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: 39

Main chapters (excluding front-matter, references and appendix): 30 pages (pp 7–36)

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

Pronto is a federated asynchronous, memory-limited algorithm proposed for online task-scheduling across large-scale networks of hundreds of workers. Each individual node to update their own local model based on the workload seen so far and 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. Kubernetes is an existing open-source container orchestration system that automates the deployment, scaling and management of containerized applications. It can run production workloads, having becoming the foundation for many cloud computing services like Google Cloud and AWS, while also being portable and extensible with a rapidly growing ecosystem of support and tools.

The standard Kubernetes scheduler uses fixed node and pod information to make scheduling decisions. I propose to apply Pronto to the Kubernetes ecosystem. The aim is to implement an online scheduler that uses online methods to both maximise resource utilisation and minimise per-pod latency while still being competitive with the industry-standard Kubernetes scheduler. During this project I had to modify the mathematics behind Pronto, proposing novel sub-space merging techniques, as well as a new signal function. In addition, I implemented a prototype of the Pronto-based Kubernetes scheduler, comparing it against the standard Kubernetes scheduler. These experiments demonstrate that Pronto is able to achieve a significantly lower pod completion time distribution while remaining competitive (Include values). Thus, Pronto may exceed in situations were pod completion is weighted more than throughput of pods, such as server-less compute.

Acknowledgements

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

Contents

Chapter 1

Introduction

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 [1]: 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.

Task scheduling as a problem spans multiple domains and scale: OS-level CPU and IO schedulers (e.g. CFQ and FCFS), workflow orchestrators (Airflow), batch scheduling workloads in data centres with thousands of machines. This project aims to tackle container orchestration within Kubernetes [2], an extensible open-source platform for managing containerized workloads and services. Originally created by Google, and thus inspired by existing in-house systems [**?**], it became open-source in 2014.

Kubernetes employs `kube-scheduler`, an In-Tree (default) scheduler that performs bin-packing, balanced allocation, topology-aware placement and pre-emption. In addition, `kube-scheduler` implements the **Scheduling Framework**, providing extension points on top of which Out-of-Tree (replacement) schedulers can extend the scheduler behaviour.

Talk about existing Kubernetes extension. Look at quality of Service plugins and how the current Kubernetes can't support that with bin-packing. Talk about rather than trying to blindly optimise for throughput, consider QoS. Look into how to combine in Serverless Compute / Lambda functions.

- Why Pronto? Pronto is a federated, asynchronous, memory-limited algorithm for online task scheduling across large-scale networks of hundreds of workers - Each node executes scheduling decisions on whether to accept an incoming job independently based on the workload seen thus far. In addition, aggregating local models can construct a holistic view of the system

Talk about how there has been few to no signal based Kubernetes plugins. Thus, this project aimed to transport Pronto into the domain of Kubernetes.

[Give an overview of the system structure: Central scheduler, remote scheduler and aggregate server]

Talk about modifying the mathematics behind Pronto to get it to work in this domain. Talk about creating the allocation function, the need for a reserve function.

To sum up, this project makes the following contributions:

- Implement a Pronto-based scheduler for Kubernetes
  - Modify the subspace merge operations to take into account a different vector space
  - Introduce signal processing techniques to calculate reserve amount
  - Implement TCP flow-control inspired technique to push throughput while still ensuring QoS
- Demonstrate that this prototype can achieve competitive throughput with substantial reduction in per-pod completion time.

Potentially discuss the drawbacks.

TODO: Check if there are any existing QoS schedulers that do not use predefined constraints. What is a use case in which one does not know exactly how much resources a task will take.

In Chapter 2, we go over existing Kubernetes schedulers and why they do not achieve true online QoS (no prior knowledge of pod resource usage and requirement). In Chapter 3, we explain our proposed system: outlining the mathematics, the system design, allocation algorithm and optimisations. In Chapter 4, we evaluate the performance of Pronto 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 Online Schedulers

Online Schedulers: *This chapter formalises the scheduling problem as well as highlighting the additional challenges of online scheduling vs offline scheduling. In addition, it will list existing scheduler taxonomies, highlighting their use cases, advantages and disadvantages.*

