UNIVERSITY OF
CAMBRIDGE
Department of Computer
Science and Technology

# Hands at the Ready: Bringing Pronto to Kubernetes

Luca Choteborsky

Selwyn College

June 2025

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

Total page count: ??

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

Main chapters word count: 467

Methodology used to generate that word count:

[For example:

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

]

# Declaration

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

**Signed:**

**Date:**

# Abstract

Kubernetes is 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. Existing Kubernetes schedulers can be classified according to the input data they use to inform scheduling decisions: pod-description vs. telemetry. The performance of pod-description based schedulers relies on accurate workload estimations which is not always known beforehand. Meanwhile, telemetry-based schedulers use live information to maximise resource utilisation of nodes and prevent them from becoming overloaded. These telemetry-based schedulers typically use utilisation metrics which can be overly sensitive and generate false notifications of full capacity.

I decided to explore the application of the theory behind Pronto, a federated asynchronous memory-limited algorithm proposed for task-scheduling across large-scale networks of hundreds of workers, to scheduling within Kubernetes. Each individual node updates their own local model based on the workload seen so far (contention-based telemetry) and uses peak-prediction generate a rejection signal which reflects the overall node responsiveness and whether it can accept an incoming task. In addition, aggregating the local models builds a global view of the system. However, as the paper assumes no communication latency and spikes within a Kubernetes node's contention-based metric are not reflective of over-contention, peak-prediction is no longer an accurate indicator.

As a result I propose a novel algorithm, Spazio[1], with the same properties of Pronto while also allowing the existance of communication latency. Nodes in Spazio update their local model with capacity-based metrics and score themselves according to their predicted capacity: how well a nodes current resource usage compliments recent measured workload. Furthermore, Spazio also utilises contention-based metrics to more accurately measure the resource capacity of a node.

To evaluate Spazio, I implement a prototype of Spazio in Kubernetes, evaluating individual components to ensure correctness of behaviour. Finally, I compare Spazio against the industry-standard `kube-scheduler` (the default scheduler of Kubernetes), evaluat-

[1]Spazio: space in Italian

ing Spazio's throughput and its quality-of-service (QoS). From the evaluation, I show how Spazio can achieve high workload-isolation, low latency while achieving comparable throughput compared to Kubernetes.

# Acknowledgements

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

# Contents


**1 Introduction (961) 9**

**2 Background 12**

2.1 Kubernetes Overview . . . 12
2.2 Scheduling in Kubernetes . . . 13
2.3 Analysis of Scheduling Approaches and QoS-Aware Trends . . . 13
2.3.1 Pod Spec-Only Scheduling vs Telemetry-Aware Scheduling . . . 14
2.3.2 Impact on QoS (Latency, Throughput, Fairness) . . . 15
2.3.3 Summary of Related Work . . . 16
2.4 Serverless Workloads and Schedulers . . . 17
2.5 Shortcomings of Common Telemetric Data . . . 17
2.6 Pronto . . . 17
2.6.1 Overview . . . 18
2.6.2 Evaluation . . . 19
2.6.3 Limitations . . . 19
2.7 Summary . . . 21

**3 Design 22**

3.1 PCA using SVD . . . 22
3.2 Local Model . . . 22
3.3 Subspace Merging . . . 23
3.4 Capacity Signal . . . 24
3.5 Reserve Cost and Capacity . . . 25
3.6 Properties . . . 26

**4 Implementation 27**

4.1 System Architecture . . . 28
4.2 Spazio Pod . . . 29
4.2.1 Metric Collection . . . 30
4.2.2 Filtering Metrics . . . 33
4.2.3 Signal Generation . . . 34
4.2.4 Calculating Cost and Capacity . . . 37

4.3 Aggregation Server . . . . . 40
4.4 Scheduler Pod . . . . . 42
4.4.1 Kubernetes Scheduler Plugin . . . . . 42
4.4.2 Filter . . . . . 43
4.4.3 Score . . . . . 43
4.4.4 Reserve . . . . . 43

**5 Evaluation** **44**
5.1 Batch workloads . . . . . 44
5.2 Sensitive Workloads . . . . . 44
5.3 Limitation . . . . . 44
5.4 Summary . . . . . 44

**6 Conclusion and Future Work** **45**
6.1 Summary . . . . . 45
6.2 Future Work . . . . . 45

**A Technical details, proofs, etc.** **48**
A.1 Lorem ipsum . . . . . 48

# 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 [**?** ]. 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 [**?** **?** ]. Furthermore, its versatility extends to supporting machine learning (ML) deployments [**?** ], edge computing [**?** ], and even serverless functions, which are short-lived, event-driven workloads that can be effectively managed and scaled on Kubernetes using frameworks like Knative or OpenWhisk [**?** **?** ]. This broad applicability solidifies Kubernetes' position as a cornerstone of the cloud-native industry.

Task scheduling is the process of organising, prioritising and allocating tasks or activities to resources. Task-scheduling is classified as a NP-hard problem, thus requiring practical solutions to juggle the quality of scheduling decisions and the computation required to reach those decisions; resources spent determining allocations could instead be spent performing the tasks. This balancing act becomes ever more difficult with online schedulers [**?** ]: schedulers which 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. As a result, much research has been focused on finding efficient scheduling algorithms that guarantee solutions as close to optimal as possible.

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. For instance, over-provisioning of resources due to inefficient bin-packing can result in substantial cloud bills. Choosing optimal bin packing strategies could result in up to 66% reduction in bills [**?** ]. Conversely, under-provisioning can lead to performance

degradation, application crashes and costly downtime. Misconfiguration in resource requests and limits are a primary cause of under-utilised resources or idle nodes, and thus higher billing [**?** ].

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, limits or QoS classes, or telemetry based schedulers, which periodically collected live information from each node. As mentioned earlier, pod-description based schedulers rely on accurate resource estimations. Telemetry-based schedulers can circumvent this problem, but rely on metrics that are reflective of a node's true capacity and current resource usage. From my investigation, I show how the popular CPU Utilisation metric can be misleading when detecting if a node is at full capacity.

I decided to explore the application of the theory behind Pronto, a novel federated asynchronous memory-limited algorithm proposed for task-scheduling across large-scale networks of hundreds of workers, to scheduling within Kubernetes. Each individual node updates their own local model based on the workload seen so far (contention-based telemetry) and uses peak-prediction generate a rejection signal which reflects the overall node responsiveness and whether it can accept an incoming task. In addition, aggregating the local models builds a global view of the system. However, as the paper assumes no communication latency and spikes within a Kubernetes node's contention-based metric are not reflective of over-contention, peak-prediction is no longer an accurate indicator.

As a result I propose a novel algorithm, Spazio[1], with the same properties of Pronto while also allowing the existance of communication latency. Nodes in Spazio update their local model with capacity-based metrics and score themselves according to their predicted capacity: how well a nodes current resource usage compliments recent measured workload. Furthermore, Spazio also utilises contention-based metrics to more accurately measure the resource capacity of a node.

To evaluate Spazio, I implement a prototype of Spazio in Kubernetes, evaluating individual components to ensure correctness of behaviour. Finally, I compare Spazio against the industry-standard `kube-scheduler` (the default scheduler of Kubernetes), evaluating Spazio's throughput and its quality-of-service (QoS). From the evaluation, I show how Spazio can achieve high workload-isolation, low latency while achieving comparable throughput compared to Kubernetes.

To sum up, this project makes the following contributions:

- Investigate the feasibility of applying Pronto to Kubernetes scheduling
- Propose a novel scoring algorithm with the same properties as Pronto, while also considering possible communication latency.

[1]Spazio: space in Italian

  - Novel application of FSVD to build a local model of recent resource usage.
  - Adapted the standard SVD subspace merge operation to account for lack of mean-centering.
  - Provide a novel signal that scores a node's predicted capacity according to recent workloads.
  - Provide a method of reserving the signal to account for communication latency.
- Implement a prototype of Spazio to evaluate the strengths and weaknesses of the algorithm.
  - Investigate different metrics to use to build the local model
  - Investigate signal behaviour under different loads
  - Investigate different approaches to estimating node capacity and pod cost
  - Investigate the overall performance of the prototype under different loads to evaluate its performance: throughput, QoS, performance isolation.

In Chapter 2, I go over existing Kubernetes schedulers and why they do not achieve true online QoS (no prior knowledge of pod resource usage and requirement). Furthermore, I conduct a brief investigation into the applicability of Pronto in Kubernetes, highlighting flaws in communication latency assumptions and the behaviour of contention-based metrics in Kubernetes that makes it difficult to perform peak-prediction. In Chapter 3, I propose the novel algorithm, Spazio, providing proofs for correctness while also exploring potential workload scenarios to explain the expected behaviour of the signal. In Chapter 4, I outline the structure of the Spazio prototype, delving into design decisions while performing micro-experiments to evaluate individual components. In Chapter 5, I evaluate the performance of the Spazio scheduler against the standard `kube-scheduler`, showing how it greatly reduces individual pod completion time. Lastly, we conclude our work and propose potential future directions.

