

Priority Matters: Optimising Kubernetes Clusters Usage with Constraint-Based Pod Packing

Henrik Daniel Christensen¹, Saverio Giallorenzo², Jacopo Mauro¹

¹University of Southern Denmark

²University of Bologna

hench13@student.sdu.dk , saverio.giallorenzo@unibo.it, mauro@imada.sdu.dk

Abstract

Distributed applications employ Kubernetes for scalable, fault-tolerant deployments over computer clusters, where application components run in groups of containers called pods. The scheduler, at the heart of Kubernetes’ architecture, determines the placement of pods given their priority and resource requirements on cluster nodes. To quickly allocate pods, the scheduler uses lightweight heuristics that can lead to suboptimal placements and resource fragmentation, preventing allocations of otherwise deployable pods on the available nodes.

We propose the usage of constraint programming to find the optimal allocation of pods satisfying all their priorities and resource requests. Implementation-wise, our solution comes as a plug-in to the default scheduler that operates as a fallback mechanism when some pods cannot be allocated. Using the OR-Tools constraint solver, our experiments on small-to-mid-sized clusters indicate that, within a 1-second scheduling window, our approach places more higher-priority pods than the default scheduler (possibly demonstrating allocation optimality) in over 44% of realisable allocation scenarios where the default scheduler fails, while certifying that the default scheduler’s placement is already optimal in over 19% of scenarios. With a 10-second window, our approach improves placements in over 73% and still certifies that the default scheduler’s placement is already optimal in over 19% of scenarios.

Introduction

The state-of-the-art paradigms for implementing distributed systems are microservices (Dragoni et al. 2017) and serverless (Jonas et al. 2019). These paradigms decompose a distributed application into components that one can replicate independently of each other, according to their workload, avoiding wasting resources on overprovisioned components. Practically, developers and platforms use containers (Merkel 2014) to package these components and simplify their deployment/replication and container orchestrators (Jawarneh et al. 2019) to automate deployment, scaling, and management of containers on clusters of computation nodes. Nowadays, the main container orchestrator is Kubernetes (Lukša 2018). In Kubernetes, the *pod* represents the smallest deployable unit, enclosing a group of containers that share an execution context (e.g., networking, storage).

Copyright © 2025, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

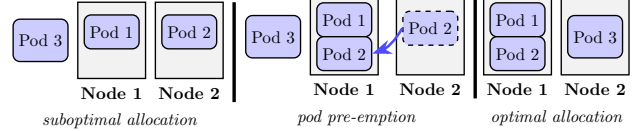


Figure 1: Example of Kubernetes scheduler suboptimal pod allocation problem – Node and Pod sizes represent their memory capacity/occupancy. Left: the heuristics place Pod 1 and Pod 2 on separate nodes, leaving insufficient memory on either node for Pod 3, despite the cluster having enough total resources. Centre: reallocation of the first two pods onto one node enables the (Right:) optimal placement of all pods.

The scheduler, at the heart of Kubernetes’ architecture, determines pods’ placement according to their requirements and node resources. The scheduler determines allocations through heuristics that prioritise speed and scalability but neglect optimal resource usage, possibly leading to system degradation (some pods are not allocated) or overprovisioning (the inclusion of new cluster nodes to host the pods not allocated) and resource waste (money, energy, etc.) (ClickHouse 2024a). Recent studies (Cast AI 2025) have shown that a staggering 99.94% of Kubernetes clusters are overprovisioned, where the average gap between provisioned and requested resources is 40% for CPU and 57% for memory.

As an illustration of this phenomenon, shown in Figure 1, let us consider a scenario with a 2-node cluster, each node with 4 GB of memory, where we want to deploy 3 pods in sequence, all with the same priority but requiring respectively 2, 2, and 3 GB of memory. When scheduling a pod, the scheduler heuristics use a LeastAllocated policy to rank nodes. Thus, it likely allocates the first two pods on one node each. When the scheduler receives the request for the last pod, it cannot find a node with 3 GB of memory available, leaving the allocation request pending, possibly requiring the addition of a new node to deploy the pod. The allocation (left of Figure 1) is *suboptimal* because the nodes have enough resources to host all pods. If we could reallocate one of two pods onto one node (centre of Figure 1), we would obtain the optimal allocation (right of Figure 1).

