



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: 34

Main chapters (excluding front-matter, references and appendix): 25 pages (pp 7–31)

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 where pod completion is weighted more than throughput of pods, such as serverless compute.

Acknowledgements

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

Contents

1	Introduction	7
2	Background	9
2.1	Online Schedulers	9
2.2	Kubernetes Overview	9
2.3	Analysis of Scheduling Approaches and QoS-Aware Trends	10
2.3.1	Pod Spec-Only Scheduling vs Telemetry-Aware Scheduling	10
2.3.2	Impact on QoS (Latency, Throughput, Fairness)	12
2.3.3	Summary of Related Work	13
2.3.4	Shortcomings of Common Telemetrics	13
2.4	Pronto	14
2.4.1	Overview	14
2.4.2	Evaluation	15
2.5	Summary	15
3	Design and Implementation	17
3.1	System Architecture	17
3.2	Applying Pronto For Kubernetes	18
3.2.1	Metric Selection	18
3.2.2	Metric Collection	19
3.2.3	Denoising and Filtering Telemetry Data	20
3.3	Federated Resource Usage Model	21
3.3.1	Local Model Construction	21
3.3.2	Aggregation	22
3.3.3	Rejection Signal	22
3.4	Signal Generation & Interpretation	22
3.4.1	Continuous "responsiveness" signal	22
3.4.2	Estimated Capacity	23
3.4.3	Reservation	24
3.5	System Design	26
3.5.1	Overview	26
3.5.2	Framework Scheduler	27

3.5.3	Reserve Function	27
3.5.4	Optimisations	27
4	Evaluation	28
4.1	CPU-centric workloads	28
4.2	Short and Long Workloads	28
4.3	Multi-Resource Workloads	28
4.4	Limitation	28
4.5	Summary	29
5	Conclusion and Future Work	30
5.1	Summary	30
5.2	Future Work	30
A	Technical details, proofs, etc.	32
A.1	Lorem ipsum	32
A.2	Homo sapiens non urinat in ventum	32

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 datacenters 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 preemption. 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