# Chapter 2

# Background

## 2.1 Kubernetes Overview

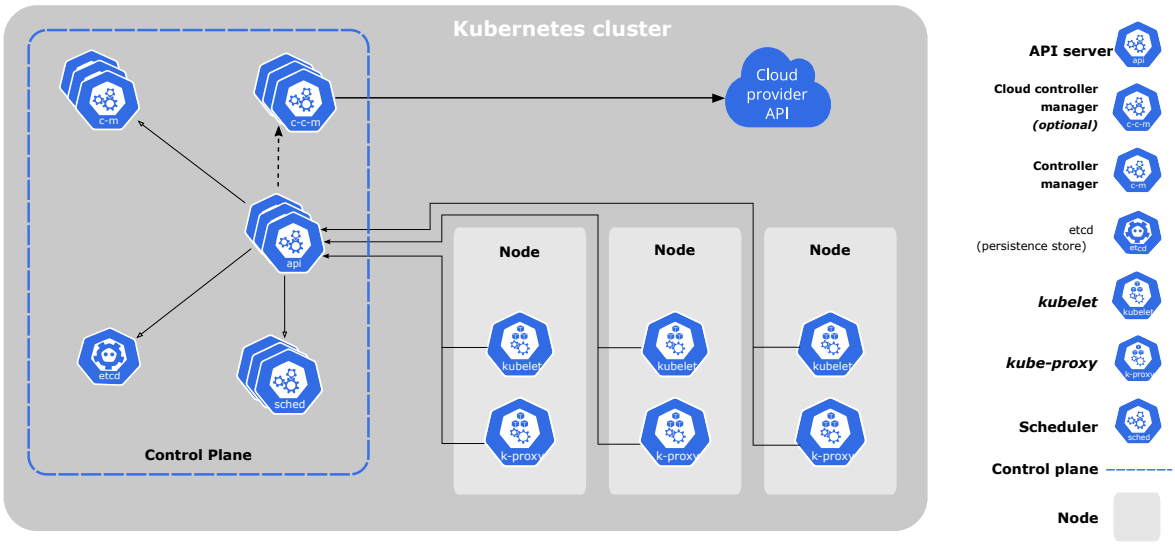



Figure 2.1: The components of a Kubernetes cluster. **Source:** [**?** ]

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. The `kube-server` looks for pods not yet bound to a pod, and assigns each Pod to a suitable node. Each node in the cluster contains a set of components which maintain the running pods and provide the Kubernetes runtime environment. `kubelet` manages Pods and ensures they and their containers are running. The `kubelet` uses a container runtime: software responsible for running containers.

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 Kubernetes objects. Almost every Kubernetes object includes two fields: `spec` and `status`. `spec` is used on creation as a description of the characteristics you want the resource to have, while `status` describes the current state of the object, supplied and updated by the Kubernetes system. These fields are crucial for scheduling pods as they behave like resource constraints, with which the Kubernetes schedulers can use to determine optimal pod placement.

## 2.2 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 repressent a higher abstraction level than a Pod, and the Kubernetes control plane uses the workload's specification to manage Pod objects on your behalf. The built-in APIs for managing workloads include Deployment, StatefulSet, DaemonSet and Job.

When a Pod is created, it initially exists in a "Pending" state - it has been declared by hasn't been allocated to a Node. Kubernetes schedulers watch for newly created Pods that have yet been assigned to a Node, 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".

## 2.3 Analysis of Scheduling Approaches and QoS-Aware Trends

Different Kubernetes schedulers take varied approaches to Pod placement, from simple heuristics to global optimisations and machine learning. This diversity reflects the wide range of workload requirements and cluster scales seen in practice. In the following subsections, we compare how different schedulers leverage different data inputs for decision-making, which fundamentally influences their scheduling strategy and resulting quality of service (QoS) for workloads.

### 2.3.1 Pod Spec-Only Scheduling vs Telemetry-Aware Scheduling

**Pod Description-Based schedulers** rely solely on the information declared in the pod specification and cluster configuration (static data). The default Kubernetes scheduler [**?** ] and similar schedulers like Yunikorn [**?** ] or Volcano [**?** ] fall in this category. They consider factors such as requested CPU/memory, labels and selectors, affinity/anti-affinity rules, and priority. This approach is more reliable for ensuring that the declared needs of a pod are met. For example, a pod will only be placed on a node that has enough free capacity according to a resource requests, and it will honour policies like zone spreading or node affinity.

Benefits of this method are predictability and simplicity: by using static reservations, the scheduler guarantees that if every pod accurately requests its needs, no node will be over-committed at scheduling time. This can protect critical workloads by giving them guaranteed resources (as in Kubernetes Guaranteed QoS class). It also enables fairness policies at high level (e.g. Yunikorn's queues and priorities ensure no team monopolises the cluster).

However, the downside is a lack of visibility into actual runtime conditions - the scheduler is blind to fluctuations in load or hardware state. As a result, static-only scheduling may lead to less optimal placements: for instance, a new pod might be placed on a node that technically has room (per requests) but is currently experience high CPU utilisation from other processes, leading to contention. Meanwhile, an underutilised node could be ignored because its capacity appears full due to over-requested resources or sticky allocations. In summary, purely `spec`-based scheduling treats the cluster as a static partitioning of resources; it ensures each workload's declared needs are met, but it can't respond to real-time performance variations.

**Telemetry-Based schedulers**, incorporate live metrics and feedback from the cluster's current state to make more informed decisions. These schedulers actively monitor CPU usage, memory pressure, I/O rates, cache misses, network latency, etc., either via the Kubernetes Metrics APU, Prometheus, or custom telemetry pipelines. Examples include Intel's Telemetry Aware Scheduling [**?** ] (which uses fine-grained platform metrics like LLC cache hit rates and CPU temperature) and the Trimaran plugin suite for Kubernetes [**?** ] that uses real-time utilisation stats.

By using this data, telemetry-driven scheduling can avoid placing pods on nodes that are struggling or overloaded in real-time. For instance, a telemetry-aware scheduler might detect that Node A's CPU is 90% busy and Node B's is 20% and therefore prefer Node B for a new pod - even if, by static allocation, both nodes had equal allocable capacity remaining. This dynamic adaptability directly improves QoS: workloads get more stable performance because they are less likely to be scheduled to an over-taxed machine.

Moreover, telemetry-based rules can target specific QoS concerns: Intel's TAS can steer latency-sensitive network functions away from nodes with high cache thrashing, preventing unpredictable latency spikes [**?** ]. The benefits of telemetry awareness is higher efficiency and resiliency - clusters can run closer to their true capacity (Trimaran's TargetLoadPacking can pack workloads until it observes near-saturation) but back off when needed to maintain performance. It also helps in maintaining SLAs by reacting to early warning signals before they become failures.

The challenge, however, is complexity: collecting and reacting to metrics adds overhead and potential instability (the scheduler decisions are now as dynamic as the metrics themselves). Careful design is required to smooth out decisions. telemetry-based scheduling brings an automated, feedback-driven approach that can greatly boost performance and utilisation, but it must be tuned to avoid thrashing and to scale metric collection overhead.

**Hybrid Approaches** combine both static pod descriptions and dynamic telemetry to get the best of both worlds: guarantee the fundamental resource requests/constraints from pod specs while using telemetry to fine-tune the choice among feasible nodes. For example, Firmament (via Poseidon) [**?** **?** ] respects each pod's requirements, but when multiple multiple pods are viable it uses a global optimisation incorporating utilisation metrics to choose the optimal placement. Koordinator [**?** ] similarly uses class-based static partitioning (dedicating some portion of resources or priorities to LC vs. best-effort pods) and then uses live node feedback to adjust placements and even migrate workloads at runtime. By blending static and dynamic data, hybrid scheduling strategies can enforce high-level policies (like fairness, or priority), and ensure those policies aren't undermined by unforeseen runtime conditions. The outcome is often superior QoS in multiple dimensions, but come at the cost of the most complexity.

### 2.3.2 Impact on QoS (Latency, Throughput, Fairness)

The choice of data inputs in scheduling directly impacts various QoS metrics:

**Latency and Response Time:** Schedulers that can account for real-time load significantly reduce latency for individual requests. A telemetry-aware scheduler will avoid placing a latency-critical pod onto a node with high CPU or IO wait, thereby ensuring that the pod can get CPU time quickly. In contrast, a purely static scheduler might inadvertently co-locate many heavy pods on the same node (if their requests were low but actual usage is high), leading to contention and higher latency for all pods there. Thus telemetry-driven or hybrid scheduling is better for meeting low-latency SLAs.