Pod priorities further complicate the problem of finding the optimal schedule in Kubernetes. The scheduler must allocate high-priority pods preferentially, even if it means pre-empting queued lower-priority pods. Hence, the dy-

dynamic nature of workloads, which requires the frequent creation/deletion of pods with varying priorities, further complicates the task of making allocation decisions that optimise overall resource usage and performance. In the worst case, the default scheduler’s policy can lead to pod starvation.¹

Our contribution We tackle the shortcomings of the Kubernetes scheduler heuristics by identifying failed deployment attempts that could fit within the cluster resources and devising an optimal resource allocation that ensures efficient resource usage and wards off avoidable operational costs. In this instance, we focus our attention on the representative case of small and medium-size clusters, which are renownedly prone to overprovisioning (ClickHouse 2024b) and where even saving single-node resources can significantly impact overall resource availability and costs, especially when leveraging the cloud infrastructure.

Our proposal is a conservative one and works by complementing the heuristics of the Kubernetes scheduler. We let the default heuristics handle allocations as long as they can satisfy pod requirements. If scheduling requests become pending (signalling requirement unsatisfiability) but they can fit within the available resources, then we use a (complete) solver to possibly find a solution (within a fixed time-frame) that satisfies all the priority and resource constraints.

To find a global optimum, we solve a series of optimisation problems on the possible pod allocations, given the pods’ priority. For every priority level, we guarantee the deployment of the maximum number of pods with that priority and minimise the total number of moved pods, to curtail overall system disruptions. The allocation problem is notoriously NP-hard (Garey and Johnson 1979), which implies that finding an optimal solution may take a long time. Since we aim to have a scheduling decision within a useful time-frame (e.g., 10 sec.), we set a timeout for each priority level to provide either a proven optimum or, if the solver timed-out, the best feasible allocation found until the timeout.

We evaluate feasibility and limitations of this approach using the state-of-the-art CP-SAT solver from OR-Tools (Perron and Furnon 2024). We simulate scheduling decision requests for small-to-medium-sized clusters with up to 32 nodes and varying degrees of pod requests, resource demands, and priorities. Our findings demonstrate that our approach scales to medium-sized clusters and can usually find a better, if not optimal, solution in less than 1 seconds for cluster of up to 8 nodes, or in 10–20 seconds for clusters of up to 32 nodes. In particular, with 1 second timeouts (resp. 10 seconds), our approach finds better or optimal solutions in 44% (resp. 73%) of scenarios where the default scheduler can not allocate all the pods. Moreover, our approach proves that the solution of the default scheduler is already optimal and can not be improved in 19% (resp. 19%) cases.

Structure of the paper We introduce Kubernetes and its scheduler in Preliminaries. In Implementation we detail our optimisation approach, its realisation, and integration as a Kubernetes plugin. In Evaluation we present the experiments conducted to evaluate our approach. We conclude by discussing Related Work and future directions.

Preliminaries

Kubernetes Kubernetes is the leading container orchestration platform for running cloud application workloads. In Kubernetes, scaling cloud architecture components, like microservices or serverless functions, typically entails running on a cluster of nodes one or more pod instances – each containing one or more software components – to obtain a desired number of available component replicas.

Kubernetes allows developers to assign a priority to every pod, useful to control pod pre-emption, i.e., the removal of pods from a node to make space for higher-priority ones. In such scenarios, Kubernetes sends a signal to a pre-empted pod to gracefully terminate – the default “grace period” is 30 seconds – after which the pod is (forcefully) removed.