2.2 Kubernetes Overview

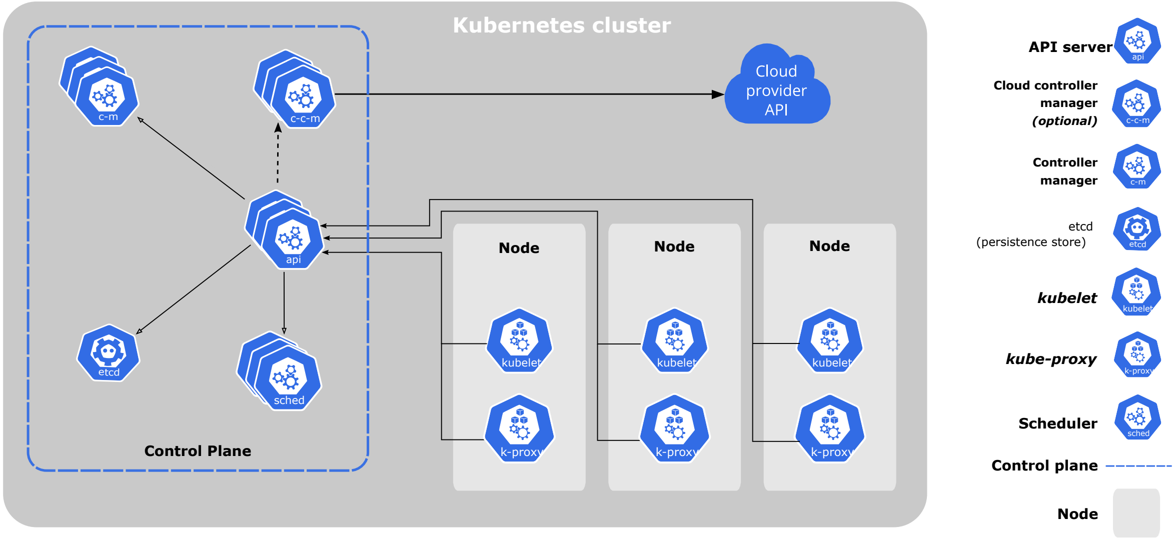

Figure 2.1: The components of a Kubernetes cluster

A Kubernetes cluster consists of a control plane and one or more worker nodes. Components in the control plane manage the overall state of the cluster. 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.

2.3 Analysis of Scheduling Approaches and QoS-Aware Trends

Different Kubernetes schedulers take varied approaches to plod 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.

2.4 Serverless Workloads

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.*

2.4.1 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.

2.4.2 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.

TODO: Should I include example workloads where incorrectly specifying requests can result in sub-par performance?

2.5 Kubernetes Serverless Schedulers

Kubernetes Serverless Schedulers: *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.*

2.5.1 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.

Figure 2.2: % CPU Utilisation and mean pod completion time against number of pods concurrently running on a node

2.5.2 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.

From figure 2.2, we can see that when running more than 5 pods on a single node, CPU Utilisation reaches 100 %. However, further increasing the pod count results in a sub-linear increase in mean pod completion time. This highlights how these metrics struggle to define the true capacity of a node.

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.

Summary of Related Work: *In this section, also discuss whether other schedulers use federation techniques to aggregate node telemetry to improve their scheduling decisions.*

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

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.

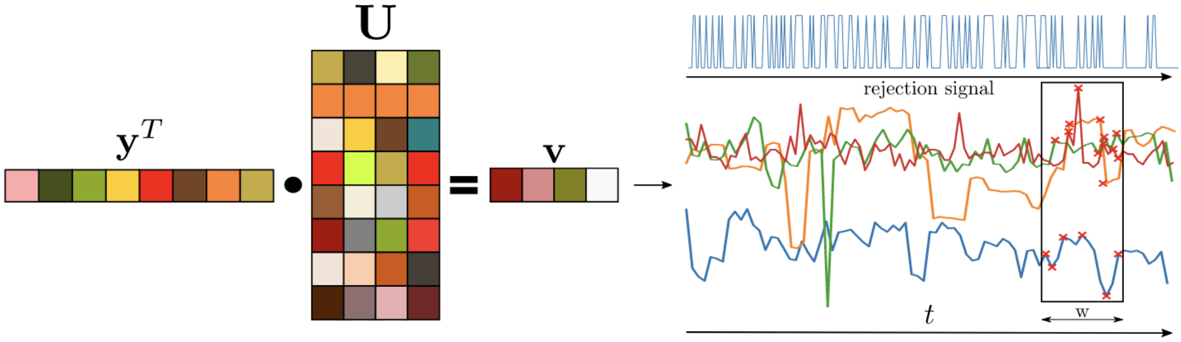

Figure 2.3: Projection of incoming $y \in \mathbb{R}^d$ onto embedding $U \in \mathbb{R}^{d \times r}$ producing R projections in $v \in \mathbb{R}^{1 \times r}$. Projections are tracked over time for detecting spikes which form the basis of the rejection signal. The sliding window for spike detection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

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

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.