**Throughput and Completion Time:** For batch workloads, scheduling strategies affects job completion times and overall throughput of the cluster. Volcano's [**?** ] gang scheduling ensures all tasks of a parallel job start together so the job can finish as quickly as possible with no stragglers waiting for resources. This improves throughput of a batch

workload and avoids resource waste. Similarly, a scheduler using telemetry can balance load by preventing certain nodes from becoming bottlenecks. However, using utilisation metrics does not always guarantee optimal throughput. For example, when 5 pods are running on a node, the node may advertise 100% CPU usage. When running an additional pod, CPU utilisation will remain at 100% but the resulting pod completion times may increase sub-linearly with the number of pods and thus result in higher-throughput. This example highlights how utilisation metrics can be misleading when detecting pod contention.

**Fairness and Resource Isolation:** Fairness can be considered from multi-tenant perspectives (each user gets a fair share) and from a workload isolation perspective (no noisy neighbour dominates a node to the detriment of others). Pod spec schedulers address the former via quotas, priority classes, etc., and the latter only indirectly (e,g, by requiring all pods to declare resources, which if done correctly prevents one pod from consuming what another needs). Telemetry-based approaches tackle isolation by detecting noisy-neighbour effects - for instance, Koordinator monitors interference metrics and will separate batch jobs from latency-senesitive ones if the interference crosses a threshold.

**Resource Utilisation:** Spec-only scheduling often relies on over-requesting (adding safety margins to resource requests) which can leave nodes underused. Therefore, these schedulers might not be able to completely use all of a node's capacity because they can't safely pack workloads beyond the declared request. If a user under-requests, it can lead to CPU throttling and OOM kills and degrade application QoS. On the other hand, telemetry-aware scheduler can confidently increase utilisation up to a safe limits - for example, Trimaran's TargetLoadPacking will keep filling a node until it observes the node is about to hit a predefined utilisation limit.

DO I NEED TO INCLUDE CITATIONS FOR SYSTEMS I HAVE ALREADY REFERENCED

### 2.3.3 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 Serverless Workloads and Schedulers

**Serverless Workloads:** *This section I will introduce the concept of serverless functions and how it is growing industry: adapted in multiple cloud services. Explain the typical workloads: resources, duration and actions. Explain why containers are perfect for these workloads due to their fast start up time compared to booting up VMs.*
*In this section I will also review existing Serverless schedulers. I will consider what type of scheduling information these schedulers will use and whether these schedulers consider very short workloads such as less than 4 minutes*

## 2.5 Shortcomings of Common Telemetric Data

Many telemetry-aware Kubernetes schedulers [**?** **?** **?** **?** ] use utilisation statistics, such as % CPU used, % memory used, as these metrics are available in most computer systems and can be efficiently collected. While they are easy to process, high values do not always signify over-contention. During the implementation of the Spazio prototype, I investigate how CPU Utilisation corresponds to Job Completion time. From the investigation, figure **??** shows that throughput continues to increase as we increase the number of concurrently running pods even after CPU utilisation reaches 100%. Therefore I conclude that CPU utilisations is not representative of the true CPU capacity. **I only investigated this on my cluster of VMs. This may just be a property of VM machines**.

Few Kubernetes schedulers use deeper contention indicators (e.g. cache pressure, memory pressure and CPU throttling) [**?** **?** ]. I believe that metrics can be used to better inform scheduling decisions compared to the standard utilisation metrics.

## 2.6 Pronto

Pronto is a federated asynchronous, memory-limited algorithm proposed for online task-scheduling across large-scale networks of hundreds of workers. Each individual node updates a local model based on telemetric data and generates a rejection signal which reflects the overall node responsiveness and whether it can accept an incoming task. In addition, aggregation is performed on the local models to build a global view of the system. Pronto takes the existing CPU Ready metric generated by the VMware vSphere and proposes a novel method that uses it as a task scheduling predictor. I believe that Pronto's use of contention indicators and federation of individual node models can be used to implement a telemetric-based QoS scheduler.

### 2.6.1 Overview

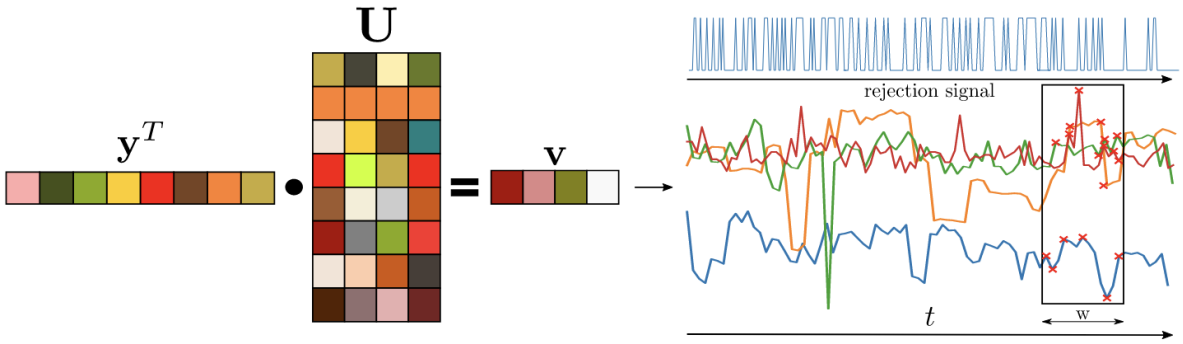



Figure 2.2: Projection of incoming $y \in \mathbb{R}^d$ onto embedding $U \in \mathbb{R}^{d \times r}$ producing $R$ projections in $v \in \mathbb{R}^{1 \times r}$. Projections are tracked over time for detecting spikes which form the basis of the rejection signal. The sliding window for spike detection for each projection is of size $w$ also shown in the figure.

PCA is a powerful dimensionality reduction technique, used to simplify complex, high-dimensional datasets while retaining the most information. Pronto use Federated PCA (a privacy preserving distributed approach to PCA) to extract features which represent the "directions" with greatest variance. Pronto can then project telemetry data onto these vector directions to predict future spikes which indicate resource contention.

Pronto makes use of the following property: The PCA of a matrix can be calculated by mean centering the data, performing SVD on the resulting matrix to give $U$, the directions of the principle components, and $\Sigma$ the variance along these directions. As Pronto aims to be memory-limited, it can't store all of the collected telemetry and must instead take a streaming approach. It periodically measures the CPU Ready metric at a given frequency. To perform FPCA, every new batch of $b$ samples is iteratively merged into the latest subspace using Incremental-SVD. In addition, to perform global model aggregation, SVD is performed on the concatenated bases of two local models. Pronto can further reduce the required computation by using Low-Rank approximations, reducing the dimensionality to $r$ where $r < m$. It uses the assumption that the first $r$ eignenvalues contain the majority of the information of the dataset.

Once a node has a local model, it can periodically generate a signal by projecting the latest telemetry data onto the principle component subspace $U$ and identifies all the spikes in the projections. If the weighted sum of the spikes exceeds a threshold, then a rejection signal is raised which indicates that node is experiencing performance problems and a job should not be scheduled.

### 2.6.2 Evaluation

In the Pronto paper, a scheduler's effectiveness was quantified by its ability to raise a rejection signal that precedes a CPU-Ready spike within a pre-defined window, while minimising the number of false positives that do not precede CPU-Ready spikes. Protono was compared against non-distributed dimensionality-reduction methods [], and was shown to predict CPU-Ready signals effectively while keeping false-positives low. Pronto's better performance over non-distributed strategies implies that there were correlations between job's resource usages on different nodes at the same point in time. As a result, FPCA could use telemetry across multiple nodes to more acurrately compute the principle components and thus give better CPU-Ready predictions.

These properties make Pronto an attractive technique that could be incorporated into scheduling within a Kubernetes cluster.

### 2.6.3 Limitations

#### 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 [**?** ][].

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 [**?** ]. 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.