To control the placement of pods, Kubernetes uses labels and selectors. Users can assign labels to nodes and pods and express (anti-)affinity constraints to influence the scheduling. For example, if a pod benefits from being scheduled on a node with an SSD, we can specify it to be affine with the label `ssd-disk` that also marks the SSD-equipped nodes.

A Kubernetes cluster includes a *control plane* and one or more nodes. The control plane exposes the API and interfaces to define, deploy, and manage the life cycle of containers. The nodes are the machines that host the containers. Each node hosts two components, kubelet – an agent responsible for actively running the pods and checking their status – and kube-proxy – a proxy that manages networking rules on nodes and supports network communication.

In production, the control plane usually runs across multiple machines and a cluster runs multiple nodes, providing fault-tolerance and high availability (Kubernetes 2024b). The control plane takes global decisions about the cluster, as well as detecting and handling cluster events, like errors. One of control plane’s central components is the scheduler.

Scheduler The scheduler decides where to place pods based on a set of rules and the available resources of the cluster. The default scheduler is called kube-scheduler.

Nodes that meet the requirements for a pod are called “feasible”. kube-scheduler’s decision process includes two successive phases: filtering and scoring. During filtering, kube-scheduler selects the feasible nodes, checking both resource and (anti-)affinity requirements. At the scoring, the scheduler finds the best node for placing the pod.

The scheduling framework consists of a set of “plugin” APIs directly compiled into the scheduler. These APIs allow one to implement most scheduling features as plugins while keeping the scheduling “core” lightweight and maintainable (Kubernetes 2024d). Two cycles determine the scheduling workflow, the *scheduling* and the *binding*. The scheduling cycle selects a node for placing the pod. The binding cycle links the pod to the selected node and prevents future scheduling cycles from allocating other pods on the portion of the node reserved to the linked pod.

Figure 2², illustrates the scheduling framework, where pentagons represent extension points and dashed rectangles indicate internal functionalities (unmodifiable); left to right:

¹<https://github.com/kubernetes/kubernetes/issues/86373>

²Adapted from <https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/>.

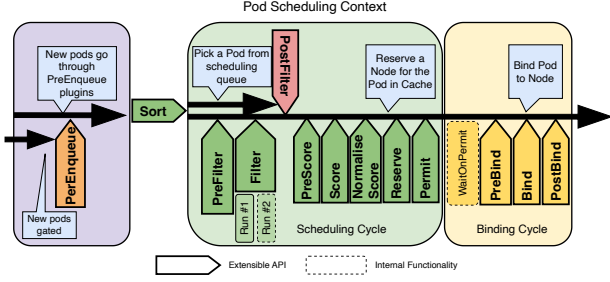


Figure 2: Scheduling Framework Extensions Points.

- PreEnqueue performs checks on the pod before it goes into the ready-for-scheduling queue. If one of these checks fails, the pod is not marked Schedulable (nor marked Unschedulable), and it goes in a special queue.
- QueueSort sorts pods in the scheduling queue. Only one of such plugins can be active at a time.
- PreFilter pre-processes information about the pod or checks conditions about the cluster. If one of these plugins returns an error, the scheduling cycle fails.
- Filter performs the filtering phase and prunes the nodes where the pod cannot run.
- PostFilter plugins activate only if the Filter plugins mark all nodes of the cluster as infeasible. This extension point is used mainly for pre-emption purposes.
- Score implements the scoring phase.
- NormalizeScore adjusts the scores produced by the Score plugins before completing the final ranking.
- Reserve activates before binding a pod to its designed node and takes care of concurrency issues that can emerge from parallel scheduling sessions. If the Reserve plugin, or a following one, fails, the resources reserved by this plugin are released by the Unreserve plugin.
- Permit delay the binding of a pod to its selected node.
- PreBind prepares the cluster (and the selected node) before the pod-node binding. If a PreBind plugin fails, the pod is rejected and placed in the scheduling queue.
- Bind plugins actively bind a pod to a node and they can choose whether to handle the pod or not. If a plugin does not bind the pod, another Bind plugin runs until either one successfully executes or the binding cycle gets cancelled, and the pod goes back into the scheduling queue.
- PostBind allow for final checks or modifications before the pod is bound to the selected node.