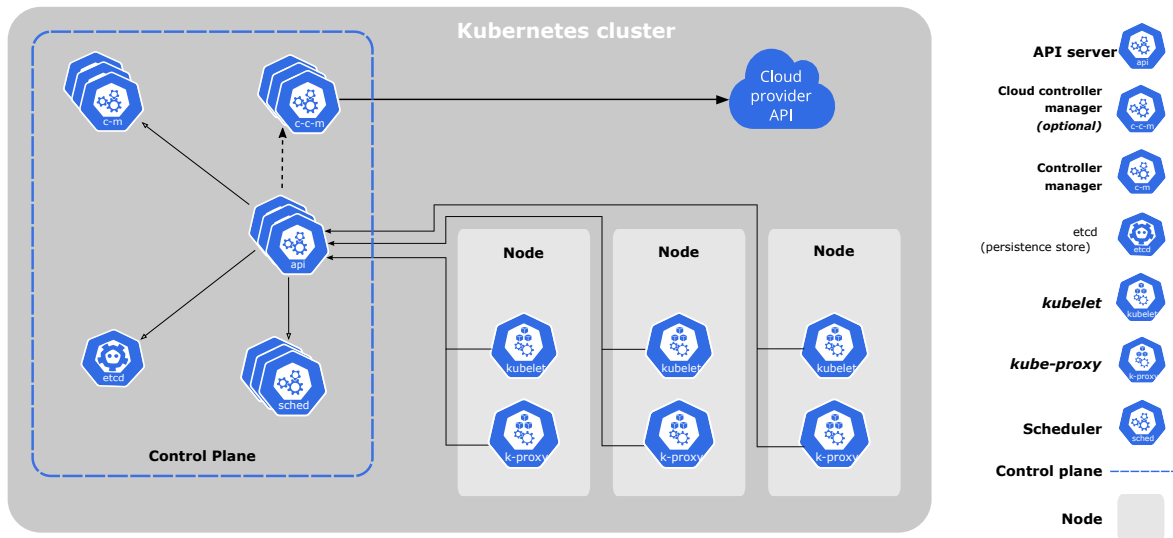


Figure 2.1: The components of a Kubernetes cluster

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-scheduler` 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 exist. 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 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 [1] and similar schedulers like Yunikorn or Volcano [2] 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 request, and it will honor 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 ot 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 performamance 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 [1] (which uses fine-grained platform metrics like LLC cache hit rates and CPU temperature) and the Trimaran plugin suite for Kubernetes [2] 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 equall allocable capacity remaining. This dynamic adaptability directly imporves 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 [1]. 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 performamance 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 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 colocate 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 affect 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 neighbor 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

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

jobs from latency-sensitive 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 limit - 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?

2.3.3 Summary of Related Work

The earlier survey compared how existing Kubernetes schedulers used two different 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.

2.3.4 Shortcomings of Common Telemetrics

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 just overcontention

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 chapter I will summarise the existing approaches to scheduling within the Kubernetes ecosystem. I will outline strengths and weakness of these systems with respect to reactive QoS. The goal of this chapter is to set up why I chose to apply Pronto to Kubernetes.*

2.4 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 the methodology behind Pronto can be used with both utilisation metrics and other existing Linux-based contention indicators.

2.4.1 Overview

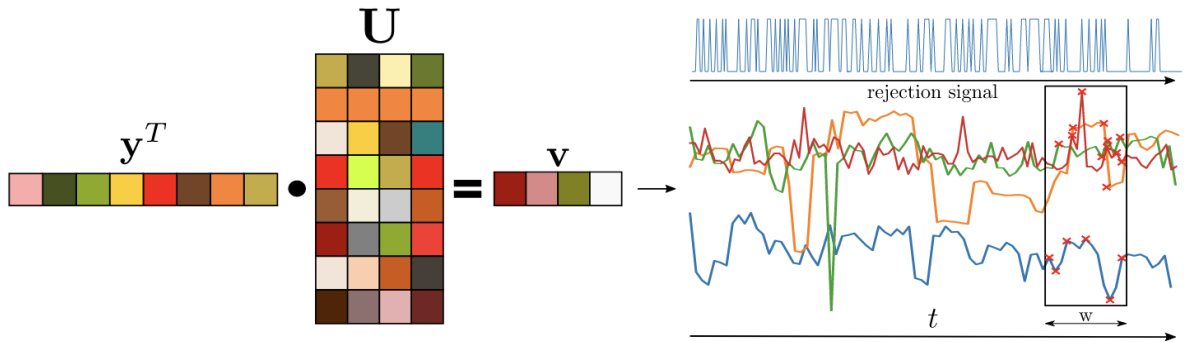


Figure 2.3: Projection of incoming $y \in \mathbb{R}^d$ onto embedding $U \in \mathbb{R}^{d \times r}$ producing R projections in $v \in \mathbb{R}^{1 \times r}$. Projections are tracked over time for detecting spikes which form the basis of the rejection signal. The sliding window for spike detection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

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 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 eigenvalues 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.

2.4.2 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. Pronto 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 accurately 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.

Why Pronto: *Pronto is promising due to two reasons, its federated nature which could allow nodes to exploit cluster-level information into their scheduling decisions, and its use of CPU-Ready, a standard metric to detect contention, but had not seen use in scheduling*

2.5 Summary

This chapter introduced the core concepts of online scheduling. 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. Finally, I introduce Pronto, a novel approach to task scheduling predictor that uses the

CPU-Ready performance metric generated by the VMware vSphere virtualisation platform.