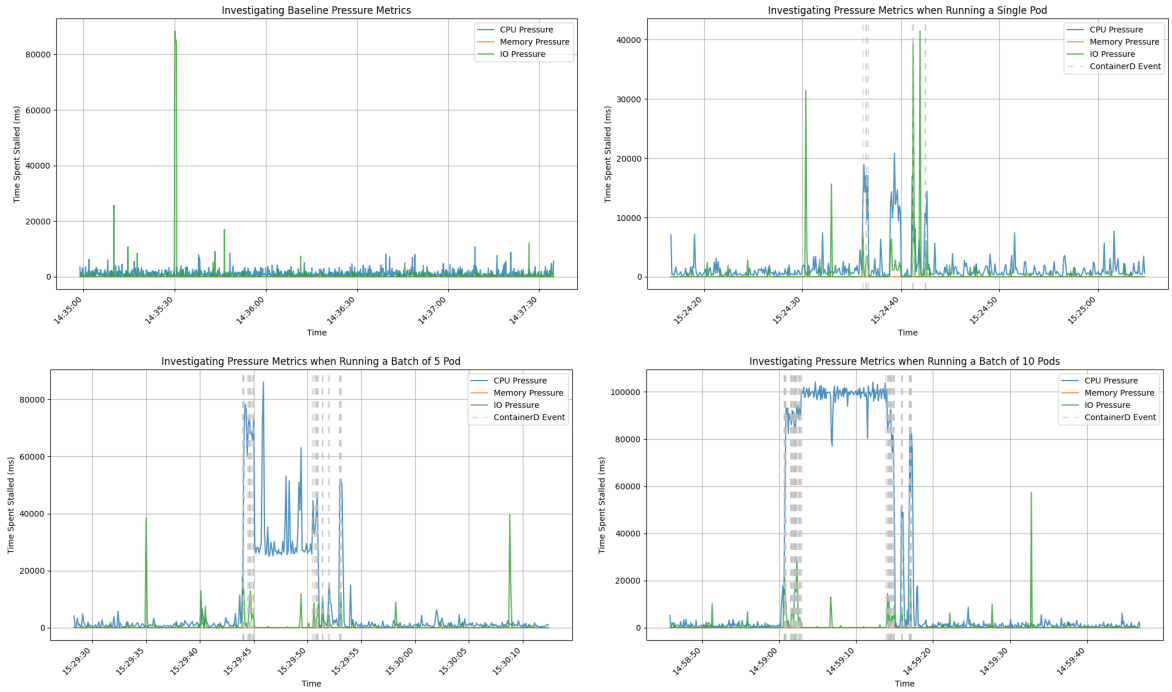



Figure 2.3: Measurements of `/proc/pressure/ total` value under different loads. The container runtime results in spikes no matter the workload.

From figure **??**, 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.

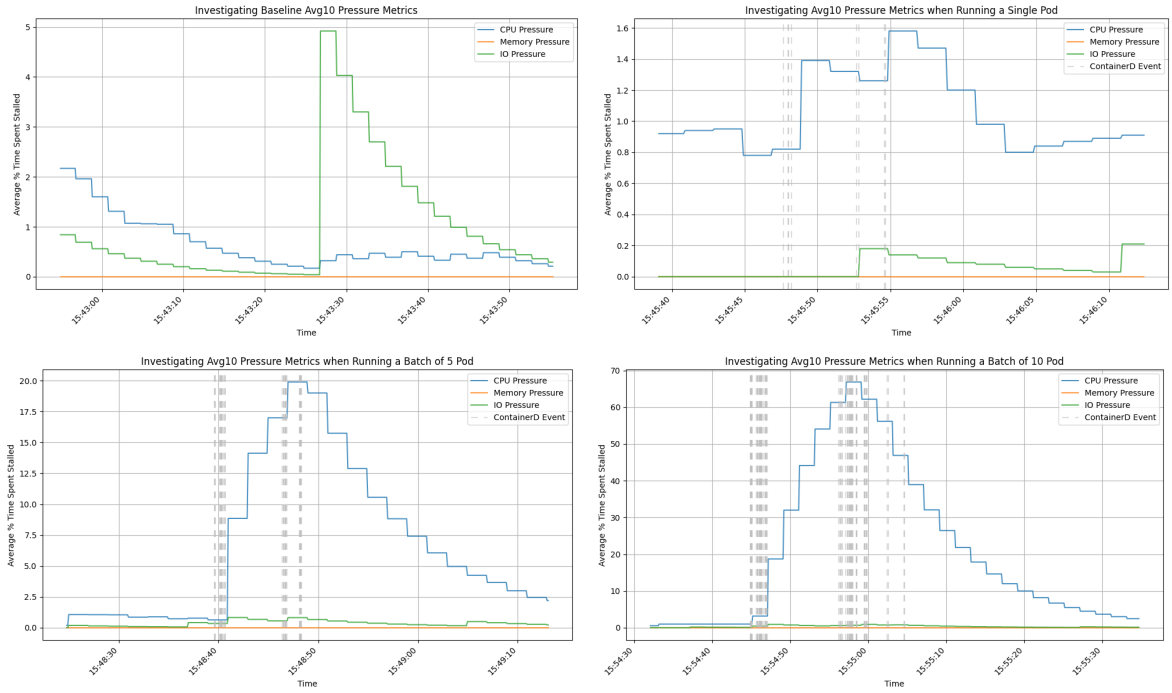



Figure 2.4: Measurements of `/proc/pressure/ avg10` value under different loads. The container runtime results in spikes no matter the workload.

Figure **??** 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.

## 2.7 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 $\mu_i$ 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 $i^{\text{th}}$ 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 $\Sigma$ 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 $j^{\text{th}}$ 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 $a_j$s. 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 $\sigma_i$ corresponds to $\text{Var}_i$, with each iterative-SVD, we increase the eigenvalues of $\Sigma$. This is problematic as exploding values can result in more expensive matrix operations. In addition, both Pronto and Spazio use $\Sigma$ as some sort of weighting. Unreasonably large

$\sigma_i$ 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, by scaling the concatenated matrix using the method above, can upper and lower bound the growth of $\Sigma$. 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: $\forall 1 < i < n, k \neq 0 : \exists j \in 1, \ldots, n\text{such that}(ku_i)_j \leq 0$. 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 $\sigma_1$ from the latest $U$ and $\Sigma$, 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 $\sigma_1$ 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.

## 3.5 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 $\text{avail.}_n$ to a central scheduler. This scheduler also tracks each Node's reserved amount as $\text{reserved}_n$. 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 $\text{reserved}_n$ by 1 for that Node. Once a scheduled Pod is no no longer in the Pending state, the central scheduler decrements $\text{reserved}_n$ 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.

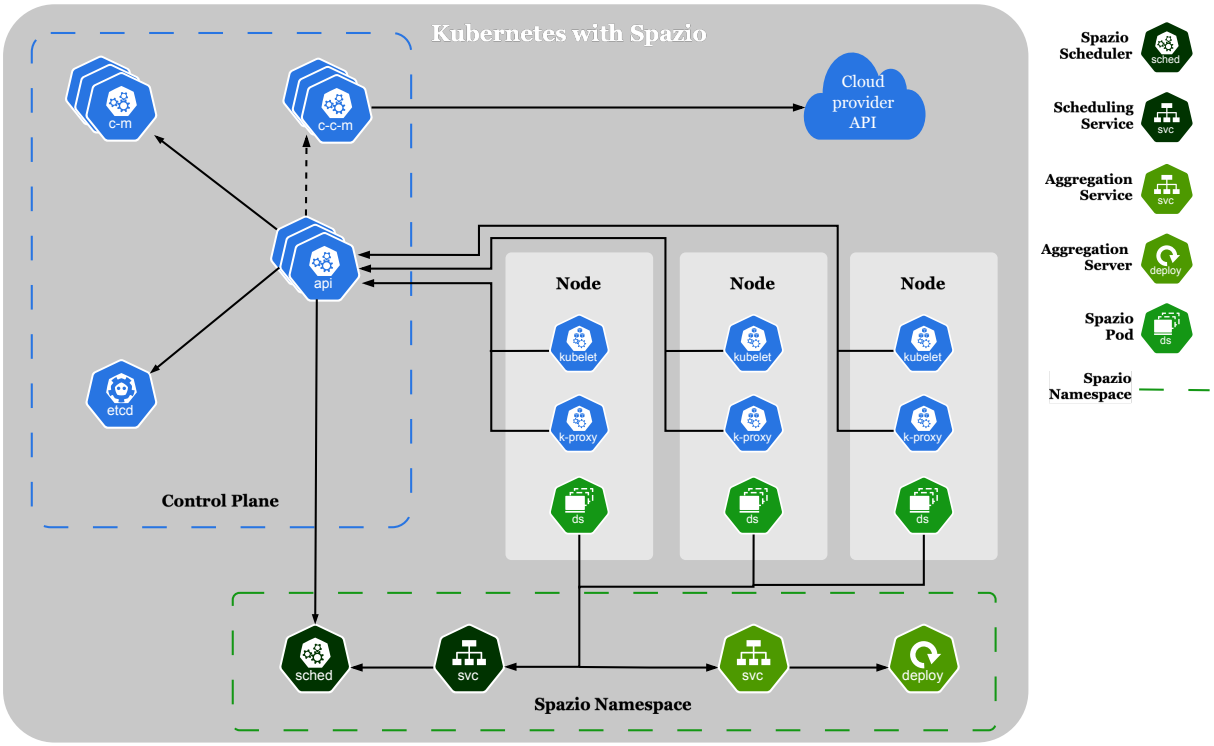
**Continuous Value** Pronto is a binary signal, which makes difficult to score Nodes. Spazio's avail. $\in \mathbb{R}$, allowing Nodes to be filtered and scored against each other.