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

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

Design and Implementation: *In this section, I will outline the goals of my system. I will give a brief overview of the chapter structure, summarising each core section and what I achieve.*

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:

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

Given $\text{SVD}(\mathbf{A}) = \mathbf{U}, \Sigma, \mathbf{V}^T$, 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 would result in the maximising a new equation:

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

$$= \left(\begin{bmatrix} - & x_i^T & - \end{bmatrix} \begin{bmatrix} | \\ a_1 \\ | \end{bmatrix} \right)^2 + \left(\begin{bmatrix} - & x_i^T & - \end{bmatrix} \begin{bmatrix} | \\ a_2 \\ | \end{bmatrix} \right)^2 + \cdots + \left(\begin{bmatrix} - & x_i^T & - \end{bmatrix} \begin{bmatrix} | \\ a_{2n} \\ | \end{bmatrix} \right)^2 \tag{3.4}$$

$$\geq \left(\begin{bmatrix} - & x_i^T & - \end{bmatrix} \begin{bmatrix} | \\ a_1 \\ | \end{bmatrix} \right)^2 + \left(\begin{bmatrix} - & x_i^T & - \end{bmatrix} \begin{bmatrix} | \\ a_2 \\ | \end{bmatrix} \right)^2 + \cdots + \left(\begin{bmatrix} - & x_i^T & - \end{bmatrix} \begin{bmatrix} | \\ a_n \\ | \end{bmatrix} \right)^2 \tag{3.5}$$

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.

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

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

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

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

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

$$y_{\text{predict}} = y + k * \frac{\sigma_1}{\text{Tr}(\Sigma)} u_1 \tag{3.10}$$

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

I scale u_1 by the proportion σ_i contributes to the trace of Σ. The idea behind this is that if u_1 projected distance is similar to the eigenvalue of other eigenvectors, the average resource usage was small or the sample in $\mathbf{A}$ are more sparse and thus we have less confidence that this is the direction. As a result, we should "expect" less of work in this direction.

3.5 System Architecture

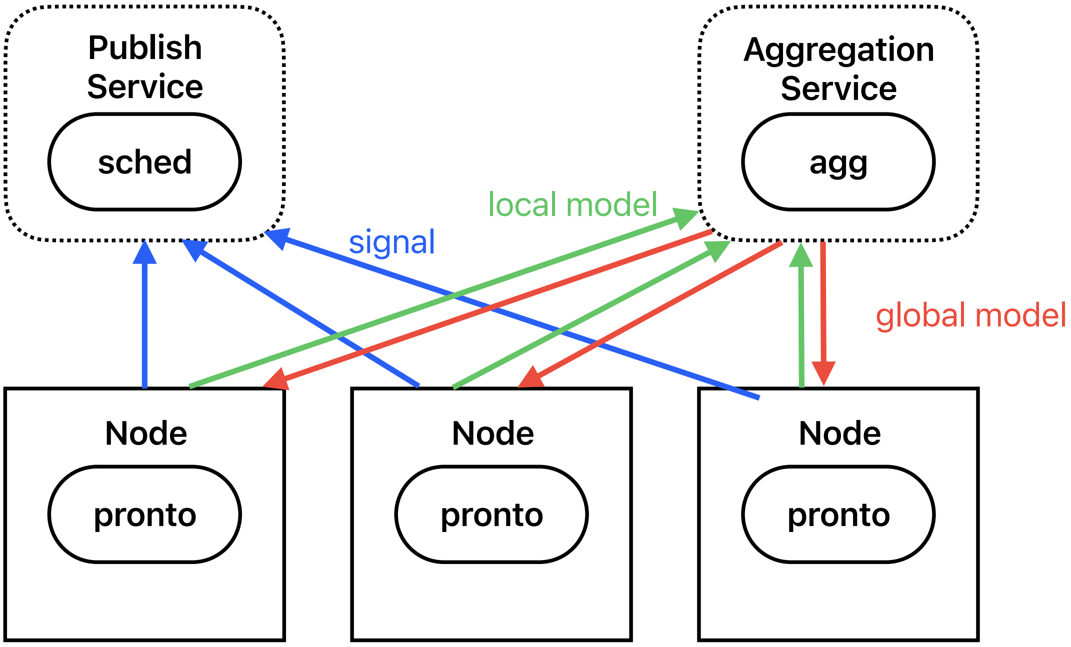

Figure 3.1: The components within the Pronto system

The Pronto system consists of three core components (shown in figure 3.1):

