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: 60

Main chapters (excluding front-matter, references and appendix): 50 pages (pp 9–58)

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 an existing 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 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 (961)

Kubernetes has established itself as the leading platform for container orchestration, experiencing widespread adoption across diverse industries and organizations of all scales [1]. From traditional stateless microservices that power modern web applications and APIs, to complex stateful databases requiring persistent storage, Kubernetes provides a robust and flexible foundation. It seamlessly orchestrates high-throughput batch processing jobs, streamlines continuous integration and delivery (CI/CD) pipelines, and has become the infrastructure backbone for cloud computing services like Google Cloud and Azure [2, 3]. Furthermore, its versatility extends to supporting machine learning (ML) deployments [4], edge computing [5], and even serverless functions [6, 7]. This broad applicability solidifies Kubernetes' position as a cornerstone of the cloud-native industry.

A core function of container orchestration is the scheduling of containers across a cluster. Task scheduling is classified as an NP-hard problem and consists of organising, prioritising and allocating tasks to a set of resources in an optimal manner. Kubernetes performs online scheduling, a more challenging subset of scheduling [8] where schedulers receive tasks over time, and must schedule the tasks without any knowledge of the future. Without entirely knowing of all tasks to come, the scheduler can't guarantee optimal schedules. Much research has been focused on finding efficient scheduling algorithms that guarantee solutions as close to optimal as possible. The standard Kubernetes approach assumes a set of pre-determined Node capacities and resource requirements, translating the problem of efficient scheduling into one of bin-packing [].

In the dynamic and resource-intensive landscape of containerized applications, efficient online scheduling is paramount for both performance and cost-effectiveness. Poor scheduling decisions can lead to significant infrastructure waste, directly impacting an organisation's bottom line. Misconfiguration in resource requests and limits are a primary cause of under-utilised resources or idle nodes, and thus higher billing [9, 10].

TODO: SHOULD I MOVE THE PARAGRAPH ABOVE TO THE TOP TO HAVE A MORE IMPACTFUL START

The available Kubernetes schedulers can be divided based on the information they use to inform their scheduling decisions: pod-description based schedulers, which use pod annotations to declare resource requests, resource limits and QoS classes, and telemetry-based schedulers, which periodically collected live information from each node. As previously mentioned, pod-description based schedulers rely on accurate resource usage estimations. Telemetry-based schedulers can circumvent this problem, collecting information about the true resource usage of a Node to then influence future scheduling decisions. However, to the best of my knowledge, all these telemetric schedulers only aim to improve the estimations used by the existing default `kube-scheduler`.

The concept of a Kubernetes scheduler operating without predefined pod resource requests remains largely unexplored due to its inherent complexity. However, this avenue warrants further investigation. We cannot always guarantee accurate workload estimations beforehand, rendering traditional bin-packing approaches "best-effort" at best. A purely telemetry-driven scheduler could potentially overcome this limitation by adapting to true resource consumption, even for unpredictable workloads.

PRONTO [] offers a promising direction. This novel federated, asynchronous, memory-limited algorithm schedules tasks across large-scale networks of hundreds of workers. Each individual worker updates their local model based on the workload seen so far which is then periodically aggregating the local models builds a global view of the system. 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, its assumptions of no communication latency, and initial empirical findings when investigating metrics indicated that a direct application would not be feasible.

Instead, I devised a new algorithm, CARICO (Italian for "load"), that is still federated, asynchronous and memory-limited like PRONTO, but accounts for communication latency. Nodes perform FSVD on capacity-based metrics, rather than FPCA, modeling past resource usage to estimate future workload. Nodes can then use their current resource usage and estimated workload to produce a measure of capacity which can be used to score Nodes. Furthermore, when combined with simple signal processing, this signal can be transformed to represent the number of Pods a Node is willing to accept, allowing schedulers to handle communication latencies by reserving portions of the signal with each scheduling decision.

In contrast, I developed CARICO (Italian for "load"), a novel algorithm that maintains the federated, asynchronous, and memory-limited properties of PRONTO while explicitly incorporating communication latency considerations. Nodes perform Federated Singular Value Decomposition (FSVD) on capacity-based metrics, rather than Federated Principal Component Analysis (FPCA), to model historical resource utilisation and forecast future

workload. Based on their current resource usage and estimated workload, Nodes compute a dynamic capacity metric which serves as a basis for Node scoring. Moreover, through the application of simple signal processing, this capacity signal can be transformed to quantify the number of Pods a Node is prepared to accept, thereby allowing schedulers to account for communication latencies by reserving portions of the signal with each scheduling decision.

For the evaluation, I implement CARICO within the Kubernetes ecosystem and run it within a Kubernetes Cluster. This required extensive investigations into various sources of telemetry, metrics and signal processing with the goal of generating an an accurate and precise capacity signal. I evaluate CARICO's overall performance when scheduling various workloads on a Kubernetes cluster, comparing it to the industry-standard default `kube-scheduler`. I show that while CARICO achieves comparable throughput, it signif-icantl outperforms as a QoS scheduler, demonstrating lower Pod Completion times and improved workload isolation.

To sum up, this project makes the following contributions:

- Investigate the feasibility of applying PRONTO to Kubernetes scheduling
- Propose a novel scoring algorithm, CARICO, with the same properties as PRONTO, while also accounting for possible communication latency.
  - Novel application of FSVD to build a local model of recent resource usage. Adapted the standard SVD subspace merge operation to account for a lack of mean-centering.
  - Present a novel signal that scores a Node's predicted capacity considering recent workloads experienced across the cluster.
  - Apply simple signal processing techniques to transform the generated capacity signal into a signal that can be reserved.
- Implement a prototype of CARICO to evaluate the algorithm when applied to a real-world cluster and workloads.
  - Investigate different metrics to use to build the local model, elucidating the problems of using metrics advertised by Virtual Machines (VMs). Explore different signal processing techniques to reduce noise generated by the container runtime.
  - Ensure correctness of the generated signal and investigate different approaches to estimating the baseline signal value and the signal cost associated with a single Pod.
  - Investigate the overall performance of CARICO under different workloads. I focus on throughput, QoS, and performance isolation.

I Chapter 2, I go over existing Kubernetes schedulers, highlighting how only Pod description-based schedulers or telemetry-based schedulers which aim to optimise decisions of already running Pod description-based schedulers exist. I conduct a brief investigation into the feasibility of applying Pronto to Kubernetes, highlighting flawed assumptions and presenting empirical evidence. In Chapter 3, I propose the novel algorithm, CARICO, using proofs and reasoning to explain the generated capacity signal. In Chapter 4, I outline the structure of the CARICO system in Kubernetes, exploring numerous metrics, signal processing techniques with the goal of generating an accurate and precise signal. In Chapter 5, I evaluate the performance of CARICO under different workloads, collecting numerous performance metrics and comparing it against the standard `kube-scheduler`. Lastly, I conclude my work and propose potential future directions.

Chapter 2

Background

In this Chapter, I explain how scheduling works in Kubernetes. I will then explain the concepts behind PRONTO, as well as the reasons for exploring PRONTO's application in Kubernetes. I will also describe related Kubernetes schedulers. Finally, I will discuss the effectiveness of applying PRONTO's algorithm within a Kubernetes system.

2.1 Kubernetes

2.1.1 Kubernetes Overview

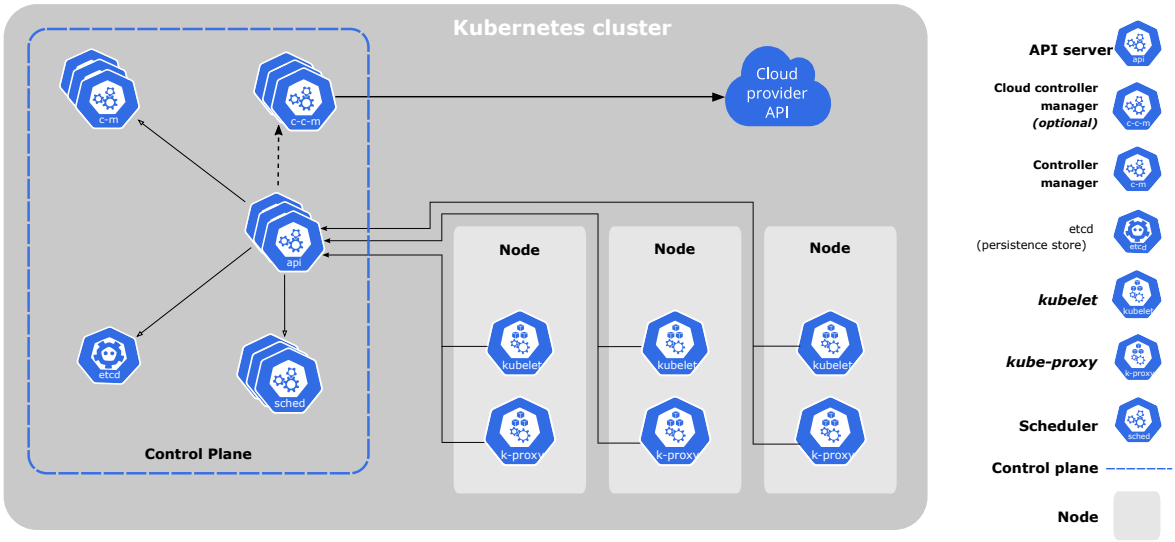


Figure 2.1: The components of a Kubernetes cluster [11]

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. The `kube-apiserver` exposes the Kubernetes HTTP API, which is used to publish objects such as Deployments,

DaemonSets and Jobs. 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 Kubernetes API is used to create, modify or delete these Kubernetes objects. Almost every Kubernetes object includes two fields: `spec` and `status`. `spec` is used on creation as a description of the Objects desired state. You can define affinities and QoS classes within this field. 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. `status` describes the current state of the object, supplied and updated by the Kubernetes system. These fields are core to scheduling in Kubernetes.

2.1.2 Scheduling in Kubernetes

In Kubernetes, Pods are the smallest deployable units of computing that you can create and manage in Kubernetes. 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. Pods can be individually created with their own Yaml files. However, the Kubernetes API also provides workload objects to manage multiple pods: these objects 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. Example workloads include Deployment, StatefulSet, DaemonSet and Job.

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 occures, the Pod transitions from "Pending" to "Running".

2.2 Pronto

2.2.1 Principle Component Analysis

This section first explains Singular Value Decomposition (SVD) and how it relates to solutions of Principal Component Analysis (PCA). I then introduce Incremental-SVD and Subspace-Merge, which are used to perform FPCA on the stream of telemetry produced by each Node.

2.2.2 Singular Value Decomposition

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

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

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

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

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 eignenvector of $\mathbf{A}\mathbf{A}^T$ and $\mathbf{A}^T\mathbf{A}$.

2.2.3 Principal Component Analysis

Pricipal 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. The matrix is first mean-centered: $\mathbf{A}_{ij} = (\mathbf{B}_{ij} - \mu_i)$ 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 a matrix $\mathbf{A}$ is given as $\mathbf{A}\mathbf{A}^T$ the Principal Components (PCs) maximise the following equation:

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

Given $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T$ 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 orginal $\mathbf{B}$ matrix.

2.2.4 Subspace-Merge

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

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

$[\mathbf{A}, \mathbf{B}]$ signifies the concatenation of two matrices with the same number of rows.

2.2.5 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 $[\mathbf{Y}_1, \ldots, \mathbf{Y}_l] = \mathbf{Y}$, 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: $\mathbf{Y} = [\mathbf{Y}_1, \ldots, \mathbf{Y}_l]$
Result: $\mathbf{U}_l, \Sigma_l$ such that $\mathbf{Y} = \mathbf{U}\Sigma\mathbf{V}^T$

```
U_1, Σ_1, V_1^T = SVD(Y_1)
for i = 2 to l do do
    U_i, Σ_i, V_i^T = SVD([U_{i-1}Σ_{i-1}, Y_i])
end for
```

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

2.2.6 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 gradually reduces the influence of previous subspaces. Like with standard PCA, FPCA can be considered as FSVD on centered data.

2.2.7 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

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

We can use $\mathbf{U}^r = \begin{bmatrix} | & & | \\ u_1 & \dots & u_r \\ | & & | \end{bmatrix}$ 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.

2.2.8 Pronto System Overview

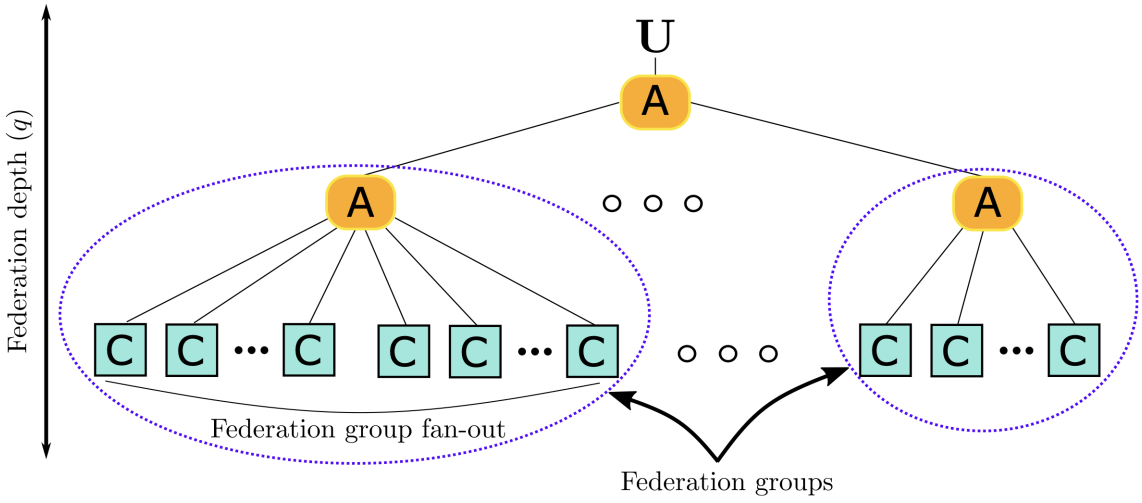


Figure 2.2: How local models are aggregated in PRONTO. Dedicated aggregator nodes propagate the updated subspaces until the root is reached [12].

There are two types of nodes in PRONTO: compute node (C) and aggregator node (A). Compute nodes collect and center node statistics (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 before Subspace-Merge on incoming subspaces, with subspace produced by the root aggregator node being propagated back to the compute nodes.

2.2.9 Pronto Reject-Job Signal

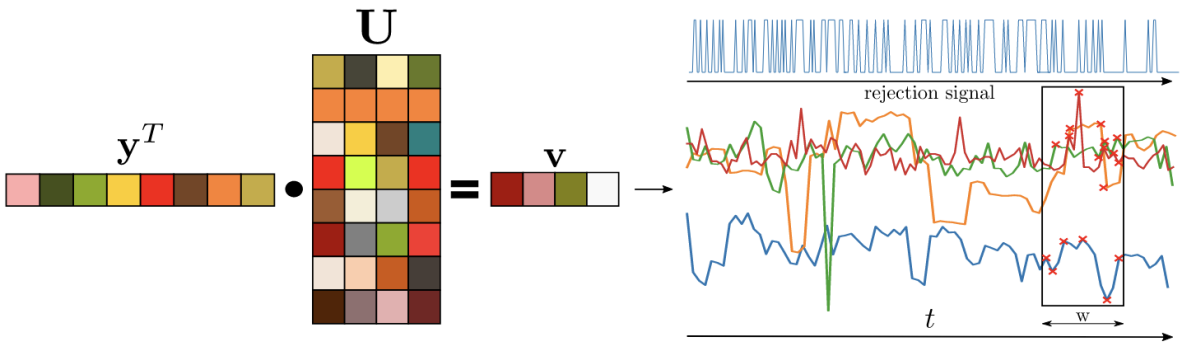


Figure 2.3: Projection of incoming $y \in \mathbb{R}^d$ onto embedding $U \in \mathbb{R}^{d \times r}$ producing R projections in $v \in \mathbb{R}^{1 \times r}$. Projections are tracked over time for detecting spikes which form the basis of the rejection signal. The sliding window for spike detection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

Each compute node projects their data onto the latest version of $\mathbf{U}$, and identifies all the spikes. If the weighed sum of these spikes, using the corresponding singular values in Σ, exceeds a threshold, a rejection signal is raised to indicate that the node is potentially experiencing performance degradation and a job should not be scheduled on that node.