**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 **??**):

- 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

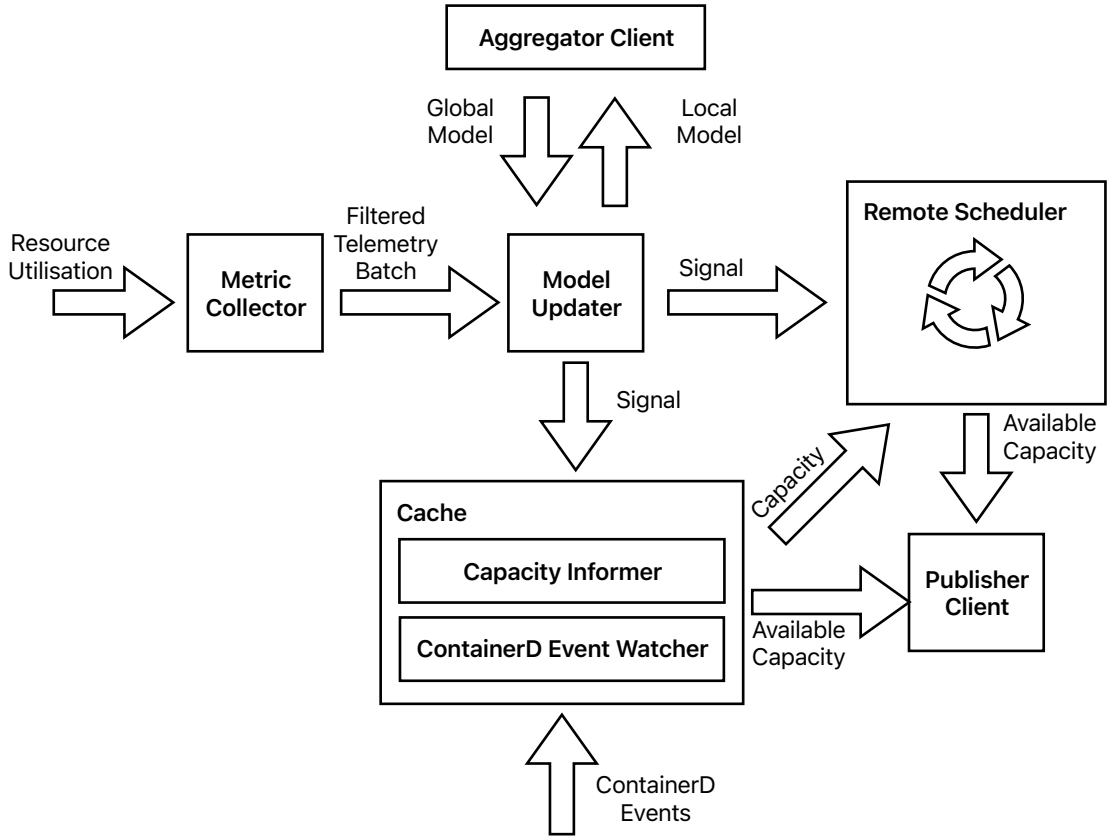



Figure 4.2: Core components within the Spazio pod

As mentioned in section **??**, 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 [**? ?** ]. 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 [**?** ]. 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:

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

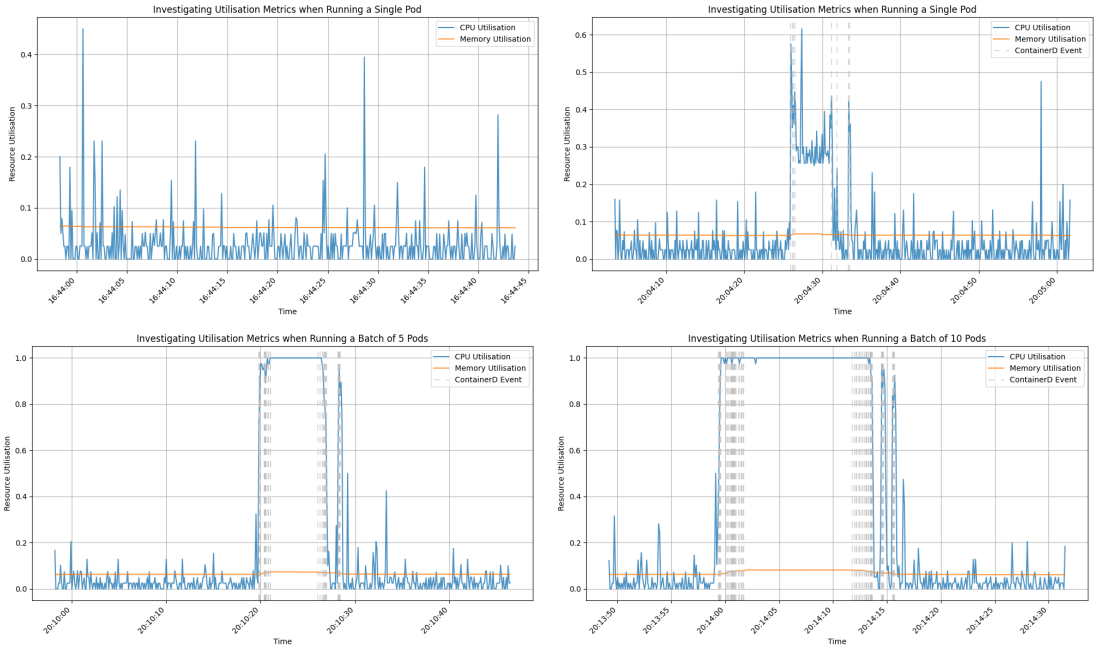



Figure 4.3: In this figure we sample CPU and memory utilisation from values of `/proc/stat`, `/proc/meminfo` at 10Hz during various Kubernetes workload.

To evaluate whether utilisation metrics were suitable for Spazio, I measured their output under different different workloads. Figure **??** presents the metrics behaviour when running different Job sizes of `pi-2000` Pods. As the cpu-intense workload involves calculating $\pi$ 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 $\approx$ 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.

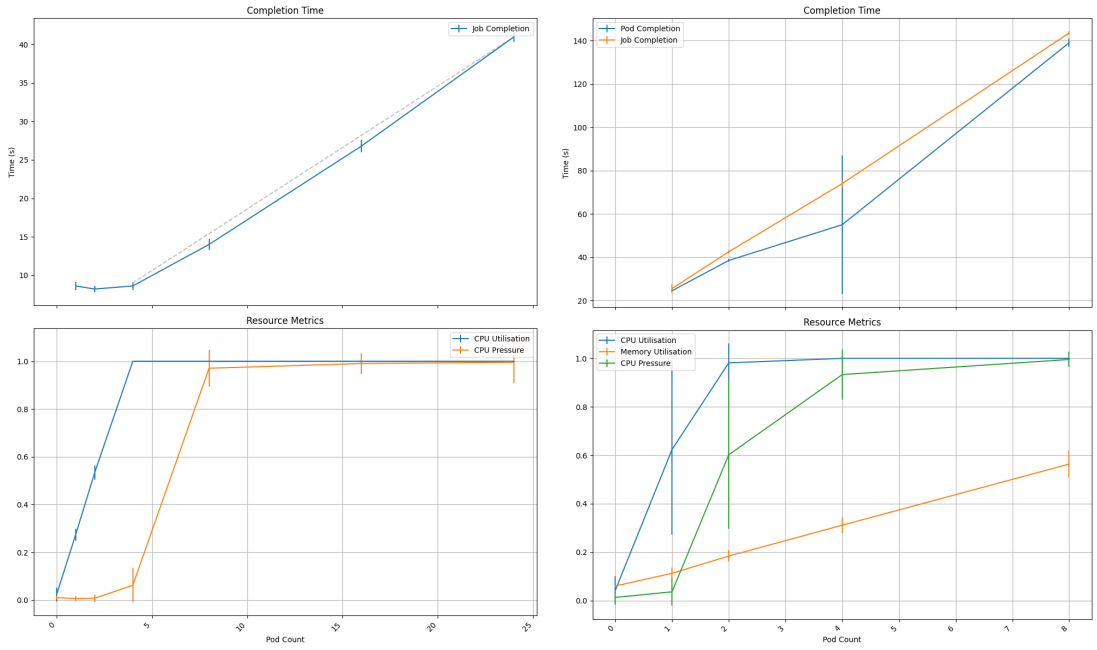



Figure 4.4: In this figure we sample CPU and memory utilisation from values of `/proc/stat`, `/proc/meminfo` at 10Hz during various Kubernetes workload.