Chapter 3

Design and Implementation

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.*

3.1 System Architecture

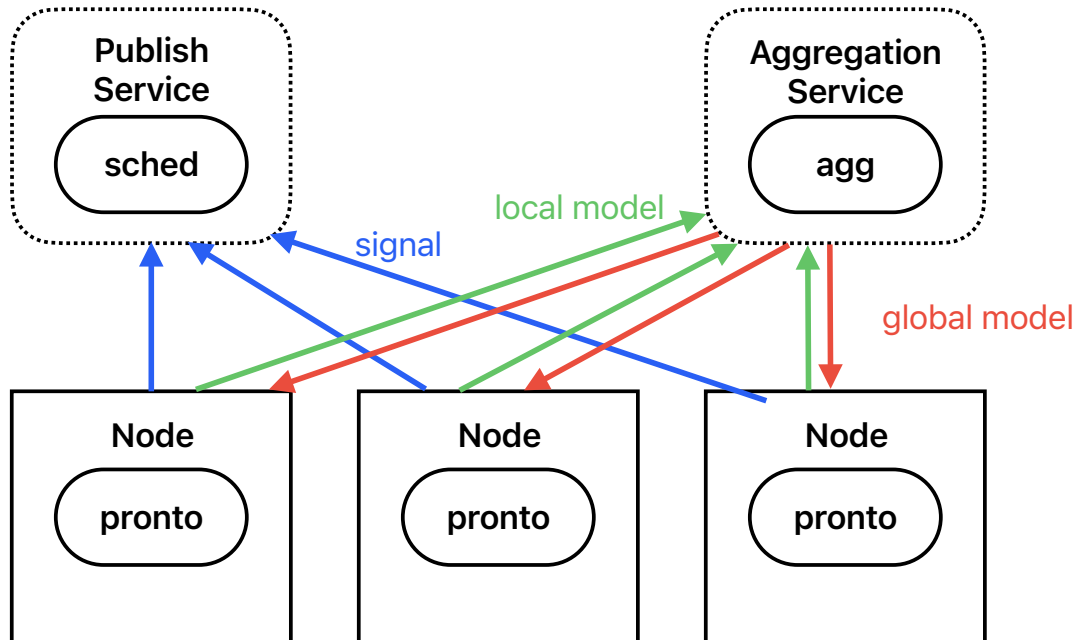


Figure 3.1: The components within the Pronto system

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.

3.2 Applying Pronto For Kubernetes

3.2.1 Metric Selection

In the paper, Pronto uses `CPU-Ready` 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 nodes as proposed in the future works section of [3].

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

3.2.2 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 15s). 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 $15 \times \text{batch size}$ 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 mem-

ory 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 ($\approx 10\text{Hz}$), and can remove the network latency that would come from making calls to the Metric Server APIService.

3.2.3 Denoising 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 denoise the original metrics. Investigation showed that 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

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. 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

3.3 Federated Resource Usage Model

3.3.1 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 2.4.1, 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 telemetry 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.

3.3.2 Aggregation

The aggregation of local models to produce a global model also uses the same iterative-SVD as defined in section 3.3.1. 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 eachother their latest models to converge.

3.3.3 Rejection Signal

Rejection Signal: *In this section I will discuss the possible drawbacks to using a binary rejection signal. Pronto proposes a means of predicting resource contention but does not include a means for allocating based on the Rejection-Signal.*

The Pronto paper assumes a system with no communication latency, which also implicitly includes no pod start-up latency. This simplifies scheduling decisions as the effects of prior decisions are immediately visible in the next signal update (typically 100ms-1s later). However, in Kubernetes, pod start-up can take from 1s-10s. If trying to schedule pods with a high throughput (the Kubernetes scheduler can support a throughput of ≈ 1000 pods per second [7]), this can result in hundreds of pods waiting to start-up on a node while it still advertises an Accept-Signal.

In addition, a binary signal is difficult to use when trying to score nodes and determine the optimal Pod placement. Therefore, I decided to extend Pronto's output to a continous signal in \mathbb{R} . By allowing nodes to advertise their responsiveness along a finite scale, more fine-grained comparisons can be made between nodes.

3.4 Signal Generation & Interpretation

3.4.1 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 visible 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 impact 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

3.4.2 Estimated Capacity

As mentioned in section 3.3.1, 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.

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

3.4.3 Reservation

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.

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 listners 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.

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

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 otherhand, 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.