2.2.10 Strengths

PRONTO's ability to accurately predicting spikes in traces from real-world datacenters was compared against non-distributed dimensionality-reduction methods [], PRONTO's improved performance over the non-distributed strategies implies that there were correlations between job's resource usages on different nodes at the same point in time. This suggested that FPCA could use telemetry across multiple nodes to more acurrately compute the principle components and thus could more accurately predict performance degradation. These results also indicated a potential benefit from applying PRONTO to a Kubernetes environment.

2.2.11 Weaknesses

Assumptions

The Pronto paper only provides a method with which to measure node "responsiveness" to future workloads. However, it does not include an allocation algorithm which then uses this scoring. Furthermore, a binary signal makes it difficult to score nodes against each other. Other schedulers typically use a scoring function to pick the "optimal" node [13][].

In addition, the paper makes multiple assumptions which don't hold in a Kubernetes

scheduler. Firstly, it assumes there is no communication latency within the system. This also implicitly assumes no binding latency: no latency between a node accepting a task and the task starting to run on the node. This is important as the spike prediction uses live telemetry data to predict spikes, and thus the score only considers currently running task. In Kubernetes, the latency between a pod being bound to a node and the pod actually running on the node is significant [14]. This introduces another challenge to directly applying Pronto to Kubernetes, as nodes may accept too many pods as the signal does not reflect the "in flight" pods.

Peak-Prediction

While CPU-Ready is specific to VMware vSphere, other contention metrics are available on a Linux-based system. The closest related metrics are the pressure stall information (PSI) metrics: pressure information for each resource is exported through their respective file in `/proc/pressure/<cpu|memory|io>`. Within each file, the metrics are broken down into two categories: some - which indicates the share of time in which at least some tasks are stalled on a given resource, and full which indicates the share of time in which all non-idle tasks are stalled on a given resource simultaneously.

Pronto performs peak-detection on the metrics to predict future spikes which then indicate high-resource contention and degraded performance. Therefore, for Pronto to make accurate predictions and produce a correct RejectJob signal, PSI must exhibit spikes during high-resource contention. To investigate the feasibility of peak prediction within a Kubernetes node using PSI metrics, I polled the `/proc/pressure` files of nodes when under different workloads.

From figure 2.4, we can see how even with lightweight workloads, the PSI metrics experience a spike in value. These spikes are due to the container runtime, in this case Containerd, using resource to create or delete the containers of the pods.

The PSI metrics also expose an average over the last 10 seconds. Figure 2.5 shows how the averaged value is able to reduce the contention spikes. However, it also reduces the responsiveness of the metrics - in the 10 pod batch, the metrics fails to rise to the same value as in the total case before the pods have completed. From this investigation, I concluded that peak-prediction with sub-second polling is not feasible within Kubernetes.

2.3 Related Work

In this section, I explore the existing Kubernetes scheduler space, identifying those that share properties with Pronto and highlighting their strengths and weaknesses.

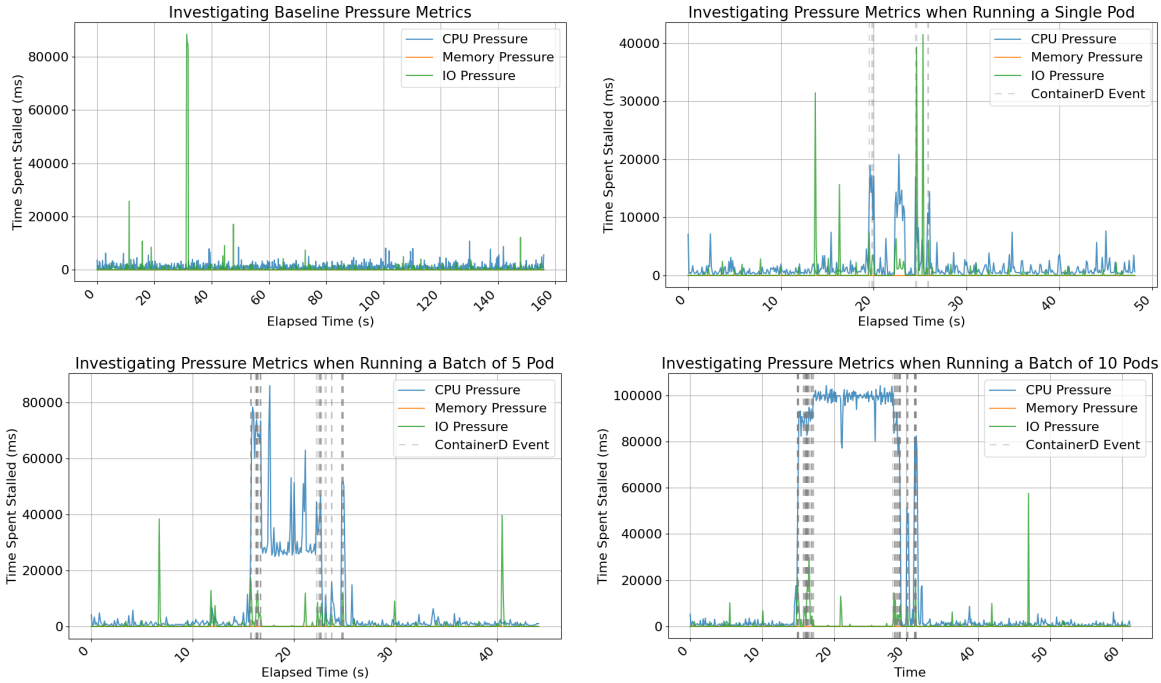


Figure 2.4: Measurements of `/proc/pressure/ total` value under different loads. The container runtime results in spikes no matter the workload.

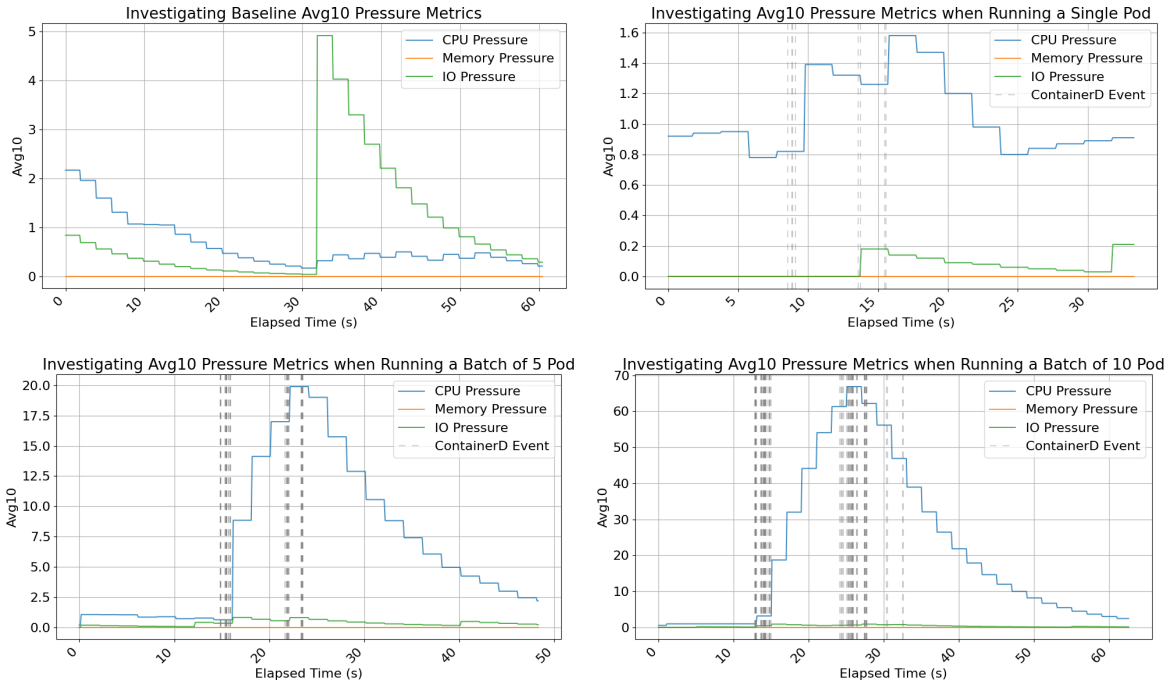


Figure 2.5: Measurements of `/proc/pressure/ avg10` value under different loads. The container runtime results in spikes no matter the workload.

2.3.1 Telemetric-based Schedulers

2.3.2 Federated Schedulers

2.3.3 Reinforcement Learning Schedulers

2.3.4 Summary of Related Work

The earlier survey compared how existing Kubernetes schedulers used two difference classes of input data - static pod information and telemetry - to guide their scheduling decisions. Pod description-based scheduling ensures that fundamental resource guarantees are respected. Telemetry-based scheduling can more precisely allocate pods to nodes at the cost of increased complexity. Hybrid schedulers combine these methods to provide guarantees of fundamental resource requests while fine tuning scheduling decisions. However, their reliance on predefined pod requirements still limits the QoS they can achieve. Without careful consideration and engineering, erroneous pod resource requests can result in under-utilisation or high resource contention.

2.4 Summary

This chapter introduced the core concepts of online scheduling and the Kubernetes environment. I explored existing Kubernetes schedulers, investigating the merits of using pod description-based schedulers vs. telemetry-based schedulers. From the brief survey, I identified that few Kubernetes schedulers use deeper contention indicators, such as cache pressure and memory pressure. A potential technique that could be applied to Kubernetes scheduling is Pronto: a novel scoring method that uses a federated approach with contention-based metric - allowing it to quickly predict performance degredation while catering for distributional shifts. While Pronto can't be directly applied to Kubernetes, its federated apsect and its use of contention-based metrics can inspire a new FL approach.

Chapter 3

Design

In this chapter I outline a novel federated, asynchronous, memory-limited algorithm similar to Pronto. However, rather than performing standard FPCA, I apply FSVD on non-centered metrics to build a model of experience resource usage. Combining this model with a new continuous signal function, nodes be scored by their capacity according to the expected workload and their current resource usage. Finally, we propose a method of capacity and cost prediction to be used by the schedulers reserve function. As this algorithm produces a signal that measures "capacity", I will refer to it as Spazio - "space" in Italian.

3.1 PCA using SVD

When applying PCA to a matrix B, the matrix is first centered by taking $A_{ij} = (B_{ij} - \mu_i)$ where μ_i is the mean of row i. The output of PCA is a set of rows $y_i^T = x_i^T A$ where $x_i, ..., x_m$ are orthonormal vectors that maximise the variance of the i^{th} row:

$$\text{Var}_i = \max_{\substack{x_i \in \mathbb{R}^m \setminus \{0\} \\ \|x_i\|=1 \\ x_i \perp x_1 ... x_{i-1}}} y_i^T y_i = \max_{\substack{x_i \in \mathbb{R}^m \setminus \{0\} \\ \|x_i\|=1 \\ x_i \perp x_1 ... x_{i-1}}} x_i^T \ \mathbf{AA} \ x_i$$

Given SVD($\mathbf{A}$) = $\mathbf{U}, \Sigma, \mathbf{V}^T$, it can be shown that $x_i = u_i$ from $\mathbf{U}$, and $\Sigma_i i = \sigma_i = \text{Var}_i$. This allows us to use iterative-SVD to efficiently perform PCA.

3.2 Local Model

In Pronto, iterative-SVD is performed on the mean-centered batch of CPU-Ready samples and its latest local model. Iterative-SVD allows Pronto to become a stream algorithm with memory usage proportional to $O(d)$ where d is the number of features in the batch. The resulting U and Σ matrix correspond to the PCs of the latest data combined with historical data.

In Spazio, the nodes perform iterative-SVD on a batch of [0,1]-normalised telemetry data. The telemetry data must have the following property: a value of 0 indicates empty while a value of 1 indicates capacity is full. As we are no-longer mean-centering the dataset before applying SVD, the value we are maximising is:

$$x_i^T \mathbf{A}^T \mathbf{A} x_i = \left\| \begin{bmatrix} - & x_i^T & - \end{bmatrix} \begin{bmatrix} | \\ a_1 \\ | \end{bmatrix} \quad \cdots \quad \begin{bmatrix} | \\ a_n \\ | \end{bmatrix} \right\|^2 \tag{3.1}$$

$$= \left(\begin{bmatrix} - & x_i^T & - \end{bmatrix} \begin{bmatrix} | \\ a_1 \\ | \end{bmatrix} \right)^2 + \left(\begin{bmatrix} - & x_i^T & - \end{bmatrix} \begin{bmatrix} | \\ a_2 \\ | \end{bmatrix} \right)^2 + \cdots + \left(\begin{bmatrix} - & x_i^T & - \end{bmatrix} \begin{bmatrix} | \\ a_n \\ | \end{bmatrix} \right)^2 \tag{3.2}$$

The j^{th} term in the above equation is maximised by choosing $x_i = a_j / ||a_j||$. Hence the x_i that maximises the entire sum can bet interpreted as a 'weighted average' over all a_js. An a_j with a greater length representing higher resource usage will contribute more to the direction of the unit vector x_i.

3.3 Subspace Merging

Given a new batch of samples A', performing incremental-SVD to merge to get the new PCA. As every element in A and A' is [0,1] normalised, the maximised expression follows this property:

$$x_i^T [\mathbf{A}\mathbf{A}']^T [\mathbf{A}\mathbf{A}'] x_i = \left\| \begin{bmatrix} - & x_i^T & - \end{bmatrix} \begin{bmatrix} | \\ a_1 \\ | \end{bmatrix} \quad \cdots \quad \begin{bmatrix} | \\ a_{2n} \\ | \end{bmatrix} \right\|^2 \tag{3.3}$$

$$= \left(\begin{bmatrix} - & x_i^T & - \end{bmatrix} \begin{bmatrix} | \\ a_1 \\ | \end{bmatrix} \right)^2 + \cdots + \left(\begin{bmatrix} - & x_i^T & - \end{bmatrix} \begin{bmatrix} | \\ a_{2n} \\ | \end{bmatrix} \right)^2 \tag{3.4}$$

$$\geq \left(\begin{bmatrix} - & x_i^T & - \end{bmatrix} \begin{bmatrix} | \\ a_1 \\ | \end{bmatrix} \right)^2 + \cdots + \left(\begin{bmatrix} - & x_i^T & - \end{bmatrix} \begin{bmatrix} | \\ a_n \\ | \end{bmatrix} \right)^2 \tag{3.5}$$

As σ_i corresponds to Var_i, with each iterative-SVD, we increase the eigenvalues of Σ. This is problematic as exploding values can result in more expensive matrix operations. In addition, both Pronto and Spazio use Σ as some sort of weighting. Unreasonably large

σ_i values will result in incorrect signal values.

$$\mathbf{B} = \left[\left(\frac{\sqrt{\alpha}}{\sqrt{\alpha+\beta}}\mathbf{A}\right)\left(\frac{\sqrt{\beta}}{\sqrt{\alpha+\beta}}\mathbf{A}'\right)\right] \tag{3.6}$$

$$x_i^T\mathbf{B}^T\mathbf{B}x_i = \frac{\alpha}{\alpha+\beta}\left\|\begin{bmatrix} - & x_i^T & - \end{bmatrix}\begin{bmatrix} | \\ a_1 \\ | \end{bmatrix} \quad \cdots \quad \begin{bmatrix} | \\ a_n \\ | \end{bmatrix}\right\|^2 + \tag{3.7}$$

$$\frac{\beta}{\alpha+\beta}\left\|\begin{bmatrix} - & x_i^T & - \end{bmatrix}\begin{bmatrix} | \\ a_{n+1} \\ | \end{bmatrix} \quad \cdots \quad \begin{bmatrix} | \\ a_{2n} \\ | \end{bmatrix}\right\|^2 \tag{3.8}$$

$$\therefore \min(\mathrm{Var}_i(\mathbf{A}), \mathrm{Var}_i(\mathbf{A}')) \leq \mathrm{Var}_i(\mathbf{B}) \leq \max(\mathrm{Var}_i(\mathbf{A}), \mathrm{Var}_i(\mathbf{A}')) \tag{3.9}$$

However, scaling the concatenated matrix using the method above can upper and lower bound the growth of Σ. This also effectively behaves as an EMA, as each contribution of the previous sample is reduced by $\alpha/(\alpha+\beta)$.