As a result, I decided to further investigate this phenomenon. Figure **??** 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.

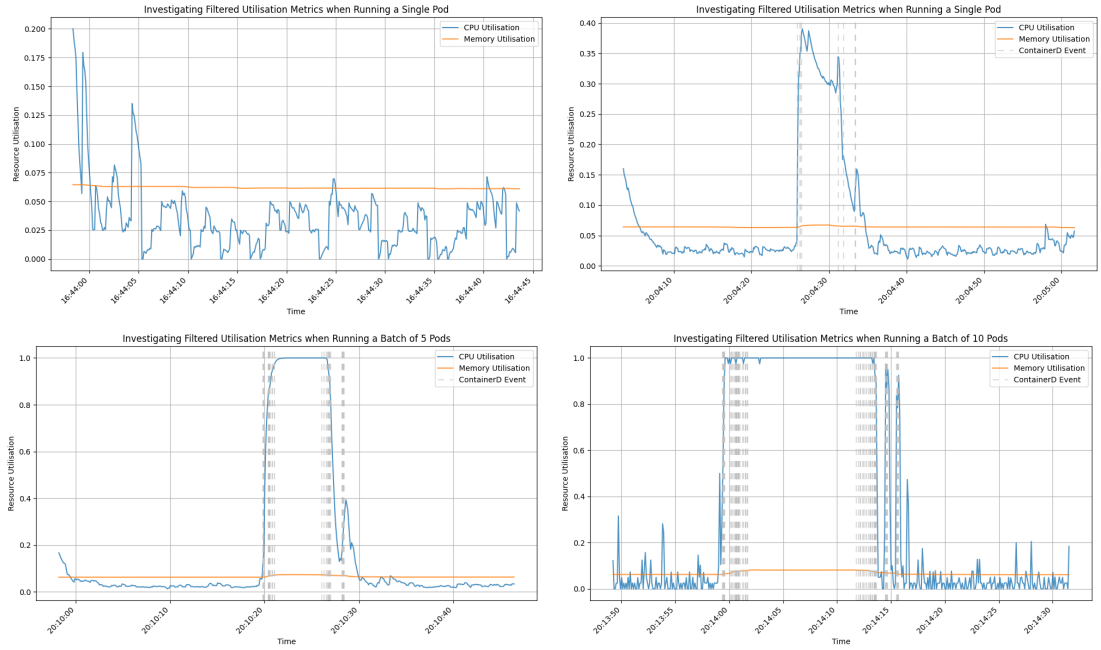



Figure 4.6: This figures shows the smoothed metrics under different workloads.

Comparing Figure **??** with **??** 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 **??**. The collected measuremets are shown in figures **??** and **??**

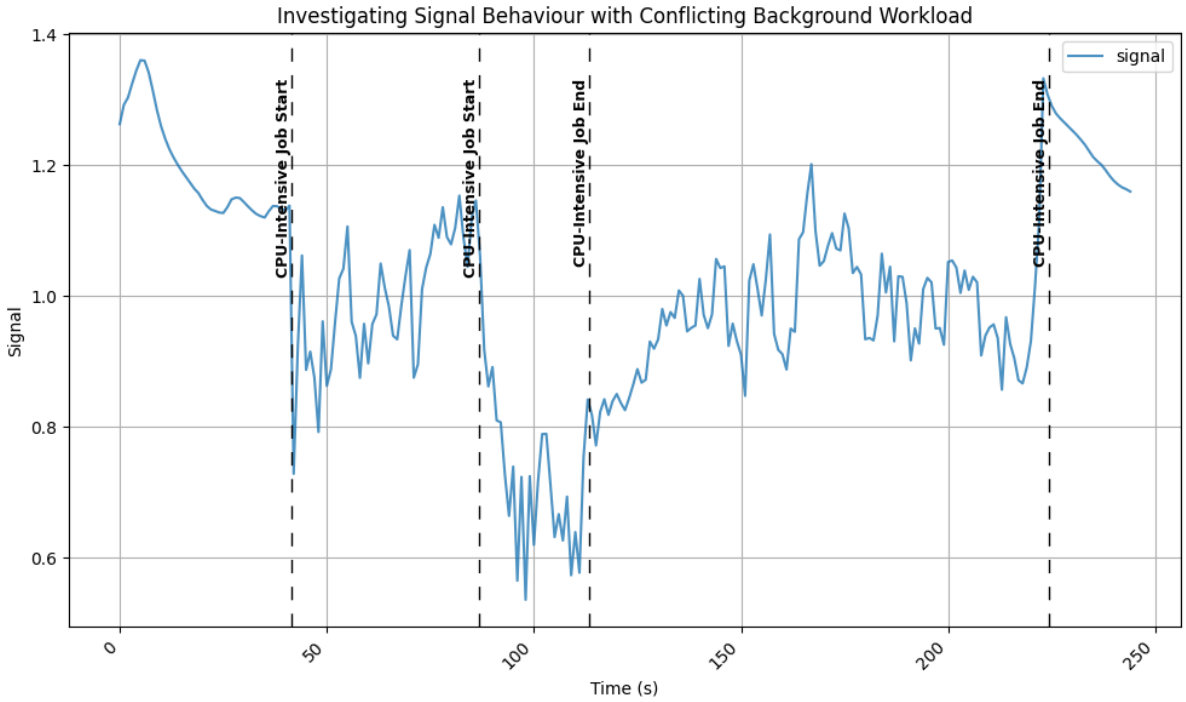



Figure 4.7: The calculated capacity signal of a Node running `ng-stress --cpu=8 --cpu-load=25`. While running this workload, 1000 Pods running `bpi(2000)` was scheduled across surrounding Nodes.

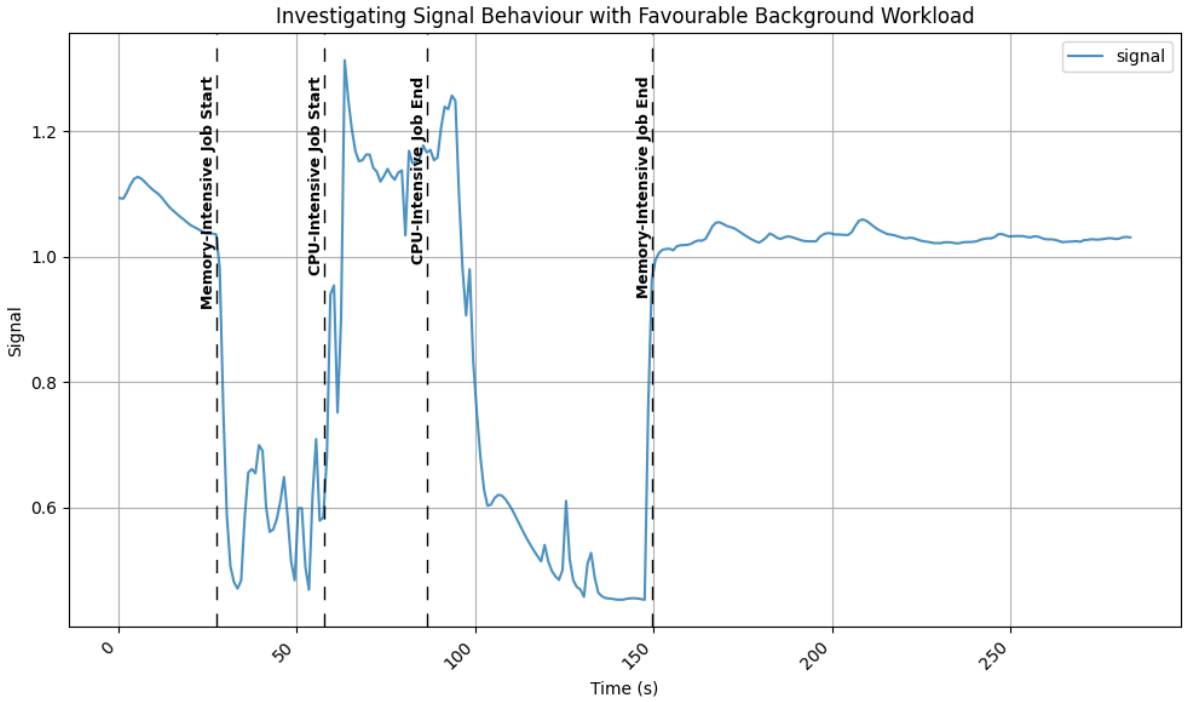



Figure 4.8: The calculated capacity signal of a Node running `ng-stress --vm=4 --vm-bytes=4G`. While running this workload, 1000 Pods running `bpi(2000)` was scheduled across surrounding Nodes.