- Pronto DaemonSet: each node in the cluster will have a pronto pod running on it. This pod collects telemetry of the node performance and generates its local model and a willingness signal. When the Pronto pod deems its local model outdated, it requests the latest aggregated global model from the Aggregation service. In addition, it periodically generates and publishes its willingness signal to the Publish service.
- Scheduler: In Pronto, the scheduler is a **Scheduler Plugin**, implementing custom Filter, Score and Reserve functions. It also acts as the server of the Publish service, collecting the latest willingness scores of each node which it uses in its scheduling decisions.
- Aggregator: This deployment provides the Aggregation service. The underlying `pronto-agg` pod receives local models from nodes and returns the latest aggregated global model.

3.6 Applying Pronto For Kubernetes

3.6.1 Metric Selection

In the paper, Pronto uses `CPU-Reeady` which is generated by the VMware vSphere virtualisation platform. This metric can't be used within a Kubernetes cluster as machines can be both virtual and physical. Since Linux 4.20+, the kernel can track how long tasks are stalled waiting for the CPU at a cgroup granularity. By inspecting the the root cgroup's CPU pressure file using `cat /proc/pressure/cpu` you can measure the total time all processes spent waiting for the CPU to be available.

While this type of metric can be used to alert of performance degradation, this metric has a few shortcomings. Firstly, it only reports CPU-centric information. This is not always representative measure of resource contention as memory-heavy workloads may starve for RAM resources while metrics like `CPU-Ready` and `/proc/pressure/cpu` remain unaffected. TODO: Look into why I did not use metrics like /proc/pressure/cpu. If they are not useful, I can use them as an argument to why I used resource utilisation and pivot to the idea of Pronto allows federated information to help scheduling decisions. If it works, can quickly implement Pronto and contrast

Secondly, a significant amount of resources are used starting up or deleting containers. This results in large spikes, as shown in figure 3.2, which are difficult to distinguish from genuine CPU-Ready spikes. As Pronto uses spike detection to predict future resource performance degradation, container start-ups could produce detectable spikes which would reduce the rate at which pods are assigned to nodes and could result in lower throughput.

I decided to adapt Pronto to use fine-grained time-series measurements such as the CPU utilisation, memory consumption to provide real-time performance data of individual

Figure 3.2: The values of `/proc/pressure/cpu` during an example Kubernetes workload.

nodes as proposed in the future works section of [3].

3.6.2 Metric Collection

I decided to collect CPU and memory utilisation as my telemetry data, as these metrics are easily accessible and are used in a variety of industry-standard schedulers [4, 5].

Metric Server is a cluster-level add-on that provides near real-time CPU and memory usage metrics for Pods and Nodes. These metrics are easily accessed through the `metrics.k8s.io/v1` APIService and are updated by the Metric Server scraper periodically (default every 15 seconds). This method of metric collection is not suitable. Certain Pods may complete in less than 15 seconds and thus may not be detected by the signal. In addition, it would take 15 × batch size seconds between model updates (required to collect a single batch before performing subspace merging), and would result in a less representative and out-of-date model of "current" resource usage.

Instead, I decided to have a pod running on each node, scraping metrics from files within the Linux `/proc` directory. `/proc/stat` reports the cumulative count of "jiffies" (typically hundredths of a second) each CPU spent in a specific mode [6]. I can then calculate CPU utilisation using:

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

`/proc/meminfo` shows a snapshot of memory usage in kilobytes. The percentage of memory used can then be calculated from the provided fields:

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

Due to the higher refresh rate, we can poll these files more frequently (≈ 10Hz), and can remove the network latency that would come from making calls to the Metric Server APIService.

3.6.3 De-noising and Filtering Telemetry Data

While the modified version of Pronto that uses utilisation metrics won'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. To prevent a non-representative local model of pod resource utilisation, as well as a noisy willingness signal, we needed to de-noise the original metrics. Investigation 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

Figure 3.3: Resource utilisation telemetry of a node during the lifecycle of a pod. It will have tags for when container events occurred. In addition, I will demonstrate the effects of different filters when applied to the trace

spike exceeded the 300 millisecond threshold. From figure 3.3, we can see how Dynamic EMA outperforms other low-cost filters. 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.