3.4 Capacity Signal

From the model section, we understood $\mathbf{U}$ as the orthonormal vectors that maximise the projected squared distance of the telemetry dataset $\mathbf{A}$. As every element in $\mathbf{A}$ is in the range [0,1], u_1 within $\mathbf{U}$ will have either all positive or all negative values. This means that the other orthonormal vectors $u_2, \ldots, u_n$ will have at least one element that differs in sign from another. As a result, the remaining eigenvalues of $\mathbf{U}$ are not valide resource-usage directions: $\forall 1 < i < n, k \neq 0 : \exists j \in 1, \ldots, n$ such that$(ku_i)_j \leq 0$. As a result, only u_1 gives us a meaningful result: a weighted average of resource usage.

From this we can build a new capacity signal that considers both the estimated workload resource usage and its current resource usage. Given the current workload y, u_1 and σ_1 from the latest U and Σ, a Node's estimated capacity signal k is given by:

$$y_{\text{predict}} = y + k * \sqrt{\sigma_1}u_1 \tag{3.10}$$

$$\max_k \forall i : y_{\text{predict}} < 1 \tag{3.11}$$

As σ_1 is the sum of the projected square distances of the recent workload, we can scale u_1 by this distance to represent the size of expected workload. We could have also averaged the squared distance using $\sqrt{\sigma_1/b}$, but this would just scale k for all nodes and not provide any more information.

3.4.1 Example Scenarios

To better understand how this signal works, I will explore how the signal changes under different different loads. This scenarios will also be used to control the correctness of the signal implementation in the prototype.

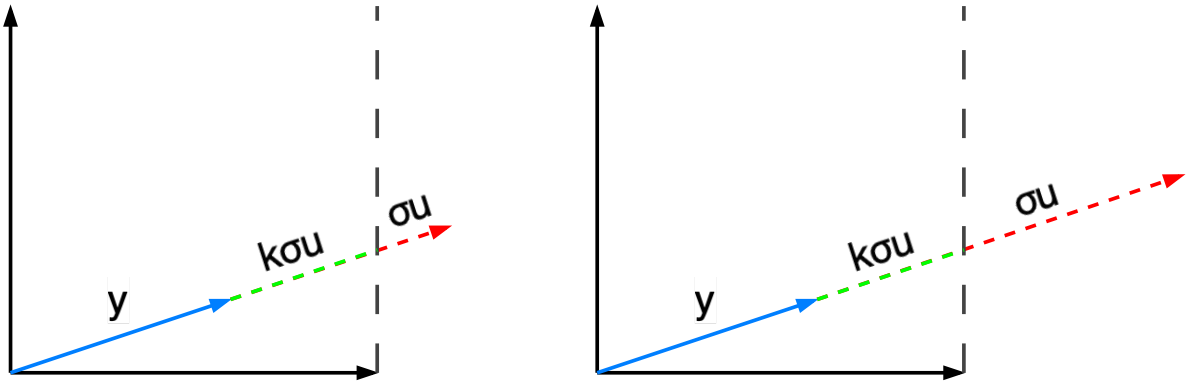


Figure 3.1: Visualisations of a Node's resource usage y and expected resource usage $\sigma_1 u_1$ when learning of conflicting resource utilisation.

Figure 3.1 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. As a result, a Node advertises a smaller capacity signal as it has less capacity for the expected resource usage.

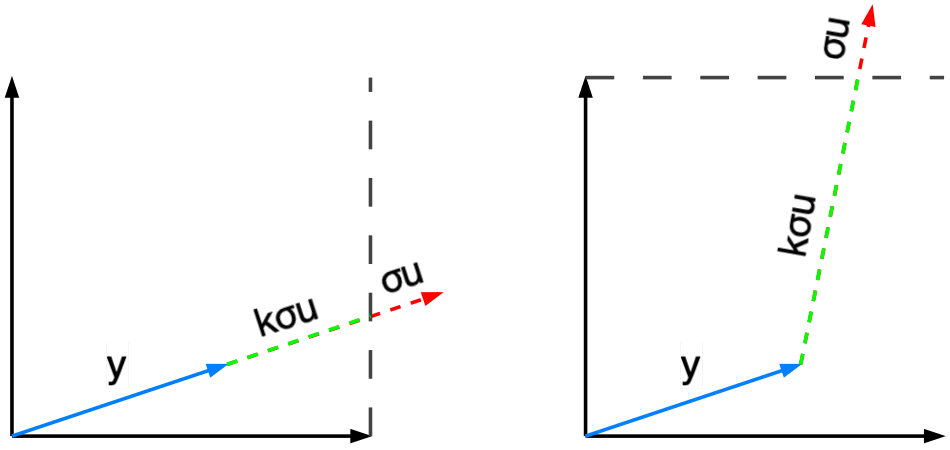


Figure 3.2: Visualisations of a Node's resource usage y and expected resource usage $\sigma_1 u_1$ when learning of complementary resource utilisation.

Figure 3.2 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 dot product of the current resource utilisation and the expected resource utilisation is much smaller. Therefore, a larger constant k is needed to cross a resource boundary. This results in a Node advertising a higher capacity signal as it has more capacity for the new expected resource utilisation.

3.5 Reserve Cost and Capacity

Like Pronto, Spazio's signal uses its current resource usage, and therefore, only reflects Pods that have been scheduled and are running on the Node. For Spazio to work in a system with Pod startup latency, the scheduler must be able to predict the effect a Pod will have on a Node's signal. Spazio assumes that Nodes know the number of currently running Pods, as well as, have the ability to estimate their signal capacity and per-Pod cost (I will explain the prediction methods used in the prototype in section **??**). From With information, a Node can calculate its available capacity from:

$$\text{avail.} = \frac{\text{signal}}{\text{per-Pod cost}} \tag{3.12}$$

$$\text{avail.} = \frac{\text{capacity}}{\text{per-Pod cost}} - \text{pod count} \tag{3.13}$$

$$\tag{3.14}$$

This metric has two useful properties:

- **Dual-Mode:** The available capacity can be calculated using two equations. This is especially useful in Kubernetes as during the creation or deletion of a Pod, the current measured resource usage can experience large spikes. To combat this noise, Nodes can switch to predicting available capacity using estimated capacity and Pod count. This reduces fluctuations in a Node's advertised capacity and improves scheduling decisions
- **Unit of Measure:** This metrics' unit of measure is Pod count. Sending the latest signal, capacity and per-pod-cost values would increase the amount of data sent between Nodes and increase the complexity of the central scheduler's algorithm.

Each Node n will broadcast its avail._n to a central scheduler. This scheduler also tracks each Node's reserved amount as reserved_n. For each Pod waiting to be assigned to a Node, the scheduler performs the following operations:

- **Filter:** Filters out all Nodes n with $\text{avail.}_n - \text{reserved}_n < 1$. Ensures we do not schedule on Nodes that do not have enough resources for another pod.
- **Score:** Score Nodes n by $\text{avail.}_n - \text{reserved}_n$. This ensures we allocate to Nodes which can fit more Pods.
- **Reserve:** Once a node is Node has been chosen, we increment reserved_n by 1 for

that Node. Once a scheduled Pod is no no longer in the Pending state, the central scheduler decrements reserved_n by 1 for the Node n the Pod was assigned to.

3.6 Properties

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

Federated: Pronto executes scheduling plans in a decentralised fashion without knowledge of the global performance dataset. Such approach in Kubernetes could actually decrease performance. The Kubernetes API server handles the publishing and updating of Kubernetes objects. A decentralised system with no global synchronisation or coordination could result in a stampede of Bind requests that could overload the Kubernetes API server and slow down the publishing of incoming Pods. Instead, Spazio uses a central scheduler to score Nodes and perform the final Bind operation. However, it can still be considered federated because of its use of federated PCA []: individual Nodes can have a unified view of the global workload while maintaining their individual autonomy to set their score as they deem fit.

Streaming: Spazio's use of iterative-SVD, like Pronto, means it only requires memory linear to the number of features considered; the required memory is proportional to $\mathcal{O}(d)$. Furthermore, like Pronto, Spazio only requires a single pass over the incoming data without having to store historical data in order to update its estimates.

Unsupervised: Spazio uses a derivative of PCA, a popular technique for discovering linera structure or reducing dimensionality in data. Like Pronto, it exploits the resulting subspace estimate along with the incoming data to reveal patterns in recent resource-usage.

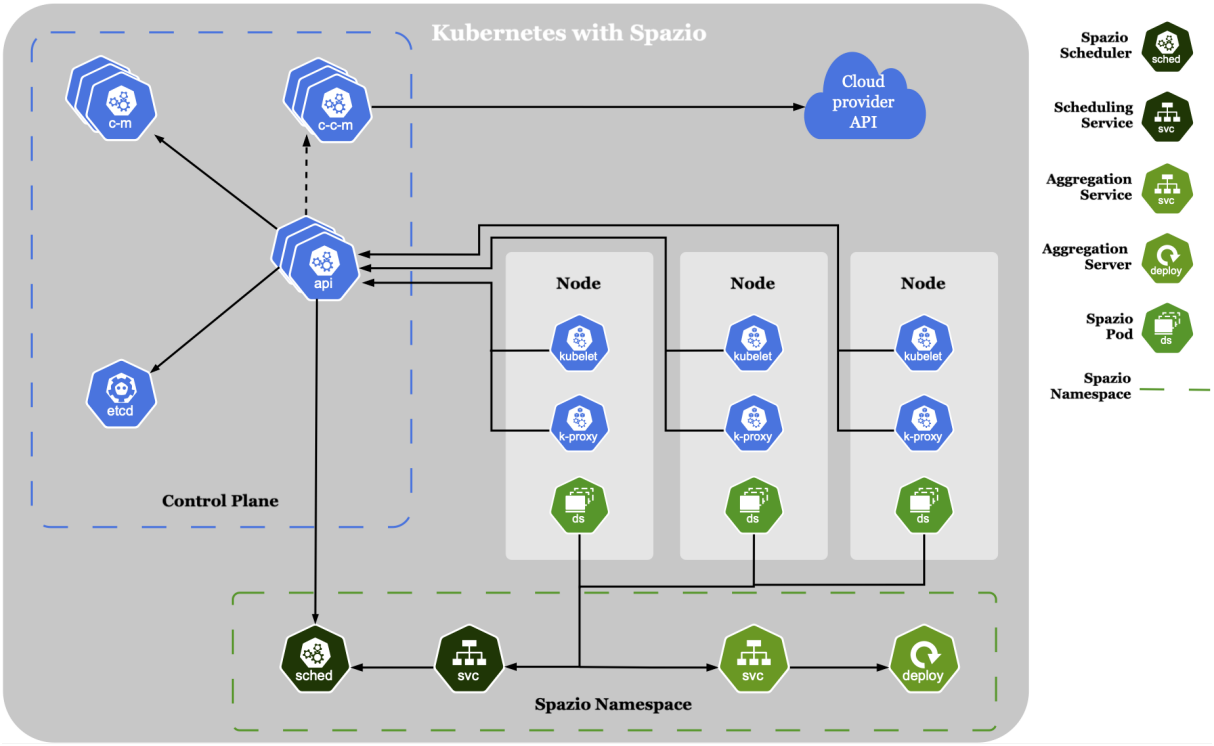
Continuous Value Pronto is a binary signal, which makes difficult to score Nodes. Spazio's avail. $\in \mathbb{R}$, allowing Nodes to be filtered and scored against each other.

Latency Resilient: Unlike Pronto which assumes no communication latency, Spazio was designed with Kubernetes in mind, and thus must consider possible latency in communication and Pod startup. Spazio's avail.'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 avail..

Chapter 4

Implementation

4.1 System Architecture



The Spazio system consists of three core components (shown in figure 4.1):

- Spazio DaemonSet: each Node in the cluster will contain a Spazio Pod. This pod periodically collects telemetry from the Node and generates its local model and a capacity signal, which it sends to the Scheduling Service. When the Spazio Pod deems its local model outdated, it requests the latest aggregated global model from the Aggregation service.
- Scheduler: In Spazio, the scheduler is a **Scheduler Plugin**, implementing custom Filter, Score and Reserve functions. It also acts as the server of the Scheduling service, receiving the latest capacity scores of each Node to inform its scheduling decisions.
- Aggregation Server: This deployment provides the Aggregation service. The Aggregation Server Pod receives local models from Nodes and returns the latest aggregated global model.

4.2 Spazio Pod

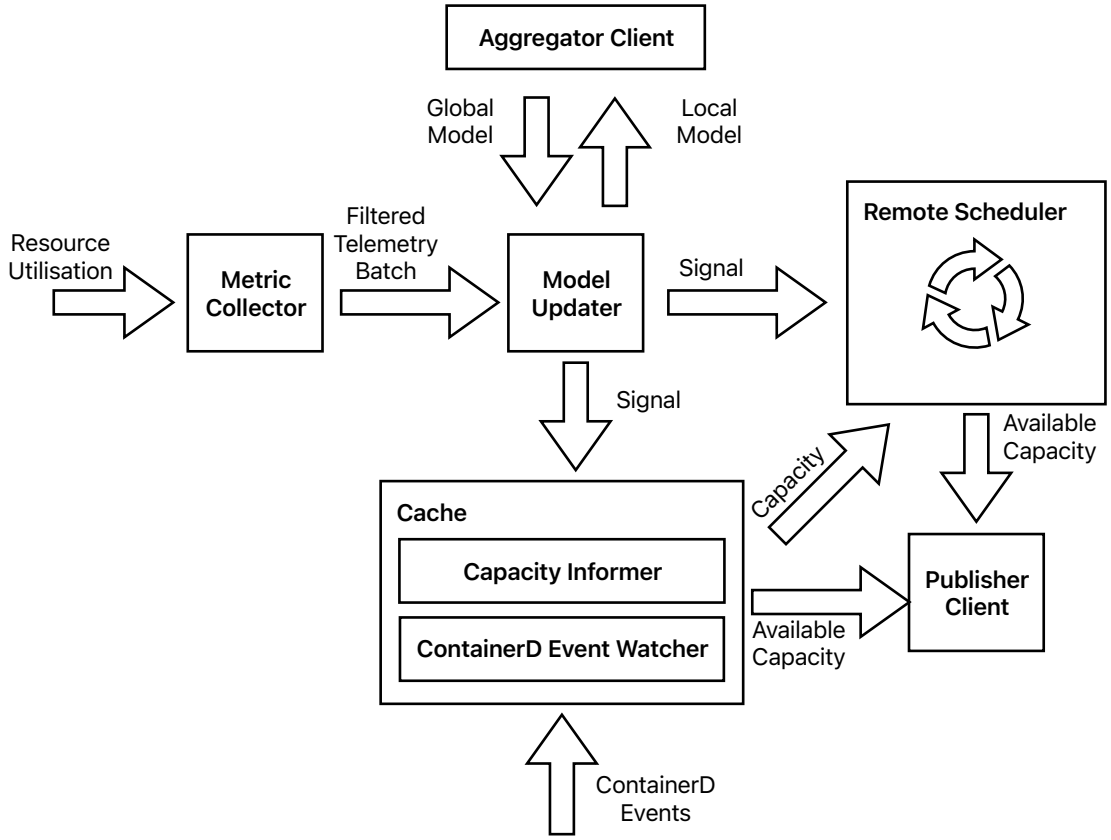


Figure 4.2: Core components within the Spazio pod

As mentioned in section 4.1, the Spazio Pods are defined within a DaemonSet - a DaemonSet ensures that all Nodes run a copy of a Pod. The Spazio Pods collect telemetry from the Node and generate its local model and capacity signal. The following sections will delve deeper into the implementation decisions behind the Spazio Pods.

