbf{B}}^2 \geq 0$ and $x \leq y$).
- $\max(x, y) = y = (\gamma_{\mathbf{A}}^2 + \gamma_{\mathbf{B}}^2)y = \gamma_{\mathbf{A}}^2 y + \gamma_{\mathbf{B}}^2 y \geq \gamma_{\mathbf{A}}^2 x + \gamma_{\mathbf{B}}^2 y = S_{\mathbf{C}}^2$ (since $\gamma_{\mathbf{A}}^2 \geq 0$ and $x \leq y$).

A similar argument holds if $y \leq x$. Therefore:

$$\min(P_{\mathbf{A}}(u_{\mathbf{C}}), P_{\mathbf{B}}(u_{\mathbf{C}})) \leq \sigma_{\mathbf{C}}^2 \leq \max(P_{\mathbf{A}}(u_{\mathbf{C}}), P_{\mathbf{B}}(u_{\mathbf{C}}))$$