Figures **??** and **??** demonstrates how a Node's capacity signal reacts to changes in surrounding workloads when experiencing different resource usage. In Figure **??**, 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 $\sigma_1$ value. As the measured Node's directions of the measured Node's resource usage and expected workload are still CPU-focused, the increased $\sigma_1$ value results in a lower $k$, and thus a lower capacity signal.

In Figure **??**, 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 **??** 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 **??**.

### 4.2.4 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: 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.

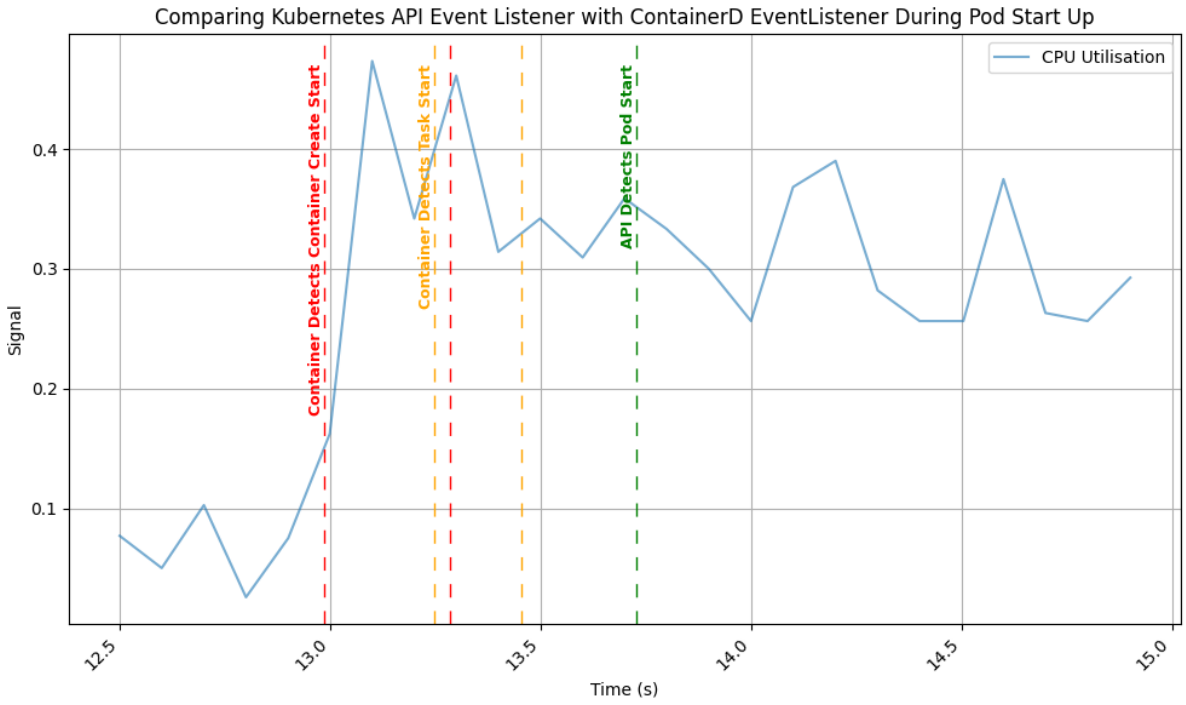



Figure 4.9: When different event listeners detected the creation of a Pod.

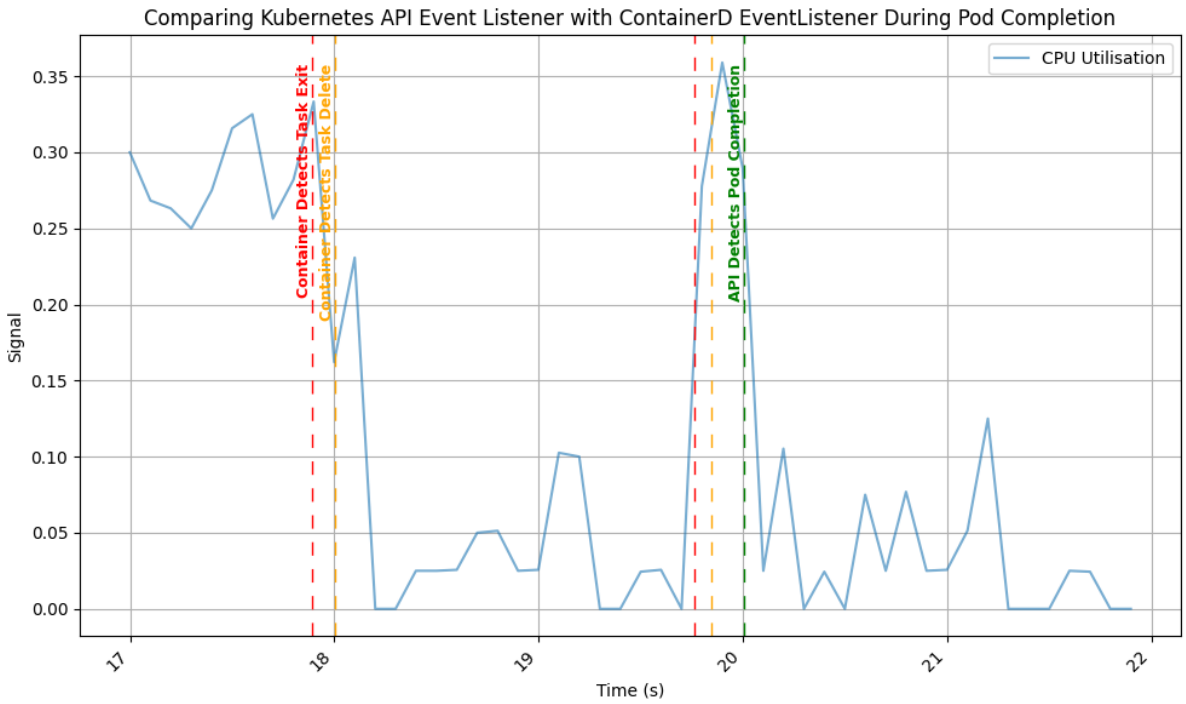



Figure 4.10: When different event listeners detected the completion of a Pod.

As shown in figures **??** and **??**, 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 **??**, 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.

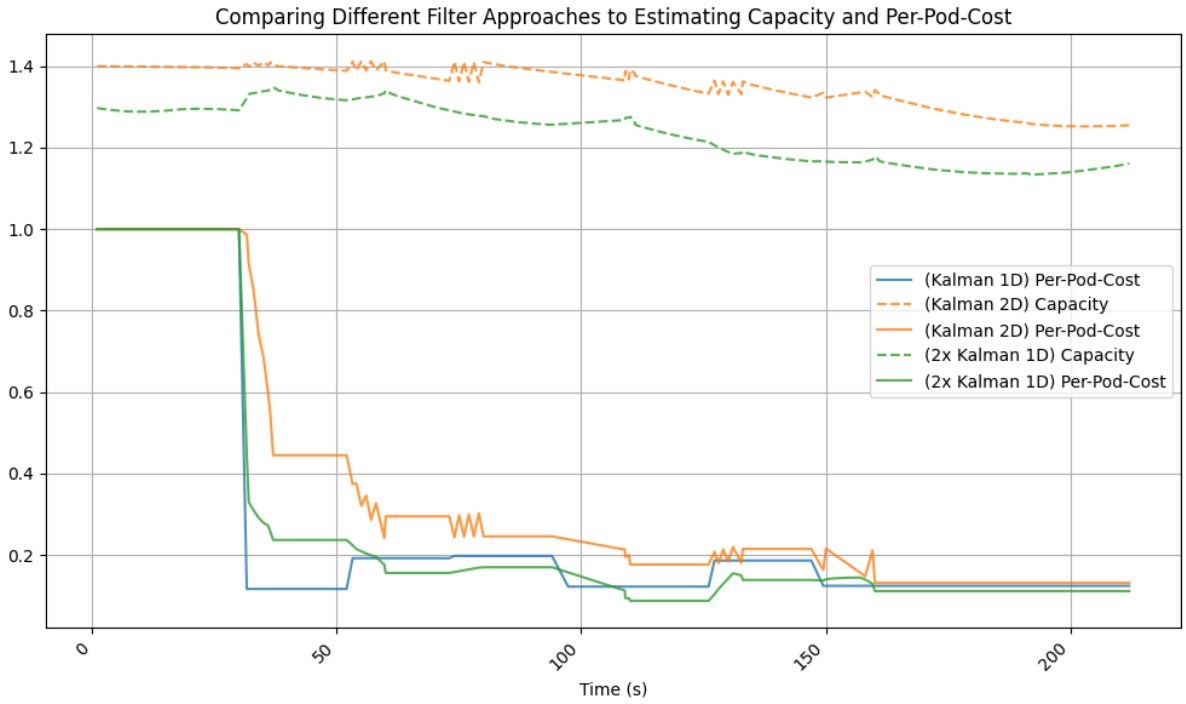



Figure 4.11: The estimates of the Kalman filters when Node experiences variable-sized bursts of `bpi(2000)` Pods.