4.2.1 Metric Collection

To build its local model, the Spazio Pod must first collect telemetry. In Kubernetes there are numerous ways to obtain telemetry data. During my project, I explored two 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 real0time information about running processes and system's hardware.

With Metrics Server, a scraper is used to periodically (default every 15 seconds) collects resource metrics from Kubelets and exposes them from its `metrics.k8s.io/v1` APIService. While it is simple to use, it provides a limited range of metrics, only CPU and RAM utilisation, and introduces another layer of latency. Furthermore, 15 seconds between scraping is too long as Pods may complete in under 15 seconds, and therefore, risk not being detected at all.

TODO: INCLUDE POINT THAT LOCAL MODEL TAKES B SAMPLES AND THEREFORE INCREASES LATENCY.

On the other hand, `/proc/` can be read from with very little latency, providing the most up-to-date view of the current state of the system. Furthermore, `/proc/` contains various files and subdirectories, each providing specific information. Examples include, `/proc/stat` which contains the amount of time CPU cores spend in different states, `/proc/meminfo` provides statistics about memory usage, `/proc/diskstats` presents the raw, low-level I/O statistics of all block devices. Finally, the metrics 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.

Due to the overwhelming benefits, I decided to use `/proc/` as the source of my telemetry data. The following sections the different metrics I considered, providing explainations to implementation decisions.

Utilisation Metrics

Utilisation metrics, such as % CPU utilised or % Memory available, are de-facto metrics when implementing industry-standard [15, 16]. These metrics can also be collected from `/proc/`.

To collect CPU utilisation, I used the `/proc/stat` file. This file reports the cumulative count of "jiffies" (typically hundredths of a second) each CPU spent in a specific mode [17]. I can then calculate CPU utilisation using:

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

`/proc/meminfo` can also be used to collect memory utilisations. This file shows a snapshot of the memory usage in kilobytes. The percentage of memory used can then be calculated from the given field:

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

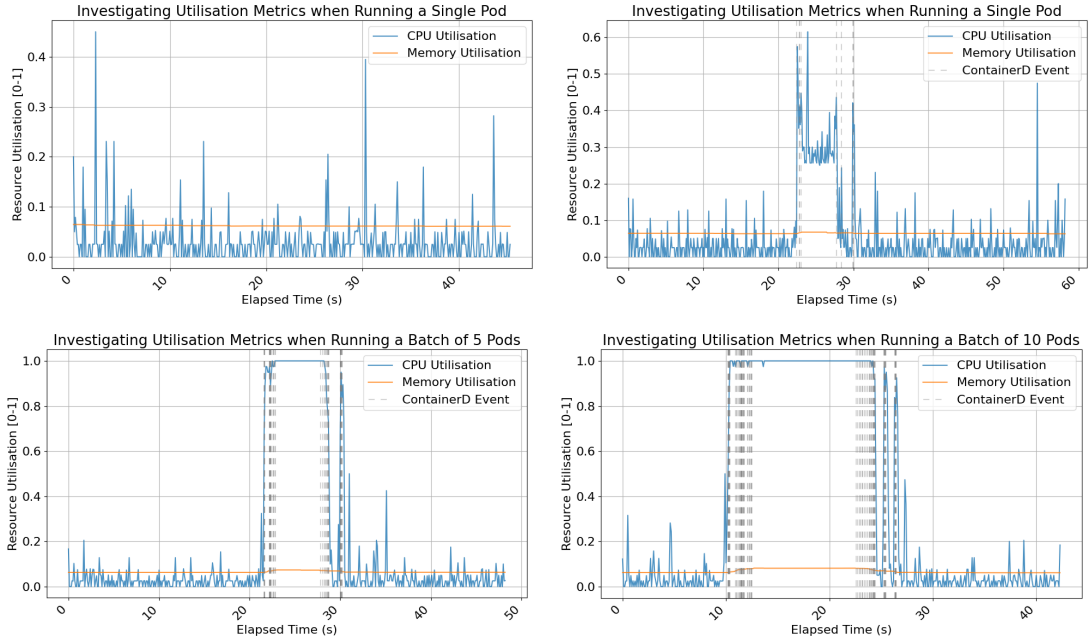


Figure 4.3: In this figure we sample CPU and memory utilisation from values of `/proc/stat`, `/proc/meminfo` at 10Hz during various Kubernetes workload.

To evaluate whether utilisation metrics were suitable for Spazio, I measured their output under different different workloads. Figure 4.3 presents the metrics behaviour when running different Job sizes of `pi-2000` Pods. As the cpu-intense workload involves calculating π to 2000 digits, we can see how the CPU utilisation correctly increases for that node while memory utilisation remains constant.

Issues of using CPU Utilisation

While evaluating early versions of the prototype which only used utilisation metrics, its lackluster throughput compared to the default `kube-scheduler` illuminated a problem of CPU utilisation. When deploying 1000 Pods, each requesting 100 milliseconds of CPU

time, across 19 Nodes, the `kube-scheduler` would immediately allocate all pods evenly across the Nodes. This would result in ≈ 45 pods running on each node. Meanwhile, Spazio with only utilisation telemetry would allocate at most 5 Pods at once on a Node. In both situations, CPU utilisation was 100%, however, the default `kube-scheduler` managed to achieve a high throughput with a long-tailed distribution of individual Pod completion times.

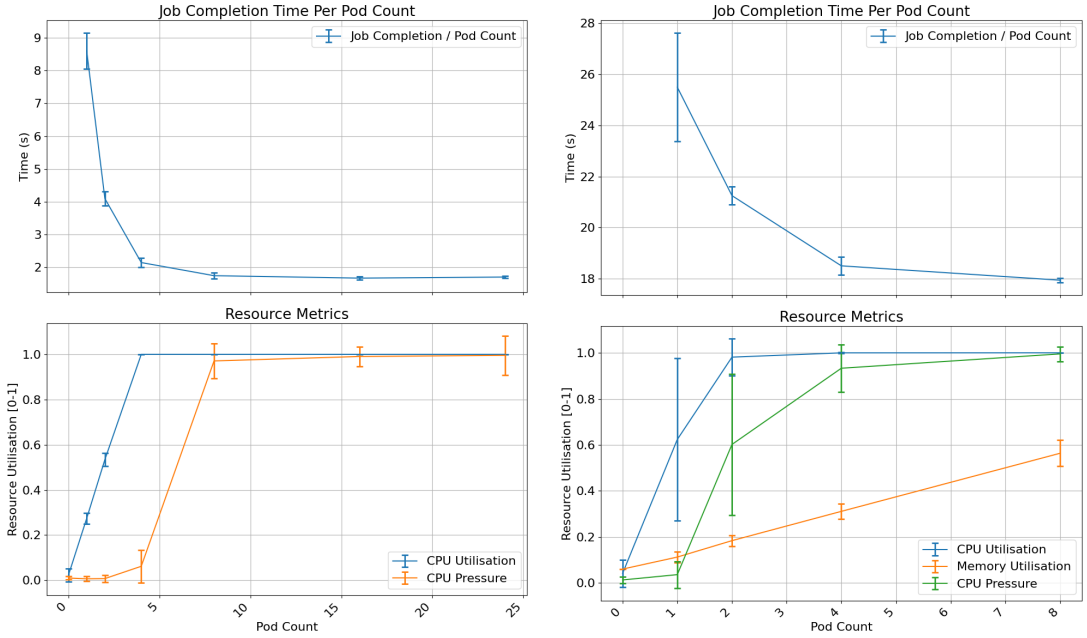


Figure 4.4: In this figure we sample CPU and memory utilisation from values of `/proc/stat`, `/proc/meminfo` at 10Hz during various Kubernetes workload.

TODO: CHANGE THIS GRAPH TO JOB COMPLETION / POD COUNT

As a result, I decided to further investigate this phenomenon. Figure 4.4 shows how a Job's completion time changes as you increase its completion and parallelism count when running different Pods: cpu-intense Pi-2000 and a small ML workload (training and inference). I also decided to include measurements from `/proc/pressure`. This investigation revealed that the relationship between the number of pods running on a node at a time and their completion time showed a close-to linear relationship. I hypothesise that this is caused by the cluster being run on top of VMs. As hypervisors aim to abstract the underlying hardware from the VMs, it also inadvertently hides hardware effects, such as cache contention and CPU thrashing. This means that high CPU utilisation may not truly signify performance degradation. Therefore, CPU utilisation is not a definitive measure of resource capacity, and explains why older versions of the prototype were not able to push through in terms of pod count and achieve higher throughput.

Combining CPU Utilisation and CPU Pressure

Using raw `/proc/pressure` metrics would also not suffice as these metrics barely increase until the pod count pushes past the number of cores. Therefore, the intial per-Pod costs for the first few Pods would be unreasonably low and would result in a Node advertising a falsely high Pod capacity. I instead chose to combine both CPU utilisation and CPU pressure into a single metric using:

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

This ensures the metric steadily increases with the number of Pods running on a Node, but prevents the CPU metric from reaching 1 to quickly. Instead, the CPU metric is only maximised once `/proc/pressure` measures that there was always at least one thread waiting for the CPU.

Figure 4.5: In this figure we investigate the behaviour of combining both CPU Utilisation and CPU Pressure.

4.2.2 Filtering Metrics

While Spazio doesn't perform peak detection, the signal will still be influenced by short-lived spikes. As mentioned in the earlier section, pod creation and deletion incurs a visible spike in resource usage. This spike introduces noise into a Node's local model, as well as, its capacity signal. A noisy signal can also make it difficult to achieve accurate capacity and per-Pod cost estimation. As such, I needed to de-noise the original metrics.

Investigation into the recorded telemetry showed that the spikes caused from container events would last ≈200 milliseconds. Thus when sampling at a 10Hz frequency we can use Dynamic EMA to suppress container-event caused spikes but allow the smoothed metric to quickly converge on the new utilisation if the spike exceeded the 300 millisecond threshold.

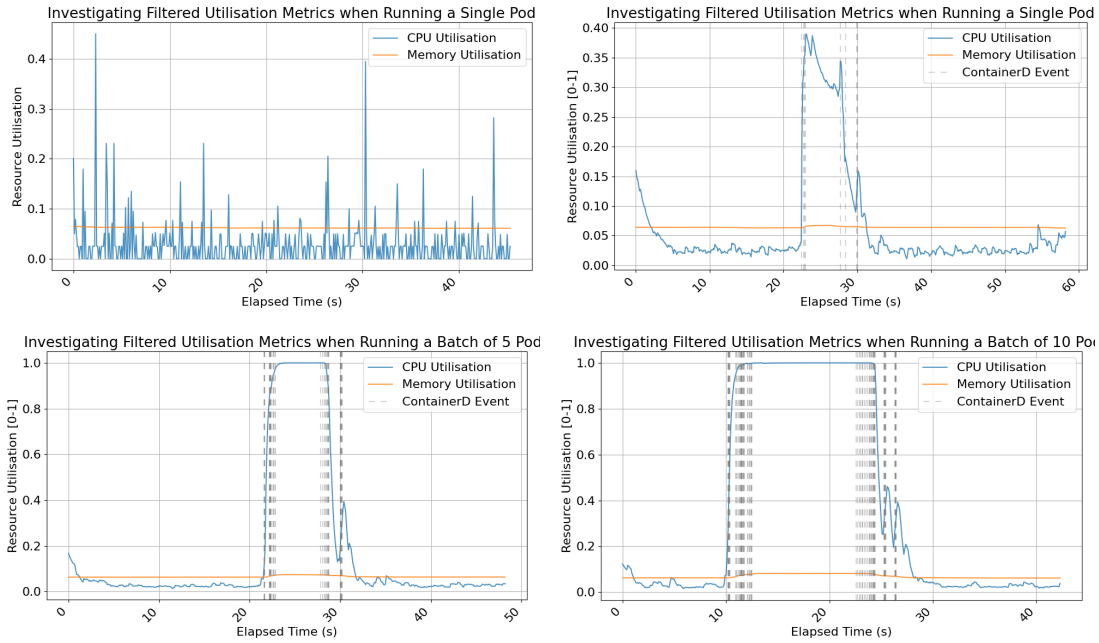


Figure 4.6: This figures shows the smoothed metrics under different workloads.

Comparing Figure 4.3 with 4.6 demonstrates the dampening of the Dynamic EMA and its quick responsive to prolonged changes in workloads. While more sophisticated filters exist, they were not considered due to their higher computational cost and thus would rob scheduled pods of the available resources. I had considered applying the filter directly to the signal instead of the collected telemetry, but by only filtering the signal, it would allow the local model to be polluted by these container-event resource spikes.

4.2.3 Signal Generation

In the Spazio Pod, the signal is calculated at a frequency of 1Hz. This frequency is the same frequency at which the local model is updated. This ensures that the signal tracks with the local model. Increasing the frequency would give the central scheduler a more up-to-date view of a Node's resource status, but would incurr additional resource overhead. Not only would this overhead reduce the available resources to other Pods, it would be reflected by a lower signal, potentially reducing capacity a Node advertises.