TODO: Could do a more thorough investigation with a table containing event time distribution

3.7 Federated Resource Usage Model

3.7.1 Local Model Construction

To build its local model, each node periodically samples its telemetry data $y \in [0,1]^2$. Once it has b samples, the resulting telemetry dataset $A \in [0,1]^{2\times b}$ is merged into the previous subspace using iterative-SVD. We do not mean-center the telemetry data before applying SVD because we are interested in the magnitude of resources rather than their changes. If we applied standard Pronto to utilisation metrics, the creation or deletion of a pod would result in a spike in resource usage. Even if the resource usage peaked at 50% utilisation, Pronto only uses peak detection and thus would detect the spike and create a false-positive rejection signal.

The lack of mean-centering does violate the original assumptions of PCA. As previously mentioned in section 2.6.1, the resulting U and Σ from the SVD of a mean-centered matrix correspond to the principle components of the original matrix and the variance within the original matrix in those directions. When we don't mean-center the telemetric data, the resulting matrices produced by SVD have a different meaning: the resulting U and Σ matrices correspond to the principle components of the original matrix assuming that the mean-center was the $\mathbf{0}$ vector. In other words, U's column vectors form a basis along which maximises the sum of square distances from projecting the telemetry data along u_i. Crucially, u_i can be understood as the typical directions of resource usage of a pod running on a node.

One caveat of not mean-centering the matrix is that adding more non-zero vector samples to the telemetry data before running SVD will increase the resulting σ_i values. This also applies to running iterative-SVD to merge new samples into the local model. As σ is used in Pronto's signal function, and in later sections will be used in the new proposed signal, we want to prevent the value of σ from arbitrarily exploding. σ_i is equal to sum of square distances, and so scaling the concatenated matrix by $\frac{1}{\sqrt{2}}$ before applying SVD averages the square projected distance of both matrices in all vector directions. This results in a new "average" σ which merges new telemetry into the local model with a forget factor of

0.5.

3.7.2 Aggregation

The aggregation of local models to produce a global model also uses the same iterative-SVD as defined in section 3.7.1. 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.

TODO: Investigate the latency of throughput/limit of the aggregation server. TODO: Could have multiple aggregation server pods with a load balancer. Could then have these servers periodically share with each other their latest models to converge.

3.8 Signal Generation & Interpretation

3.8.1 Continuous "responsiveness" signal

The Pronto paper implements a binary "responsiveness" signal which predicts upcoming performance degradation. Because the authors assume a system with no communication latency (implicitly assuming that scheduled workloads were immediately visisble in the signal as well), they could send this signal directly to a central scheduler which could then stop assigning workloads once a node sent a Reject Signal.

However, due to significant pod startup latency, the method can't be used in a real-world Kubernetes cluster is infeasible. When measuring pod startup in a 100 node clusters, the more than 50% of pods took more than ≈ 1 second to startup. In addition, when nodes were 100% full, pod startup could reach up to 4 seconds. This latency is significant when Kubernetes schedulers can support a throughput of ≈1000 pods per second [7]. Applying the same approach as used in the paper, could result in nodes advertising a "willingness" to take on new pods while a large number of pods are in "flight" and once running will immediately overload the node. To prevent this runaway train type problem, I need to define a reservation function: a function that reserves an amount of the signal for a bound pod. This is necessary to allow previous scheduling decisions to have an imnpact on the signal while the signal updates to take into account the scheduled pods.

In addition, telemetry-based schedulers can use individual node performance information to score and fine-tune pod allocations. A binary signal does not provide the necessary information for scoring nodes, potentially resulting in worse pod allocations.

In summary, the requirements of the signal are:

- Reservable: the scheduler must be able to track the pending impact of previous scheduling decisions until the pods have begun running.
- Comparable: the signal must provide enough information to score nodes

3.8.2 Estimated Capacity

As mentioned in section 3.7.1, u_i can be understood as the typical directions of resource usage of a pod running on a node. In addition, σ_i can be understood as a measure of the amount of measured resource utilisation in the direction u_i. Therefore we can estimate future pod resource usage using a weighted average:

$$u_{\text{expected}} = \sum_i \frac{\sigma_i u_i}{\text{Tr}(\Sigma)}$$

We could then combine this with the latest measures of resource utilisation to estimate the amount of resource capacity we have left in the estimated future direction of resource utilisation.

$$y_{\text{expected}} = y + k u_{\text{expected}} max_k . \forall i : y_{\text{expected}} < 1$$

k gives us the capacity of a node until at least one of its resources are maximised. This k also allows us to assign higher scores to nodes which have more "capacity" than nodes which are experience more incompatible resource utilisation.