Framework Scheduler: *In this section, I explain how I calculated the reserve quantity. I needed a method to estimate the amount of the signal to reserve when assigning a pod such that following scheduling decisions can see the future impact of the pod without having to wait for the signal to update / wait for the pod to start running.*

Explain how I used several methods:

- *Fixed cost determined before initialisation. Does not adapt to different workloads.*
- *Kalman Filter to predict the change in signal given the number of pods added. Difficult to perform when multiple pods are being added and deleted at a time. Hard to extract the actual impact of a pod.*
- *2D Kalman Filter to predict the signal based on the underlying equation: $\text{signal} = \text{capacity} + \text{per pod cost} \times \text{no. of pods}$. This method was relatively accurate, but would result in oscillations as the learned covariance between capacity and per-pod-cost resulted in fluctuations. In addition, trying to increase reactivity to changes in the signal, resulted in this fluctuations becoming extreme.*
- *2 separate 1D Kalman Filters, one predicting the per-pod-cost while the other predicts the capacity. This method worked the best. By separating the variables, I was able to prevent the oscillations in the previous method.*

3.5 System Design

3.5.1 Overview

Overview: *In this section, I give a brief overview of the Pronto system. I outline the individual remote schedulers which reside on each node and collect resource usage metrics and generate the acceptance signal.*

In addition, outline the Aggregation Service, with a central server and the remote schedulers which act as clients. The remote schedulers send their latest local model and the aggregate server responds with the latest aggregate model. To reduce latency, the aggregate model services each aggregation request by first enqueueing the local model to be aggregated and immediately sending a response with the latest version of the global model. This system trades consistency for latency, which is important when scaling to hundreds of nodes.

TODO: Investigate the latency of throughput/limit of the aggregation server.

Finally, give a brief overview of the central scheduler. Explain how it is a Framework Scheduler and thus replaces the Filter and Score functions

3.5.2 Framework Scheduler

Framework Scheduler: *In this section, I explain design of the central scheduler. I will give the reasons for why I decided to use a Framework Scheduler: use of efficient data structures, extensibility allows me to encode the new Filter and Score behaviour. Briefly explain the Filter, Score and Reserve semantics.*

3.5.3 Reserve Function

3.5.4 Optimisations

Optimisations: *In this section I explain the optimisation I implemented to improve scheduler throughput. I will include a figure of node utilisation when using the standard system, showing how extra capacity remains do to error between the true per-pod-cost and the estimated value.*

I also show the relation of pod-completion time to number of pods running on a node, alongside the signal. I will argue there are downsides to using CPU Utilisation. That once a CPU is fully utilised (no "jiffies" are spent in the idle state), the addition of more pods can't increase the CPU Utilisation and thus the signal does not change. In addition, the relation between pod-completion time vs pod count is typically sub-linear and thus encourages packing more pods on a node, even when the signal identifies full resource utilisation.

An optimisation I proposed was to borrow the concept from TCP flow-control. This method uses a moving window of the latest n pod completion times, which it uses to identify trends or potential spikes. The challenge in this situation 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 prior knowledge of different classes.

I compare the change in the sum of square completion times against a threshold. I use the sum of squares to penalize more an increase in the completion of longer running jobs. This is because a proportional increase in completion time has a worse impact the longer the job runs for. Whenever an incoming entry keeps the sum below the threshold, the overprovision capacity increases, while any completion times that exceed the threshold result in the overprovision capacity to halve. This results in a similar saw-tooth behaviour seen in TCP and allows nodes to accept more pods while keeping.

Chapter 4

Evaluation

4.1 CPU-centric workloads

CPU-Centric Metric: *In this section, I will compare Pronto against the Kubernetes scheduler when running simple CPU-based pods. I will measure job completion time, resource utilisation and individual pod completion time distribution*

4.2 Short and Long Workloads

Short and Long Workloads: *In this section, I will compare Pronto against the kubernetes scheduler when running pods of variable run-times.*

4.3 Multi-Resource Workloads

Multi-Resource Workloads: *Finally, I will compare Pronto against the kubernetes scheduler when running pods .*

4.4 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.*

4.5 Summary

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

Chapter 5

Conclusion and Future Work

5.1 Summary

5.2 Future Work

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 dui dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetur 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, consectetur 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, consectetur 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, consectetur 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.