To check for a correct implementation of the signal, I measured the calculated signal when under the situations described in 3.4.1. For the conflicting workload, I had the measured Node execute a light CPU-focused workload (`ng-stress --cpu=8 --cpu-load=25`) with the surrounding Nodes executing a CPU-intense workload (`bpi(2000)`). For the complementary workload, the measured Node executed a light Memory-focused workload (`ng-stress --vm=4 --vm-bytes=4G`) with the surrounding Nodes also executing the same CPU-intense workload as in the conflicting scenario. The measured signals are shown in figures 4.7 and 4.8

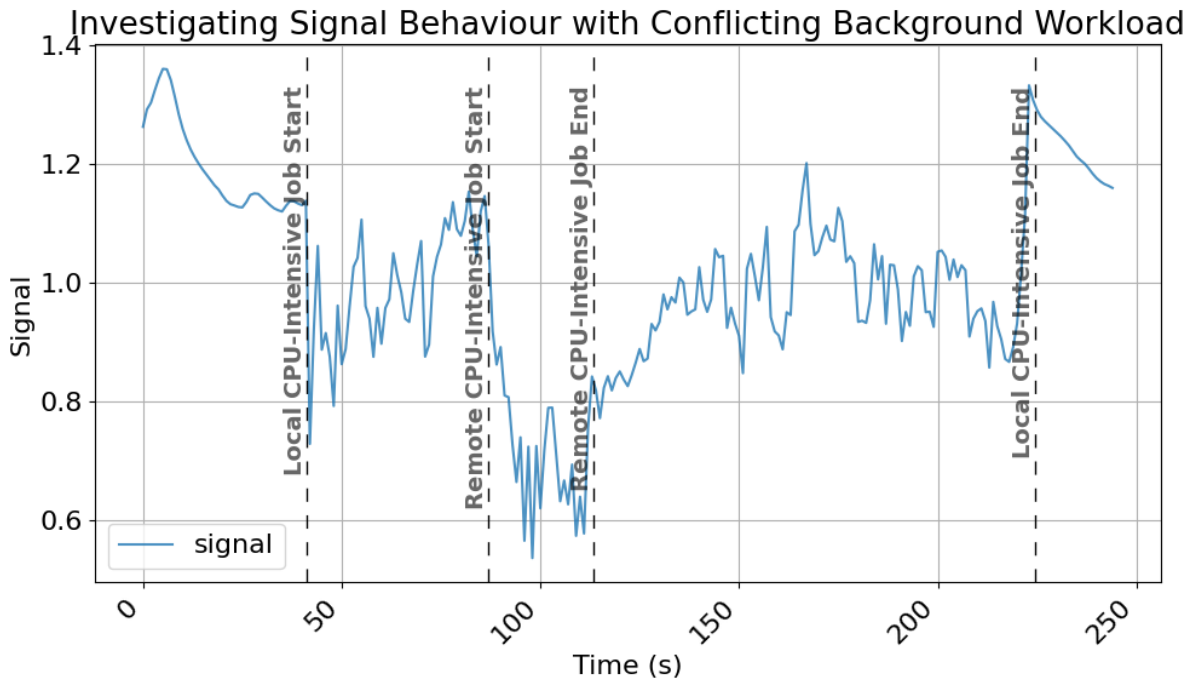


Figure 4.7: The calculated capacity signal of a Node running `ng-stress --cpu=8 --cpu-load=25`. While running this workload, 1000 Pods running `bpi(2000)` was scheduled across surrounding Nodes.

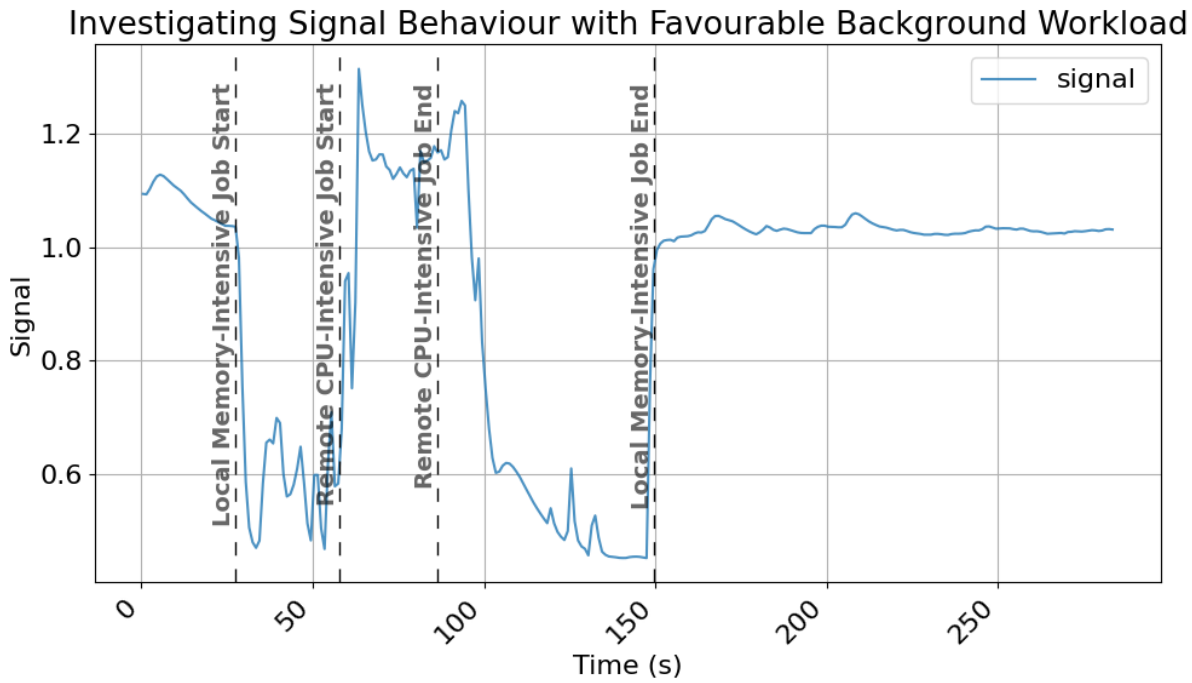


Figure 4.8: The calculated capacity signal of a Node running `ng-stress --vm=4 --vm-bytes=4G`. While running this workload, 1000 Pods running `bpi(2000)` was scheduled across surrounding Nodes.

Figures 4.7 and 4.8 demonstrates how a Node's capacity signal reacts to changes in surrounding workloads when experiencing different resource usage. In Figure 4.7, the measured Node is running a light CPU-focused workload. Once the CPU-intense workload is scheduled on surrounding Nodes, the measured Node's capacity signal drops. This is expected as the local models of surrounding Nodes will reflect a more costly CPU-focused workload. As a result, when the measured Node aggregates its local model, its new model will reflect this higher CPU usage with a larger σ_1 value. As the measured Node's directions of the measured Node's resource usage and expected workload are still CPU-focused, the increased σ_1 value results in a lower k, and thus a lower capacity signal.

In Figure 4.7, the measured Node is running a light Memory-focused workload. When the CPU-intense workload is scheduled on surrounding Nodes, the measured Node's capacity signal increases. This matches 3.4 hypothesis. Like in the previous scenario, the global model reflects a heavy CPU-focused workload, and thus, so will the aggregated local model of the measured Node. However, as the measured Node's resource usage is now orthogonal to the expected resource usage, a larger constant k is needed to maximise a resource. This reflects the sharp increase in capacity signal we observe in figure 4.8.

4.2.4 Calculating Cost and Capacity

As the new proposed signal also used its current resource usage in the calculation, pods scheduled on a node would not impact the signal until they had started running. Thus, I needed a means of reserving the signal to prevent the scheduler from running away and greedily assigning all pods to the node with highest score. As pod workload may vary over time, I needed a means of dynamically estimating the "signal cost" of assigning a pod to a node. In addition, I needed the method to be able to handle multiple pods being created and deleted at once.

Detecting Pod Events

There are numerous ways to detect the addition and removal of pods from a node. I investigated two approaches : watching the Kubernetes API and watching the ContainerD events. The goals of the listeners were as follows:

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

The latter requirement is needed as the Dynamic EMA used on the resources won't be able to smooth longer resource spikes caused by a burst of multiple container events. Instead, by detecting pod events earlier, we can halt capacity and per-Pod cost estimations until the burst has passed.

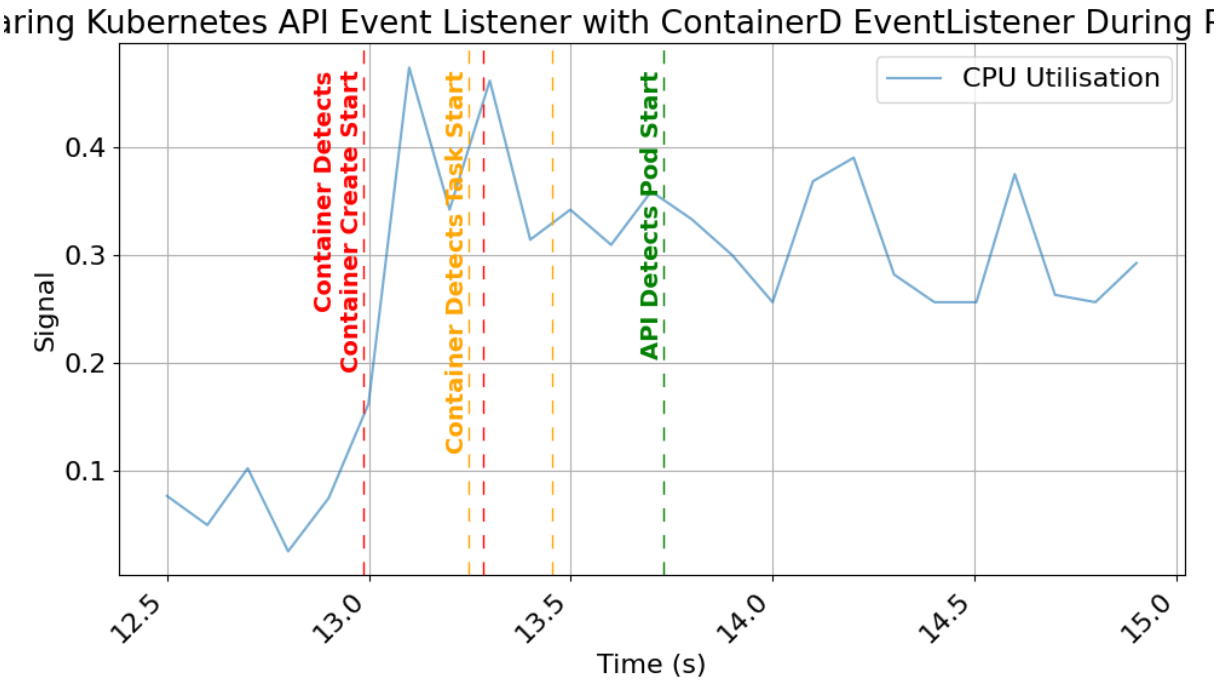


Figure 4.9: When different event listeners detected the creation of a Pod.

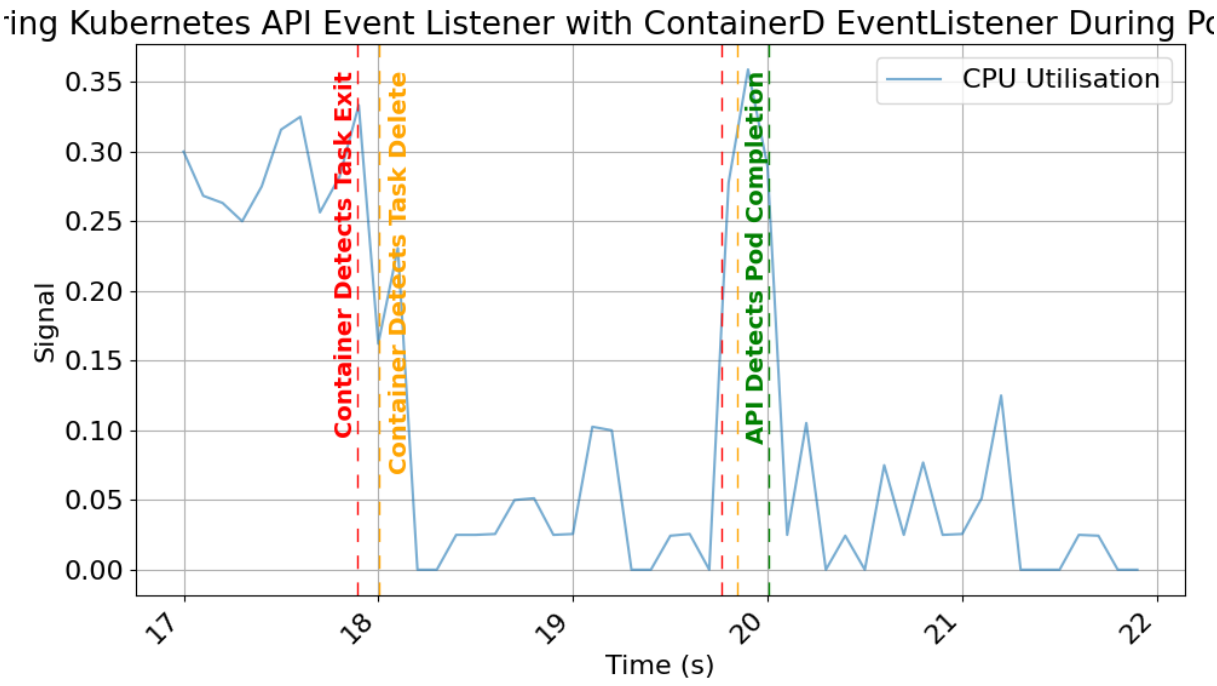


Figure 4.10: When different event listeners detected the completion of a Pod.

As shown in figures 4.9 and 4.10, the two-way latency from sending events to the `kube-apiserver` before then detecting results in the Kubernetes API listener to miss the spikes caused by container events. Without a forewarn, nodes will include container event resource spikes into their resource predictions. On the other hand, we can see that certain container events precede the spikes. While handling container events is more complex, it can alert the node of potential spikes and thus reduce the introduction of noise into our calculations.

Estimating Pod-Cost

As mentioned in Section 3.5, Spazio assumes a Node has the capability of estimating its capacity and per-Pod cost. I needed a streaming signal processing technique with a low-overhead and the ability to work with a dynamic system (handle changes in workloads over time). A Kalman filter [] is a powerful 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.

TODO: Should I explain what a Kalman filter is in more detail? I devised three Kalman filter-based approaches to estimating reservation costs.

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

$$\Delta\text{signal} = \Delta\text{no. of running pods} \times \text{cost}$$

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

$$\text{signal} = \text{capacity} + \text{per pod cost} \times \text{no. of pods}$$

- Two separate 1D Kalman Filters predicting the equation:

$$\text{signal} = \text{capacity} + \text{per pod cost} \times \text{no. of pods}$$

  . This dual filter approach has each filter learn a separate variable. By splitting the filter into two, it prevents non-zero covariance entries in the P matrix which cause the oscillations seen in the 2D Kalman filter.

To smooth out predictions, I employed two techniques. The first technique utilised the container event listener: on detection of container churn, the Spazio Pod would temporarily halt estimations until it had passed. Secondly, I halted estimates once the signal reached 0. When the capacity signal is 0, it indicates that at least one resource is at capacity. If this occurs and another Pod begins running on the same Node, the signal still outputs 0. Without halting estimates, the filters would continue to decrease the per-Pod cost, resulting in an inaccurately low per-Pod cost and thus advertise too large of a capacity.

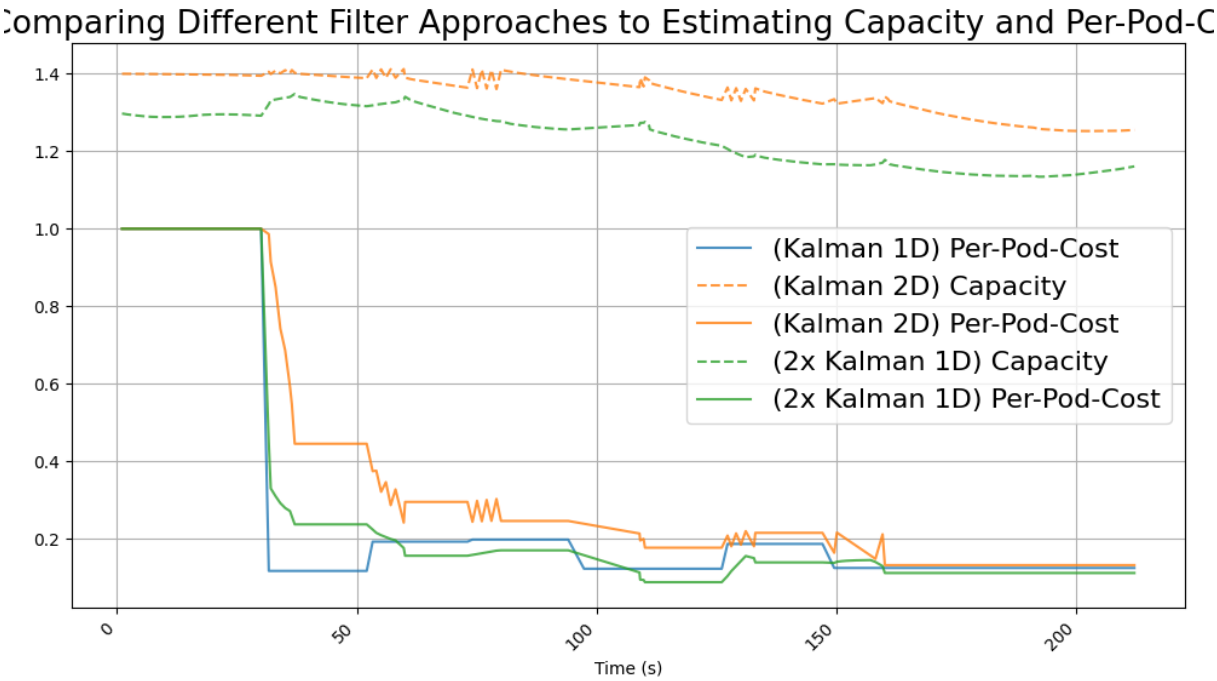


Figure 4.11: The estimates of the Kalman filters when Node experiences variable-sized bursts of `bpi(2000)` Pods.