3.9 Reserve-Quantity Estimation

3.9.1 Problem Statement

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.

Figure 3.4: Examples of generated signals in different circumstances: favourable resource utilisation vs incompatible resource utilisation

3.9.2 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 reserve estimations until the burst has passed.

As shown in figure 3.5, 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.

Figure 3.5: Figure displaying a runtime trace of the signal. Add marks to the trace when container events were triggered and when Kubernetes API events were triggered

3.9.3 Reserve Estimation Techniques

I also investigated several methods of estimating reservation costs:

- Kalman Filter predicting reservation cost based on the function: Δsignal = Δno. of running pods$\times$ cost.
- 2D Kalman Filter to predict the signal based on the function: signal = capacity + per pod cost $\times$ no. of pods.
- Two separate Kalman Filters predicting the equation: signal = capacity+per pod cost$\times$ no. of pods. One filter predicts per-pod-cost while the other predicts capacity.

The Δ-based Kalman filter fails to accurately estimate the per-pod-cost once a node achieves full utilisation. When adding new pods to already saturated nodes, the change in the signal is very small (can be 0 if the node is already saturated). This results in a per-pod-cost that is far too low and would result in too many pods being added to a node by the scheduler.

The 2D Kalman filter fairs slightly better as it ensures that the per-pod-cost does not decrease to far. However, to ensure faster convergence, we used a Kalman filter with large constants in the Q matrix. This then resulted in large oscillations as the filter attempts to correct any error by modifying both the capacity and cost variables.

Figure 3.6: Figure displaying each methods per-pod-cost over an existing trace

Finally, I decided to use two separate Kalman filters; each filter learns its separate variable. By splitting the filter into two, we can prevent non-zero covariance entries in the P matrix which cause the oscillations seen in the 2D Kalman filter. This filter gave the most stable results while still being able to converge quickly.

TODO: Maybe also talk about the dynamic convergence.

3.9.4 Integrating into Signal

There are several methods to integrating the reserve cost into the signal to be sent to the central scheduler. The first method involves sending the signal and the reserve cost as separate values. On pod bind, we reserve the latest pod cost from the signal, and only relinquish the reserved amount once the central scheduler `kube-apiserver` listener detects the pod is no longer Pending. This method requires the central scheduler to keep track of both the pods on each node and the pod-cost when the pod was bound.

I instead chose to integrate the reserve quantity directly into the signal: each node calculates its available capacity in terms of no. of pods using two equations:

$$\text{avail. capacity from signal} = \frac{\text{signal}}{\text{per-pod-cost}}\text{avail. capacity from no. of pods} = \frac{\text{capacity}}{\text{per-pod-cost}} - \text{no. of pod}$$

We use to functions for different situations. Typically, we calculate the available capacity from the latest signal measurements. However, if we detect a recent container event, the remote `pronto` pod calculates the capacity from the pod count. This is to reduce the instability introduced by the container runtime. Furthermore, with this signal, the central scheduler does not have to record the per-pod-cost at bind time as the signal is now given in terms of individual pod counts. This simplifies the central scheduler and reduces the memory demand on.

3.10 Scheduler Integration

In this section, I will explain further how I integrated the previous components into the Kubernetes ecosystem.