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

of various sizes. This is depicted in Figure **??**. 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

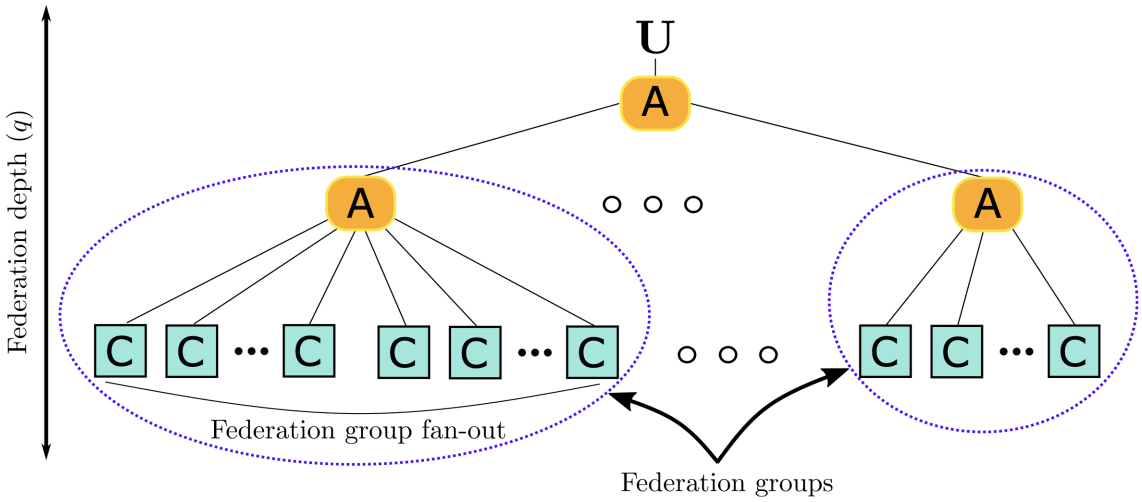



Figure 4.12: How local models are aggregated in Pronto. Dedicated aggregator nodes propagate the updated subspaces until the root is reached.

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 **??**. 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.

The Aggregation Server Pod

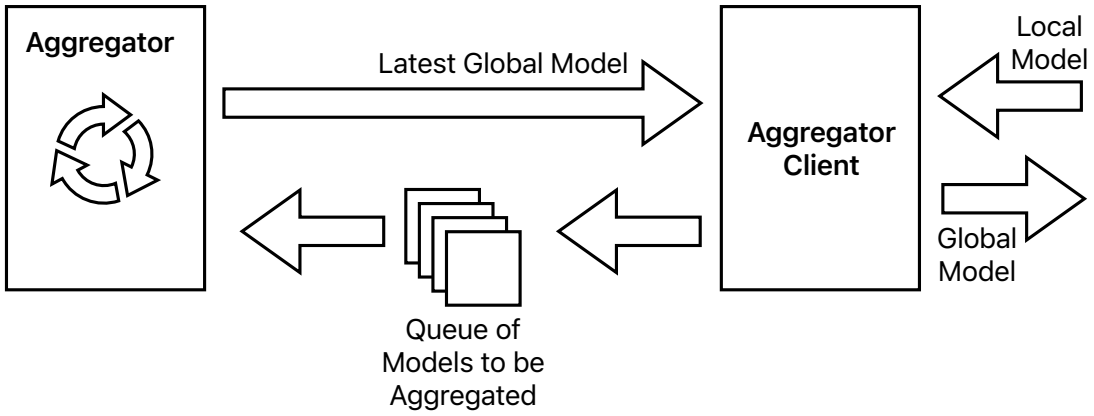



Figure 4.13: Core components within the aggregator pod

As mentioned earlier in section **??**, the aggregator trades . On aggregation request it returns the latest global model and enqueues the local model it received. Another worker thread running in the background dequeues local models and merges them into the global model.

Explain why I use a flat style aggregation rather than a hierarchical design. Core idea is the choice of efficiency and lateny vs consistency.

The aggregation of local models to produce a global model also uses the same iterative-SVD as defined in section **??**. In addition, instead of using a hierarchical aggregation system, I use flat aggregation service - all aggregation requests are handled by a single node. In the original Pronto paper, the authors assumed that there was no communication latency. However, in a real-world cluster this assumption does not hold. Adding additional aggregation layers would increase the overall aggregation latency.

Instead with a aggregation server, I can reduce the overall latency of aggregation requests. To reduce request latency further, actual model aggregation is not performed on the request's critical path. On receipt, the local model is enqueued to be aggregated and the latest global model is returned. A background thread iterative-SVD merges the queued local models into the global model. In summary, this system trades consistency for latency and throughput, which becomes a dominant factor when scaling to hundreds of nodes.

## 4.4 Scheduler Pod

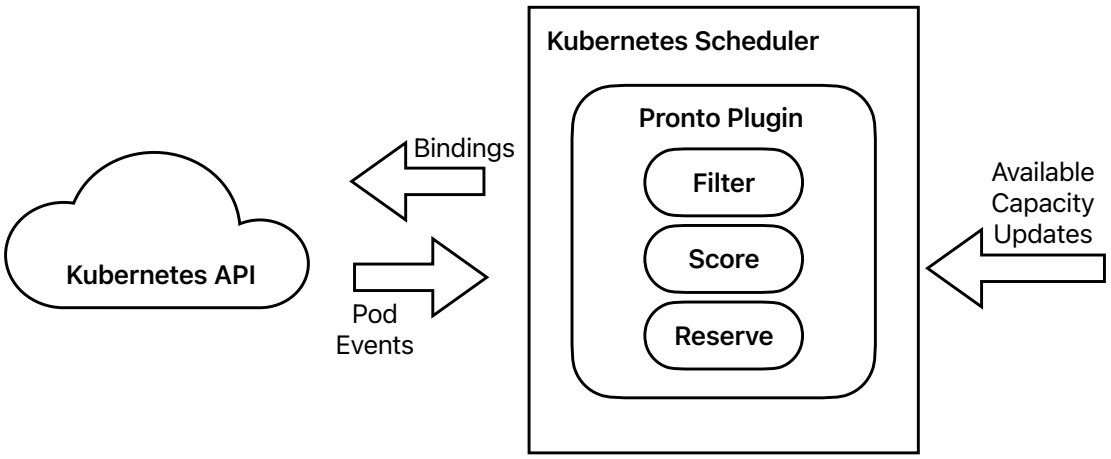



Figure 4.14: Core components within the central scheduler pod

### 4.4.1 Kubernetes Scheduler Plugin

This section gives the motivation for using a Kubernetes Plugin. Core Ideas: ease of use, access to efficient data structures.

I decided to implement the scheduling component as a Kubernetes Framework Plugin. The standard Kubernetes scheduler exposes a series of extension points, which allows users to define custom functions within the scheduling cycle. A Scheduler Plugin is a more favourable approach as it allows me to use an existing system designed to operate at an industrial level. The Pronto Plugin implements the following extension points:

- : Filter - this function filters out any nodes based on capacity available$-$capacity reserved $<$ $\epsilon$.
- Score - this function assigns a score based on the available capacity. The greater the available capacity, the greater the score and thus the more likely it is to be chosen for pod placement
- Reserve - once a node is chosen to host the pod, the reserve function records the pod's name and increments the node's reserved capacity.

This pod also has a Kubernetes API Pod Event listener which listens out for pods that have transitioned from the Pending status. For each event that this occurs, it checks if this pod was scheduled by the scheduler and decrements its value from reserved.

### 4.4.2 Filter

This section explains the Filter function

### 4.4.3 Score

This section explains the Score function

### 4.4.4 Reserve

This section explains the Reserve function

# Chapter 5

# Evaluation

## 5.1 Batch workloads

**Batch Workloads:** *In this section, I will compare Pronto against the Kubernetes scheduler when running batch jobs. In this section I will explore both CPU, Memory-Intensive and Multi-Resource workloads. I will evaluate the throughput, completion time and response time.*

## 5.2 Sensitive Workloads

**Sensitive Workloads:** *In this section I will investigate how this scheduler protects sensitive workloads. I will have a server running in the background on a node and will schedule a batch of jobs. I will measure how a servers response time changes.*

## 5.3 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.*

## 5.4 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

## 6.2 Future Work

# 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.