To decide the optimal approach, I observed each methods' behaviour when executing

Jobs of various sizes. This is depicted in Figure 4.11. While Δ-based Kalman filter produced accurate estimates of per-Pod cost, its one-dimensional property prevents it from estimating the capacity of the Node. However, its stability inspired the double Kalman filter approach. The 2D Kalman filter approach provides a simple method for estimating both the Node's capacity and its per-Pod cost. However, to ensure faster convergence, I used large constants in the Q matrix. This resulted in large oscillations as the filter attempts to correct any error by modifying both the capacity and cost variables. Finally, the dual Kalman filters approach converged quickly on an accurate estimation without exhibiting the oscillations seens in the 2D Kalman filter. This made the Dual 1D Kalman filter the optimal choice.

4.3 Aggregation Server

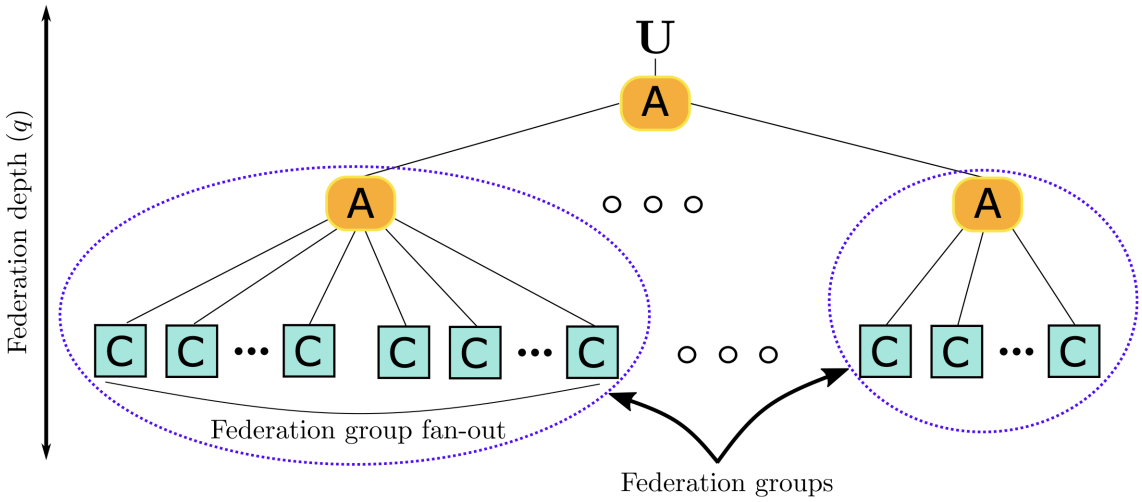


Figure 4.12: How local models are aggregated in Pronto. Dedicated aggregator nodes propagate the updated subspaces until the root is reached.

Pronto aggregation approach is similar to the distributed agglomerative summary model (DASM) []. Local models are aggregated in a "bottom-up" approach following a tree-structure depicted in Figure 4.12. While the this approach is stated to required minimal synchonisation, it requires mutliple dedicated Pods and Kubernetes inherent communication latency could result in a significant latency between a change in workload and the global model reflecting the change.

Therefore, I decided to use a flat on-the-fly approach to aggregation: when the Aggregation Server receives an aggregation request with a Node's local model, rather than aggregating the model before returning the result, the server enqueues the model to be aggregated and returns its latest view of the aggregated model. This implementation

trades consistency for latency.

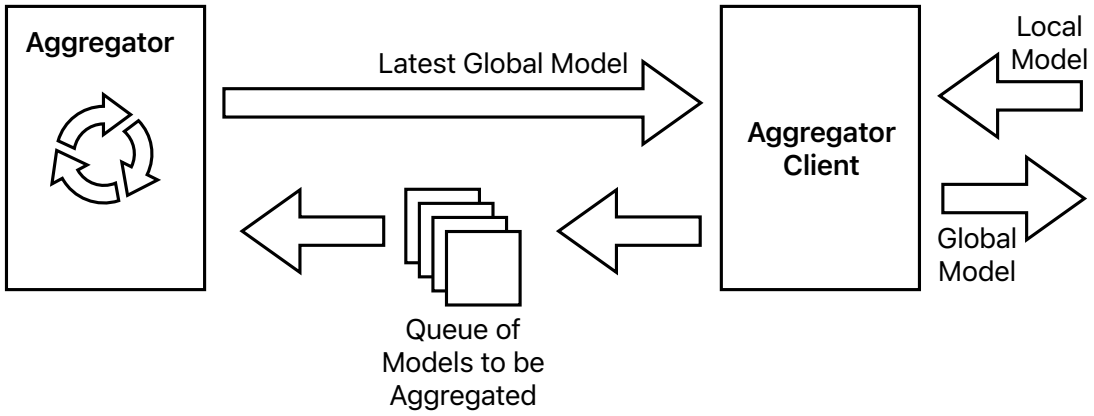


Figure 4.13: The components within the Aggregator Server.

The Aggregation Server executes a thread which waits on a queue of local models to aggregate. When the queue is non-empty, the thread pops the local model and performs a Subspace Merge operation (as defined in 3.3). To prevent a single local-model from influencing the global model completely, it uses an $\alpha = \#$ of Nodes$-1, \beta = 1$. Aggregation clients perform a gRPC call to the address specified by the Aggregation Service. This gRPC function is implemented by the server and takes the local model of a Node as its argument. The function first enqueues the local model to be merged and returns the latest view of the global model. This has the clients themselves aggregate their local model with the received global model. By removing the aggregation of the model from the gRPC call's critical path, we reduce the work done by the Aggregation Server and allows the Server to handle more clients.

4.4 Scheduler Pod

4.4.1 Kubernetes Scheduler Plugin

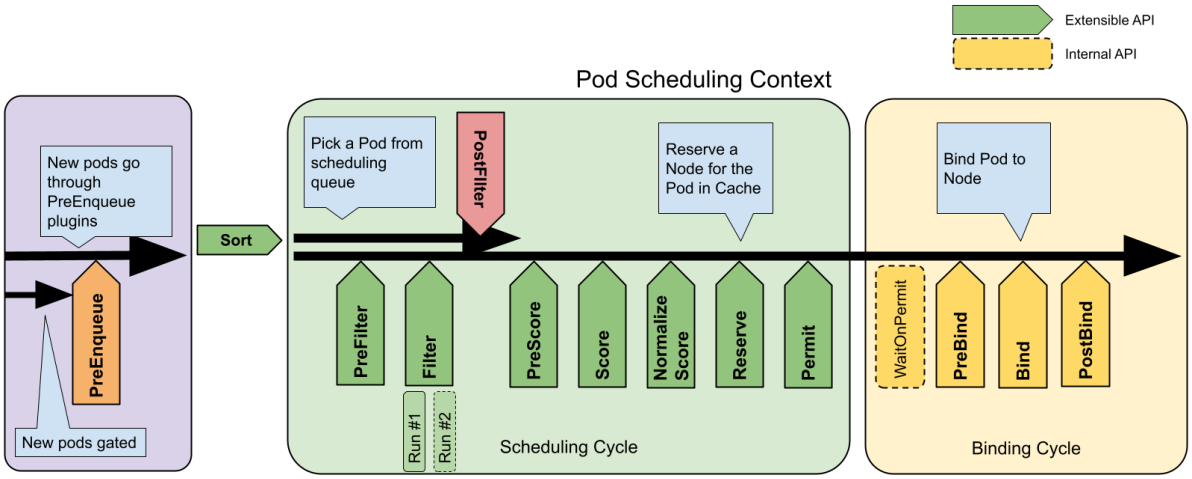


Figure 4.14: The available scheudling framework extension points [18].

The scheduling framework (depicted in Figure 4.14) is a pluggable architecture for the Kubernetes scheduler. The scheduling framework defines a set of extension points, which scheduler plugins have to 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.

Implementing a Spazio-based scheduler plugin is much simpler than implementing a scheduler from scratch, while offering many performance benefits. Firstly, the scheduler operations defined in 3.5 align with the extension points offered by the scheduling framework. Secondly, the scheduling framework gives me access to efficient data structures and algorithms designed by Kubernetes engineers. Finally, scheduling plugins provide another layer of customisability: you can selectively enable or disable default plugins, or define the order in which custom plugins run to augment the overall behaviour. This means that the Spazio plugin could be used in tandem with other plugins to improve scheduling decisions.

To keep track of pod reservations, the Spazio scheduler uses a `map` from Pods ID to Node ID and a Kubernetes API Pod Event Listener. On Reserve, the scheduler increments a Node's reserve quantity and stores a record in the `map` with the Pod ID and the corresponding Node ID. The scheduler's API listener watches for any Pods that move from the 'Pending' status and if there exists a record within the `map` with the Pod's ID, the corresponding Node's reserve quantity is decremented.

Chapter 5

Evaluation

This section is focused on the empirical evaluation of Spazio as a Kubernetes scheduler. As I was unable to find any telemetric-only schedulers, I decided to compare SPAZIO against the default Kubernetes scheduler throughout this section. `kube-scheduler` is an industry-standard scheduler that was built upon the lessons learnt from Borg [], and thus, has been thoroughly optimised and battle tested. As `kube-scheduler` is a Pod description-based scheduler, I use mutliple different resource requests to highlight how the performance achieved can vary depending on description.

The chapter is divided into the following sections: **Overhead - Section 5.2:** I measure the computational overhead incurred from running the SPAZIO Pods on every Node. To ascertain this effect, I complare the Job Completion time with and without the deployment of the SPAZIO Pod DaemonSet

Different Workloads - Sections 5.3,5.4, 5.5: These sections investigate the performance of SPAZIO when deploying different workloads. For each workload, I investigate four metrics: Job Completion, Number of Running Pods, Pod Completion and Resource Usage. Job Completion measures the time it takes for all the Pods specified by a Job to complete. This then acts as a strong indicator of overall throughput of a scheduler. The Number of Running Pods will be used to explain the achieved Job Completion. Pod Completion instead focuses on the distribution of the time it takes for the Pods to complete. The distribution gives an indication of the amount of contention that occurs during the deployment of the Job. A tail-heavy distribution indicates high levels of contention and that the scheduler is overloading a Node. Finally, resource utilisation will focus on CPU and Memory utilisation. I decided to only measure these two resources as they are the only two that are currently considered in the scheduling decisions made by both SPAZIO and the default Kubernetes scheduler.

Workload Isolation - Section 5.6: In this section I investigate the workload-isolation provided by SPAZIO and the default Kubernetes scheduler. Workload isoolation is considered a core component of QoS scheduling, and thus is a useful metric to observe.

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 Spazio, it contains the Spazio 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 Spazio Overhead

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

Number of Pods	**% Overhead with Spazio**
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 5.1: The overhead incurred when running Spazio 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 Spazio's overhead, I used Pod's executing `bpi(2000)`. Table 5.2 presents the relative change in Job Completion time with Spazio 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 Spazio is more visible. From these observations we can conclude that Spazio has $\approx 2\%$ overhead.

5.3 CPU-centric Workloads

For this experiment, I used a CPU-heavy workload: Pods calculatig π to 2000 digits (`bpi(2000)`). This simple workload helps visualise the impact of combining CPU Pressure with CPU Utilisation.

5.3.1 Throughput

During the measurements with the default Kubernetes scheduler, I used the following CPU requests: 100m, 200m, 500m, 1000m. The results are given in Table 5.2

Scheduler	**Requested**	**Job Completion (s)**				
	CPU	**100 Pods**	**250 Pods**	**500 Pods**	**750 Pods**	**1000 Pods**
Default	100m	15.7 ± 0.6	31.7 ± 2.1	56 ± 1.7	84.7 ± 0.6	112 ± 0.0
Default	200m	15.7 ± 0.7	30.7 ± 0.6	55 ± 1	79 ± 0.0	103 ± 1
Default	500m	15.7 ± 1.2	32 ± 2.6	57.7 ± 0.6	81 ± 2	104 ± 2.1
Default	1000m	21 ± 2.0	54.7 ± 0.6	96 ± 2.6	133 ± 0.6	175 ± 1
Spazio		20.3 ± 0.6	35.3 ± 0.6	60.3 ± 2.5	89 ± 2	109± 1

Table 5.2: Job Completion of Job deployments with different Pod counts. Each Pod executed `bpi(2000)`. For the default scheduler, the requested resources are also given

From Table 5.2, we can see how much the performance of the default scheduler varies depending on the amount of CPU time requested. Over-estimating requests can result in the CPU being underutilised, while under-estimating can result in too many Pods running on a Node at once. With SPAZIO, we observed only observed a 10% reduction in Job Completion time. This can be attributed to its use of telemetric data: CPU utilisation amd CPU Pressure. In Section 4.2.1, I investigated these metrics and showed how with `bpi(2000)` Pods, these metrics indicated full capacity when running 4-8 Pods. As the capacity signal of a Node is tied to these metrics, the Node's will never advertise more capacity than $\approx$ 8 Nodes.

TODO: POTENTIALLY MENTION HOW 1000M AND 500M CORRESPOND TO THE LIMIT OF USING CPU UTILISATION AND CPU PRESSURE

5.3.2 Pod Completions

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

Table 5.3: Pod 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

I decided to further investigate the behaviour of the schedulers by measuring the Pod Completion time given in Figure 5.3. From this table, we observe that the lower request default schedulers ends up with a very tail-heavy distribution for Pod Completions. Increasing the request to 500 milliseconds, reduces the mean by $\approx 80\%$ while also reducing the skew. However, CARICO is able to achieve the lowest Pod Completion distribution. To help explain the reason by this result, I also measured the number of Pods running on each Nodes during the Job execution, and present the values in Figure 5.1.