3.10.1 Remote Scheduler Component

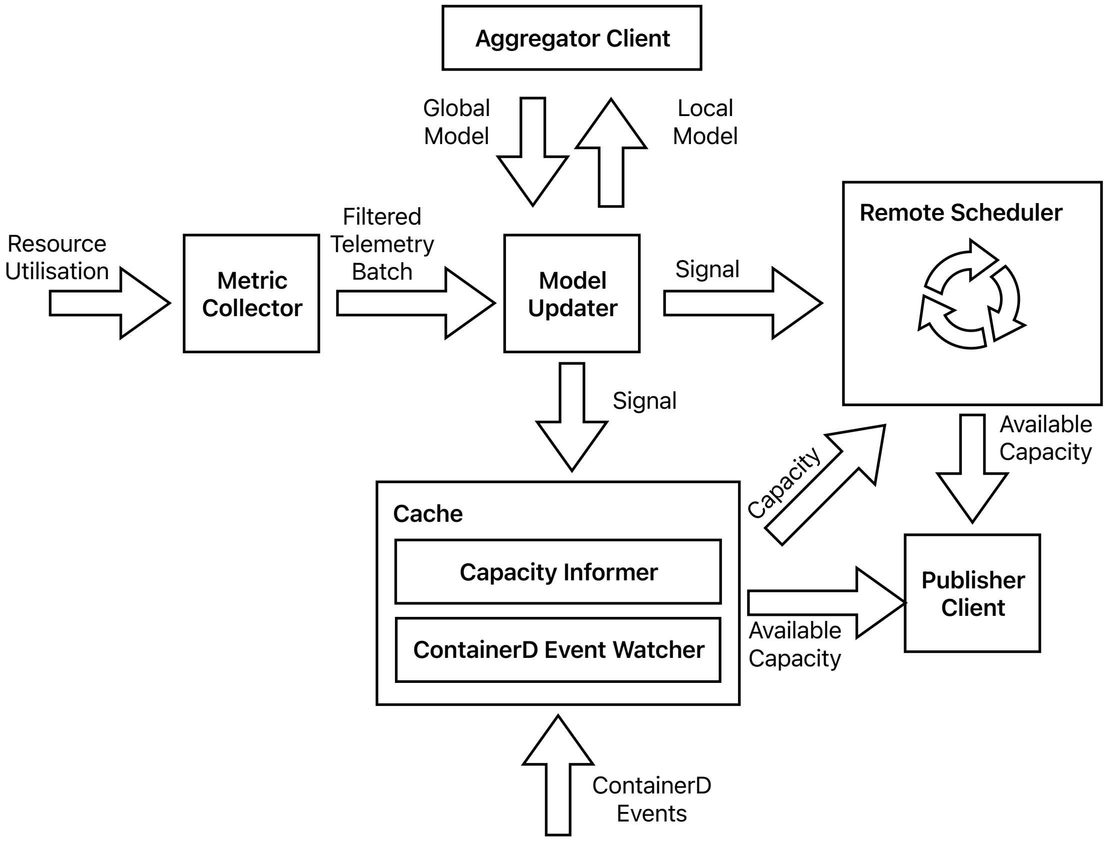

Figure 3.7: Core components within the remote scheduler pod

As mentioned in section 3.5, the signals are generated by pods defined by a DaemonSet: each node will contain a single pod which performs the core functionality of the Pronto system shown in figure 3.7.

3.10.2 Aggregation Component

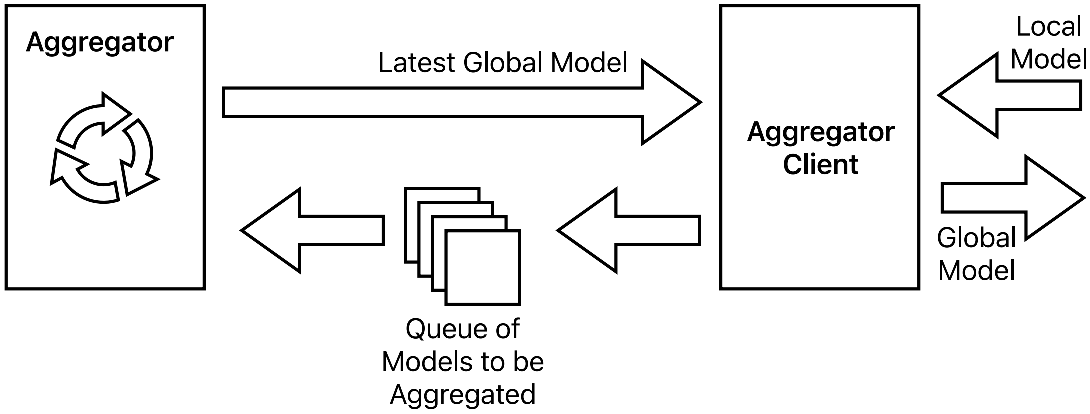

Figure 3.8: Core components within the aggregator pod

As mentioned earlier in section 3.7.2, the aggregator trades consistency for latency and throughput. 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.