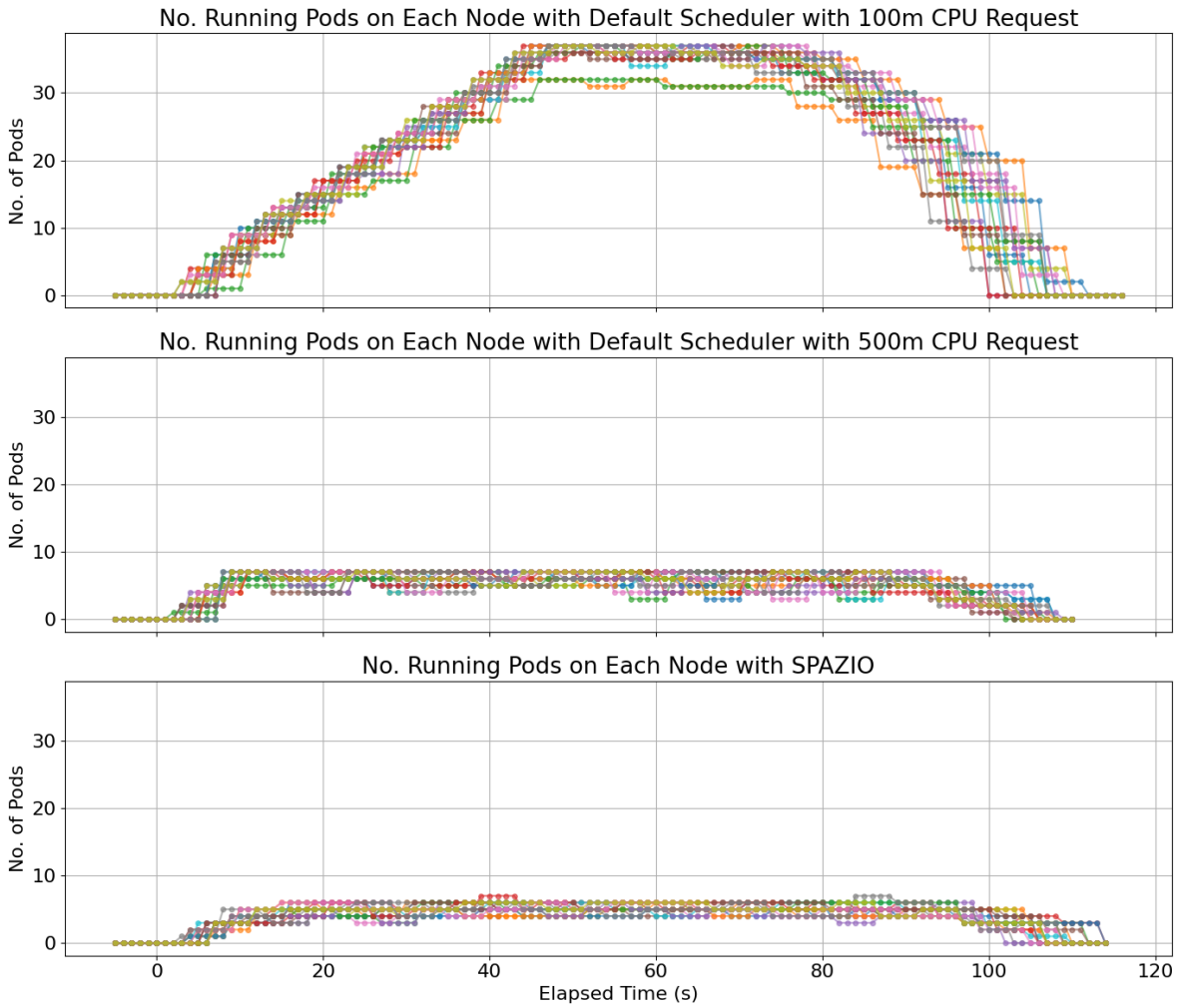


Figure 5.1: The number of `bpi(2000)` Pods running on a Node during the execution of a Job with 1000 Pods.

Figure 5.1 depicts the different approaches taken by each scheduler. As the default Kubernetes Scheduler tackles scheduling as if it were a bin-packing problem, it schedules as many pods that can fit on a Node; each Node has 4000m CPU capacity, 1000m per core. As a result, the 100m request results in a stampede of Pods while the 500m request and CARICO assign a relatively constant number of Pods. However, the default scheduler with 500m Pods takes the edge in terms of throughput as it achieves a higher number of running Pods per Node.

TODO: SHOULD I TALK ABOUT NOT OVERPROVISIONING AS MEMORY USES OOM KILLS IF FULL AND THEREFORE RISKED CORRECTNESS OF SIGNAL

5.3.3 Resource Utilisation

Figure 5.2 shows the resource utilisation during the execution of a Job with 1000 Pods. The default Kubernetes scheduler with 100m CPU requests achieves 100% CPU utilisation

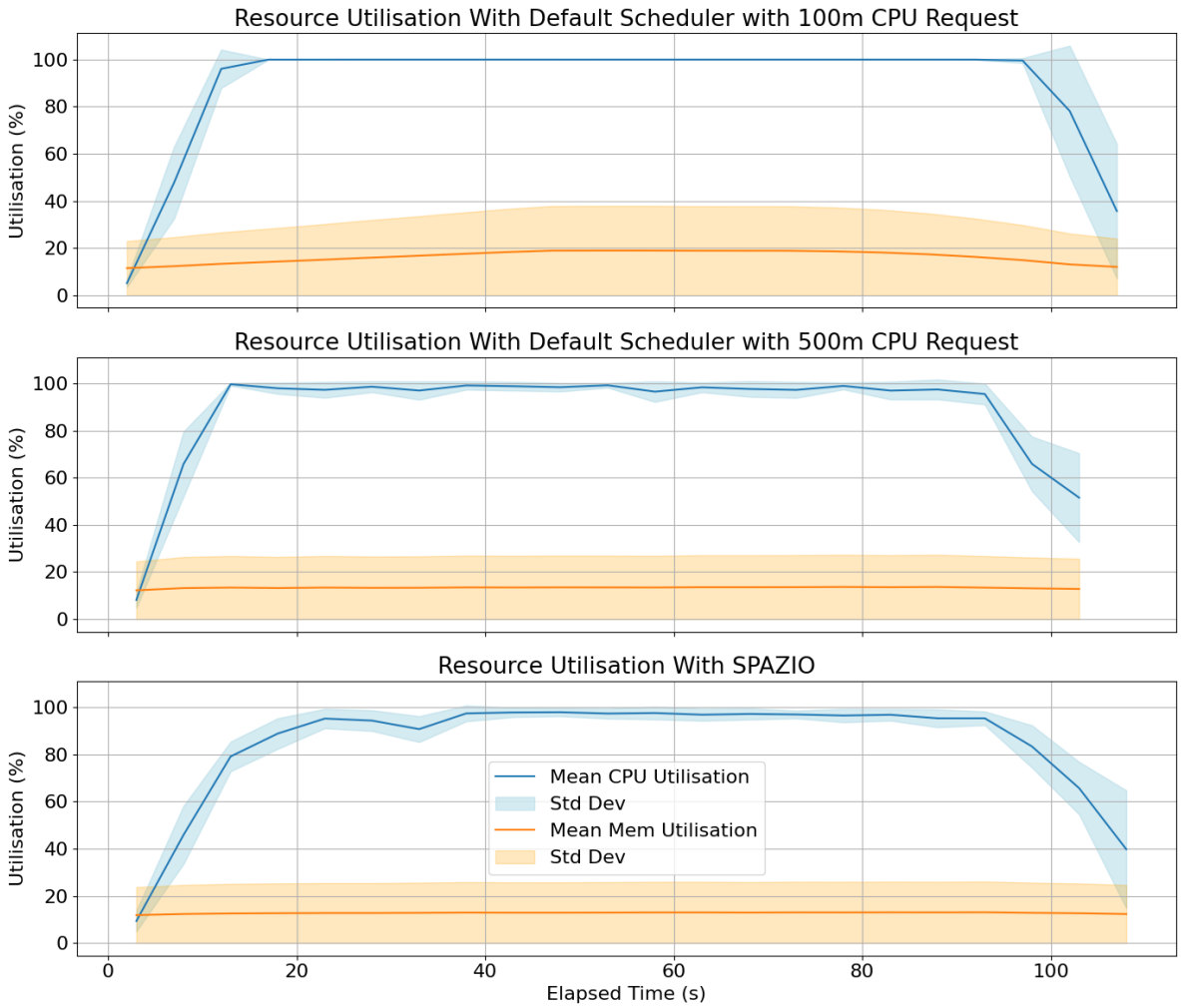


Figure 5.2: the resource utilisation when scheduling a job with 1000 pods executing `bpi(2000)`. the top figure gives the resource utilisation when scheduling with the default kubernetes scheduler. the bottom figure gives the resource utilisation when scheduling with spazio.

due to the large number of Pods running on each Node. The resulting allocation even results in a visible increase in the Memory usage.

5.4 Memory-centric Workloads

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.

5.4.1 Throughput

Scheduler	Requested		Job Completion (s)	
	CPU	Memory	100 Pods	200 Pods
Default	200m	750Mi	144.3 ± 0.6	362 ± 20.4
SPAZIO			189 ± 4.36	353.3 ± 12.7

Table 5.4: Job Completion of Job deployments with different Pod counts. Each Pod executed a small ML workload. For the default scheduler, the requested resources are also given

The Job Completions from the experiment are given in Table 5.4. CARICO achieves a worse performance in the smaller 100 Pod Job, but actually overtakes the default Kubernetes scheduler with the 200 Pod Job.

5.4.2 Pod Completions

Scheduler	# Pods	Mean	Std.	Min.	25%	Median	75%	Max.
Kubernetes	100	127.38	5.29	118	123.00	126.00	132.00	138.00
SPAZIO	100	59.02	14.62	27.00	55.00	61.00	66.00	94.00
Kubernetes	200	225.19	36.19	45.00	231.00	236.00	239.00	244.00
SPAZIO	200	64.46	14.33	23.00	57.75	67.00	73.25	89.00

Table 5.5: Pod 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

Table 5.5 gives the Pod Completion Distribution of the ML-based Jobs. When using the default Kubernetes scheduler, increasing the Pod count in a Job where the resource request is underestimated, increases drastically the average Pod Completion time as well as skewing the distribution to become more tail-heavy. However, the distribution of Pod Completion Time achieved by CARICO remains consistent as you increase the Job size.

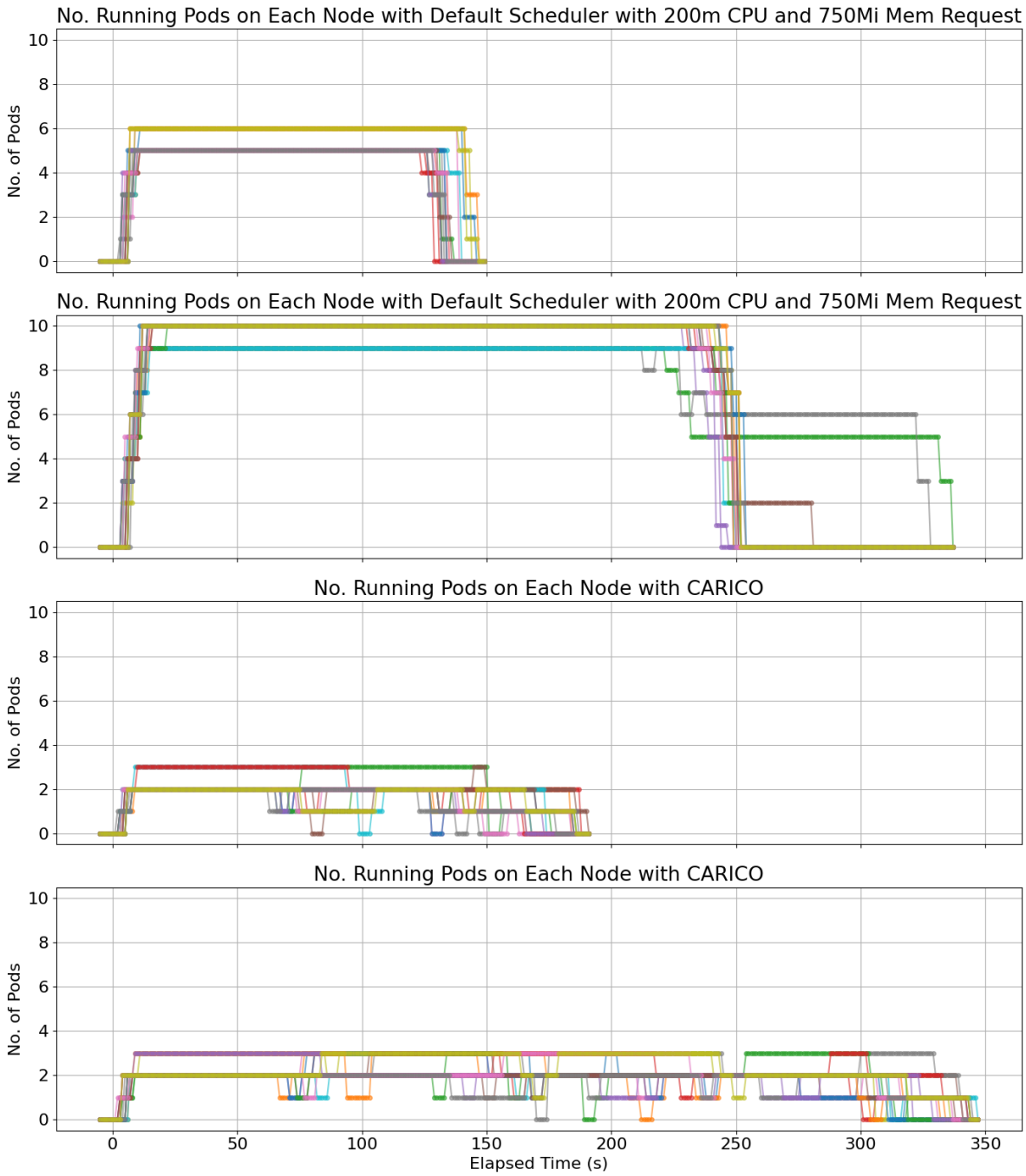


Figure 5.3: The number of ML Pods running on a Node during the execution of a Job with 100 and 200 Pods.

Figure 5.3 depicts the number of Pods running on each Node at a given time. We can see how in the larger ML Job, the default scheduler is able to allocate a majority of the Pods across the Nodes. Each Node advertises ≈ 8GB of memory while each Pod requests 750MiB of Memory. All these Pods end up competing for resources, and the overcontention increases individual Pod completion time. The scheduler still has to wait for Pods to terminate to free up memory. When space does become available, it is because a stampede of Pods have terminated. While, the scheduler was able to now allocate the

remaining Pods, only a few Nodes are used while the rest become idle. This second wave of Pods results in a slower throughput compared CARICO which ensures 1-3 Pods are always running on a Node.