3.10.3 Scheduler Component

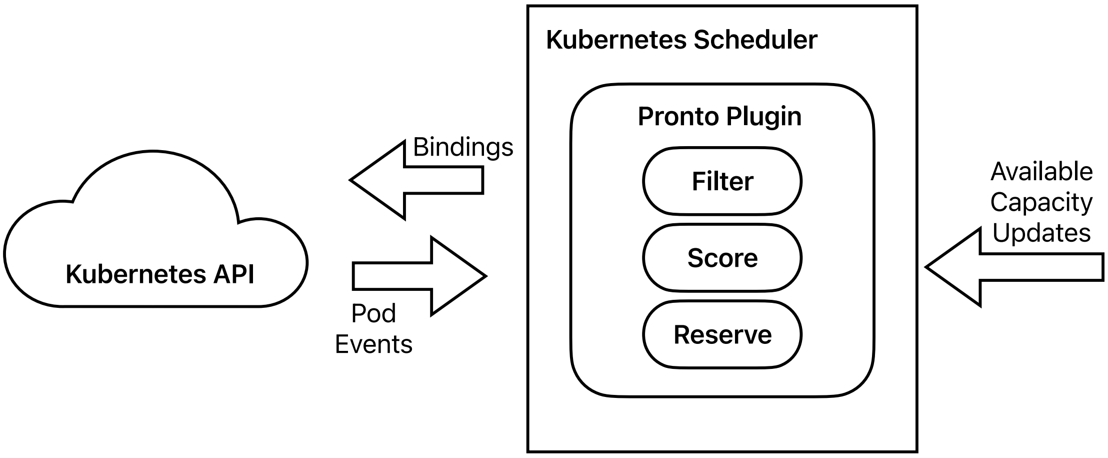

Figure 3.9: Core components within the central scheduler pod

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 $<$ ϵ.
- 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.

3.11 Throughput and Optimisations

3.11.1 Observed Sub-Linear Latency Scaling

While evaluating the prototype, I was stumped by its lackluster throughput compared to `kube-scheduler`. When deploying 1000 pods across 19 nodes, `kube-scheduler` would immediately distribute all pods fairly across all the nodes. This resulted in ≈ 45 pods running on each node. Whereas, the Pronto prototype ensured that no resource was fully utilised and had at most 5 pods running on a node at once. While the individual pod completion time was far lower, it resulted in a lower throughput.

Further investigation in the relationship between the number of pods running on a node at a time and their completion time showed a sub-linear relationship; shown in figure 3.10. I hypothesise that this is caused by CPU utilisation not being a representative utilisation metric. While a node may reach 100% CPU utilisation, it means that a CPU is always doing work, but it does not guarantee threads are experiencing high contention. Therefore, CPU utilisation is not a definitive measure of resource capacity, and explains why the prototype is not able to push through in terms of pod count and achieve higher throughput.

3.11.2 TCP-style Optimisation

The goal of the optimisation is to artificially increase the capacity of the node, as long as it does not push pod completion time beyong a limit and deteriorate QoS. An additional requirement is that the method needs to be as efficient as possible while also being able to handle pods of different workloads and completion times without the requirement of pod-descriptions.

To achieve this, I decided to borrow the concept from TCP flow-control. This method uses a moving window of the latest n pod completion times, with the goal to identify

Figure 3.10: Pod completion time against the number of pods running on a node at once.

deteriorating QoS. When receiving a new pod completion time, I update in O(1) time $\sum(\text{completion time})^2$. By squaring the completion times, the sum penalizes more longer pod completions that have increased. It then checks if the new sum is less than the threshold (previous sum scaled by a constant). If below, we can increment the over-provision capacity. However, if the new sum exceeds the threshold, the over-provision capacity is halved. This results in a similar saw-tooth behaviour as seen in TCP transmissions and allows the node to artificially increase capacity while mitigating increases in completion times.

Figure 3.11: Example trace of this over-provision capacity exhibiting TCP Saw-tooth trend.

Bibliography

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

[2] Overview. URL: https://kubernetes.io/docs/concepts/overview/.

[3] 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.

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

[5] 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.

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

[7] Abdul Qadeer. Scaling Kubernetes to Over 4k Nodes and 200k Pods, January 2022. URL: https://medium.com/paypal-tech/scaling-kubernetes-to-over-4k-nodes-and-200k-pods-29988fad6ed.

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.

A.2 Homo sapiens non urinat in ventum

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis.

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, At accusam aliquyam diam diam dolore dolores duo eirmod eos erat, et nonumy sed tempor et et invidunt justo labore Stet clita ea et gubergren, kasd magna no rebum. sanctus sea sed takimata ut vero voluptua. 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.

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.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis.

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, At accusam aliquyam diam diam dolore dolores duo eirmod eos erat, et nonumy sed tempor et et invidunt justo labore Stet clita ea et gubergren, kasd magna no rebum. sanctus sea sed takimata ut vero voluptua. 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.

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.