5.4.3 Resource Utilisation

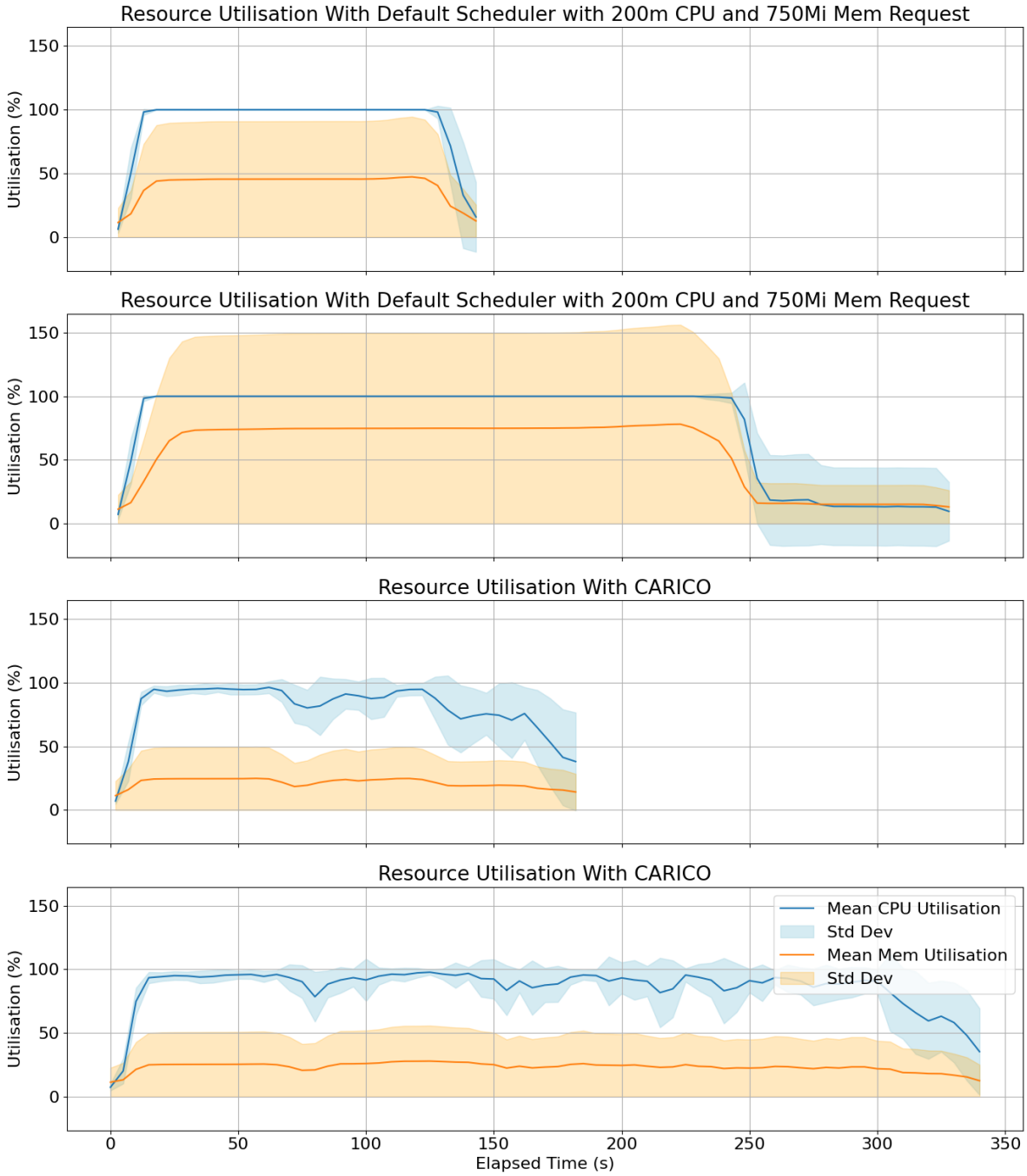


Figure 5.4: The resource utilisation when scheduling a Job with 1000 Pods executing `bpi(2000)`. The top figure gives the resource utilisation when scheduling with the default Kubernetes scheduler. The bottom figure gives the resource utilisation when scheduling with SPAZIO.

Figure 5.3 also helps to explain the resulting resource usage depicted in Figure **??**. While the ML workload is memory-intense, it still contributes significantly to CPU utilisation. As the default scheduler does not actively look at resource utilisation, it is able to allocate Pods to Node's with fully-utilised CPUs. As a result, it achieves a higher memory usage with both Jobs. However, we can also observe the effect of the second wave of Pod allocations during the 200 Pod Job. During the last minute of the Job's executing, the average CPU and Memory utilisation of the cluster have dropped to $\approx 10\%$; barely above their baseline utilisation. On the other hand, CARICO is again limited by the Node's CPU metrics, achieving a lower overall memory utilisation.

5.5 Mixed Workloads

In this experiment, I deployed the two Jobs defined above: a short-lived CPU-focused workload and a longer running ML workload. This experiment evaluates how SPAZIO handles more than one type of workload.

5.5.1 Throughput

To thoroughly evaluate SPAZIO in this scenario, I varied the distribution of Pods from each Job. The observed Job Completions are given in Table 5.6.

Scheduler	**Job Size**		**Job Completion (s)**
	`bpi(2000)`	`ML`	
Default	500	20	105.7 ± 10.6
SPAZIO	500	20	88.7 ± 2.5
Default	250	50	118 ± 4.58
SPAZIO	250	50	115.7 ± 0.58
Default	100	100	149 ± 0.0
SPAZIO	100	100	192 ± 4.0

Table 5.6: 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

The Job Completions from the experiment are given in Table 5.6. CARICO achieves a higher throughput during the 500-20 and 250-50 combined workloads. However, the final 100-100, we see how the limiting throughput is now governed by the throughput achieved with the memory-centric workload.

5.5.2 Pod Completions

Due to Spazio's lackluster throughput, I decided to investigate distribution of Pods across Nodes and how it impacted Pod Completion times.

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

Table 5.7: Pod 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

Table 5.7 gives the Pod Completion Distribution during an execution of the 500-20 Job combination. We can observe that the Pod Completion times for both Jobs is significantly higher with the default scheduler compared to CARICO. This shows how CARICO is still able to achieve a low-tailed Pod Completion time when scheduling workloads with different resource requests.

TODO: SHOULD I ONLY INVESTIGATE WORKLOADS WE HAVE SEEN BEFORE SO THAT I CAN COMPARE THE CHANGE IN DISTRIBUTION

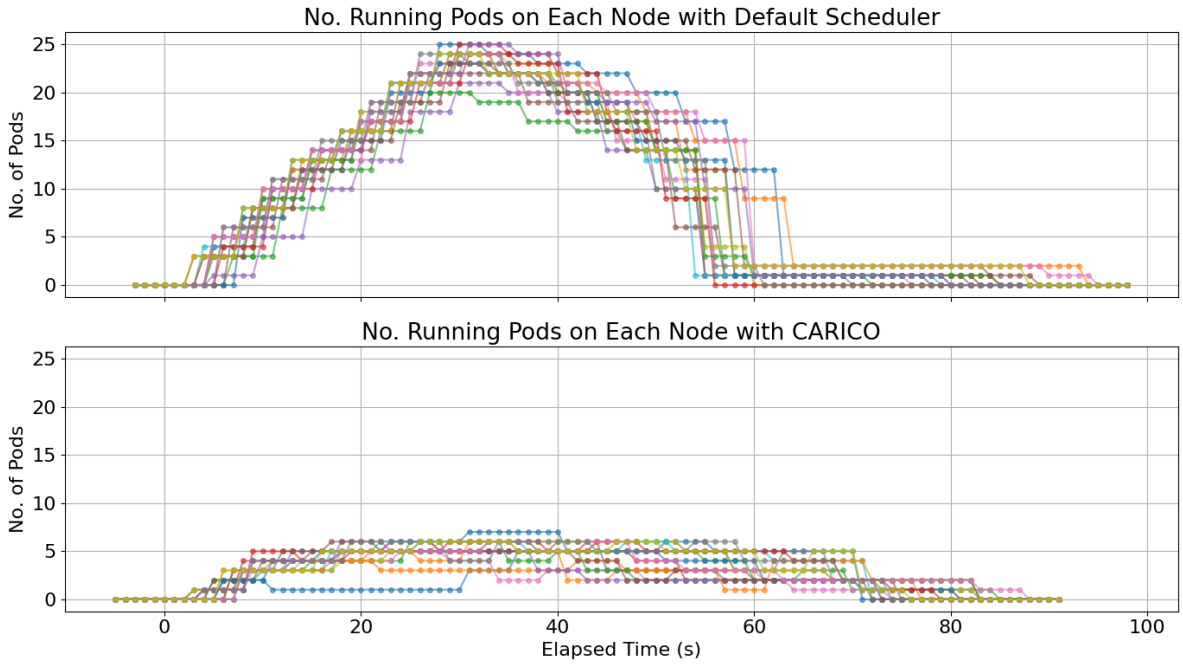


Figure 5.5: The number of Pods running on a Node during the 500-20 Job combination.

Figure 5.5 depicts the number of Pods running on each Node at a given time. Like in 5.3,

the default scheduler ends up with long-running Pods on small percentage of the Nodes in the cluster. The default scheduler see that the Node's have enough advertised capacity, and therefore, allocate all the Pods as they arrive. This again results in a stampede of completions, and only a few Nodes continue to do work. However, CARICO's rate of Pod allocation remains consistent, ensuring that Nodes have a close to constant number of Pods.

5.5.3 Resource Utilisation

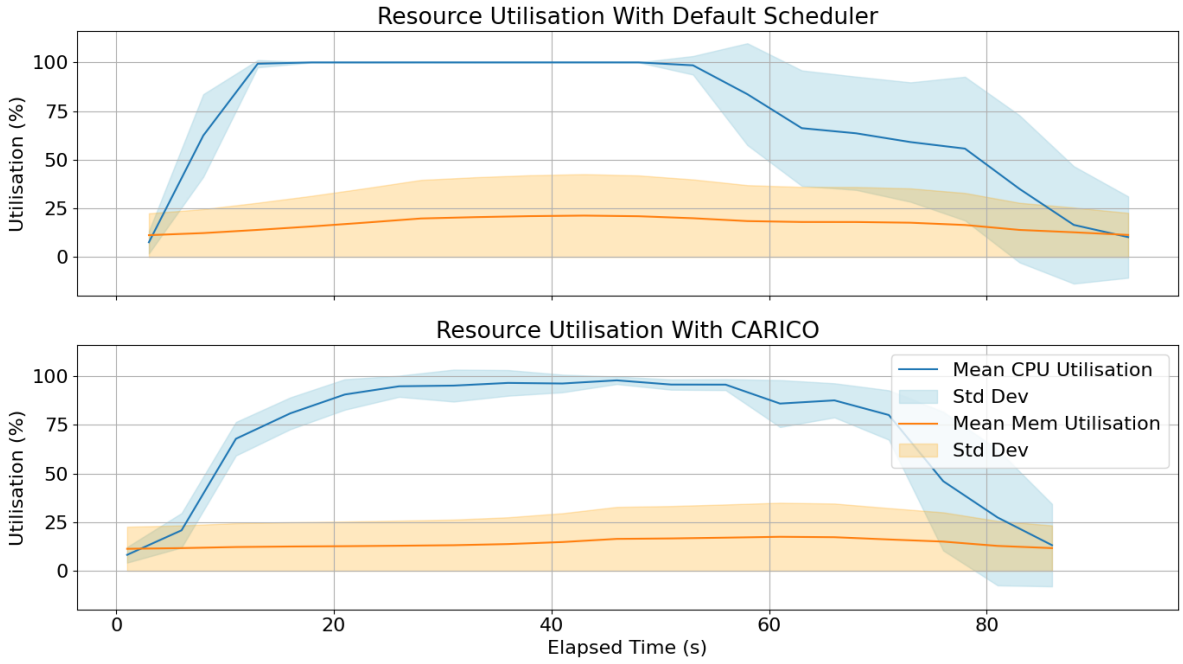


Figure 5.6: The resource utilisation when scheduling a Job with 1000 Pods executing `bpi(2000)`. The top figure gives the resource utilisation when scheduling with the default Kubernetes scheduler. The bottom figure gives the resource utilisation when scheduling with SPAZIO.

Figure 5.6 shows the resource utilisation running the 500-20 combined Jobs. For the same reasons as with the ML workload, the default scheduler ends up with a low resource utilisation for the latter portion of the Job exeuction. Instead, CARICO achieves a more consistent resource utilisations, experiencing less of a fall in CPU utilisation near the end of the Job. In this scenario, CARICO achieves a more efficient allocation of resources.

5.6 Workload Isolation

To evaluate Spazio'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
Spazio	1.00	2.48	6.09	7.82	10.53	17.39

Table 5.8: The distribution of a servers response latency when different schedulers attempt to allocate a 1000 Pod Job across the server.

Table 5.8 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.

5.7 Limitation

Limitations: *In this section, I will go over the limitations of the system. I will highlight how certain metrics like CPU-Utilisations don't give any more information once saturated. I will also have to mention how due to the sub-linear pod completion time, the Kubernetes scheduler is able to achieve higher job throughput by packing more pods into nodes.*

5.8 Summary

Summary: *In this section, I will summarise the results of my evaluation section, highlighting key findings and reasoning.*

Chapter 6

Conclusion and Future Work

6.1 Summary

The goal of this dissertation was to implement a telemetric-only Kubernetes scheduler using the theory behind PRONTO as a spring-board. I based the design of the scheduler behind PRONTO because of its federated nature and its use of contention-based metrics.

This dissertation resembles little to what was in the Project Proposal, and I feel this tells a lot about this project. A telemetric-only Kubernetes scheduler was a novel and inetresting presented an interesting and novel

6.2 Future Work

Bibliography

[1] Latest Kubernetes Adoption Statistics: Global Insights, May 2024. Section: Blog. URL: https://edgedelta.com/company/blog/kubernetes-adoption-statistics.

[2] Google Kubernetes Engine (GKE). URL: https://cloud.google.com/kubernetes-engine.

[3] schaffererin. What is Azure Kubernetes Service (AKS)? - Azure Kubernetes Service. URL: https://learn.microsoft.com/en-us/azure/aks/what-is-aks.

[4] Praneel Madabushini. Leveraging Kubernetes for AI/ML Workloads: Case studies in autonomous driving and large language model infrastructure. *World Journal of Advanced Engineering Technology and Sciences*, 15(1):1044–1052, January 2025. Publisher: GSC Online Press. URL: https://journalwjaets.com/node/480, doi:10.30574/wjaets.2025.15.1.0320.

[5] What is Kubernetes on Edge?, June 2024. URL: https://www.cloudraft.io/what-is/kubernetes-on-edge.

[6] Testimonials - Knative. URL: https://knative.dev/docs/about/testimonials/.

[7] Apache OpenWhisk is a serverless, open source cloud platform. URL: https://openwhisk.apache.org/.

[8] Kirk Pruhs, Jiri Sgall, and Eric Torng. Online scheduling., 2004.

[9] Kubernetes Cost Optimization: Strategies & Best Practices. URL: https://www.cloudbolt.io/cloud-cost-management/kubernetes-cost-optimization/.

[10] Nikhil Gopinath. Bin Packing and Cost Savings in Kubernetes Clusters on AWS. URL: https://www.sedai.io/blog/bin-packing-and-cost-savings-in-kubernetes-clusters-on-aws.

[11] Kubernetes Components. Section: docs. URL: https://kubernetes.io/docs/concepts/overview/components/.

[12] Andreas Grammenos, Evangelia Kalyvianaki, and Peter Pietzuch. Pronto: Federated Task Scheduling, April 2021. arXiv:2104.13429 [cs]. URL: http://arxiv.org/abs/2104.13429, doi:10.48550/arXiv.2104.13429.

[13] Kubernetes Scheduler. Section: docs. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/.

[14] Wojciech Tyczynski|. Kubernetes Performance Measurements and Roadmap, September 2015. Section: blog. URL: https://kubernetes.io/blog/2015/09/Kubernetes-Performance-Measurements-And/.

[15] Apache Hadoop. Apache hadoop yarn. *The Apache Software Foundation*, 2016.

[16] Shalmali Sahasrabudhe and Shilpa S. Sonawani. Improved filter-weight algorithm for utilization-aware resource scheduling in OpenStack. In *2015 International Conference on Information Processing (ICIP)*, pages 43–47, December 2015. URL: https://ieeexplore.ieee.org/document/7489348/, doi:10.1109/INFOP.2015.7489348.

[17] proc_stat(5) - Linux manual page. URL: https://www.man7.org/linux/man-pages/man5/proc_stat.5.html.

[18] Scheduling Framework. Section: docs. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/.

Appendix A

Technical details, proofs, etc.

Appendices are for optional material that is not essential to understanding the work, and that the examiners are not expected to read, but that will be of value to readers interested in additional, in-depth technical detail.

A.1 Lorem ipsum

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.