Implementation

We describe the formalisation of the optimisation problem we designed and how we integrate the formalisation into a proof-of-concept Kubernetes scheduler plugin.

Optimisation problem We solve the optimisation problem incrementally in a loop that iterates over all priorities. In every priority tier, a solver runs twice: first to maximise the number of placed pods, and second to minimise the number of allocated pods to move between nodes.

Notation: we use $n \in N$ to denote nodes. We define with $n.\text{ram}$ and $n.\text{cpu}$ the capacities of n . For each

pod $p \in P$, let $p.\text{ram}$ and $p.\text{cpu}$ be its resource requests, $p.\text{priority} \in [0, p_{\max}]$ its priority level (with lower values denoting higher priority), and $p.\text{where}$ the index of the node where the pod is currently placed ($p.\text{where} = 0$ if the pod is not scheduled). The binary variables $x_{i,j} \in \{0, 1\}$ indicate whether pod p_i is assigned to node n_j .

We report in Algorithm 1 the pseudocode of the optimisation loop, where a model m collects constraints and supports a solver that maximises a given metric.

Algorithm 1: Optimisation Algorithm.

```

1:  $m \leftarrow \text{Model}()$ 
2: for  $pr \in \text{range}(p_{\max} + 1)$  do
3:    $m.\text{add\_constraint}(\text{bin\_packing\_constraints}(pr))$ 
4:    $\triangleright (1)$  maximise number of placed pods
5:    $\text{metric} \leftarrow \sum_{p_i \in P: p_i.\text{priority} \leq pr} \sum_{n_j \in N} x_{i,j}$ 
6:    $\text{sol} \leftarrow m.\text{max}(\text{metric}, \text{get\_timeout}())$ 
7:   if  $\text{sol.status} = \text{OPTIMAL}$  then
8:      $m.\text{add\_constraint}(\text{metric} = \text{sol}(\text{metric}))$ 
9:   else
10:     $m.\text{add\_constraint}(\text{metric} \geq \text{sol}(\text{metric}))$ 
11:     $\triangleright (2)$  minimise number of pod moves
12:     $\text{metric} \leftarrow$ 
13:       $\sum_{p_i \in P: p_i.\text{priority} \leq pr} \left( \sum_{n_j \in N} x_{i,j} + 2x_{i,p_i.\text{where}} \right)$ 
14:     $\text{sol} \leftarrow m.\text{max}(\text{metric}, \text{get\_timeout}())$ 
15:    if  $\text{sol.status} = \text{OPTIMAL}$  then
16:       $m.\text{add\_constraint}(\text{metric} = \text{sol}(\text{metric}))$ 
17:    else
18:       $m.\text{add\_constraint}(\text{metric} \leq \text{sol}(\text{metric}))$ 
19: return  $\text{sol}$ 

```

First, we initialise the model without constraints (Line 1). Then, for each priority level pr from the highest (0) to the lowest ($p_{\max}$) we add (multi-dimensional) bin-packing constraints (Line 3) to restrict solutions only to valid placements, and use these to (1) maximise the pods allocated up to priority pr (Lines 5–10), and (2) to minimise the removal of already deployed pods (Lines 12–18).

In Algorithm 1, operation “bin_packing_constraints(pr)”, inspired by Shaw (2004), indicates the addition of constraints that ensure that the resource capacities of nodes are not exceeded when placing pods with priority up to pr . Formally, we add the following constraints.

$$\sum_{p_i \in P: p_i.\text{priority} \leq pr} x_{i,j} \cdot p_i.\text{ram} \leq n_j.\text{ram} \quad \forall j \in B \quad (1)$$

$$\sum_{p_i \in P: p_i.\text{priority} \leq pr} x_{i,j} \cdot p_i.\text{cpu} \leq n_j.\text{cpu} \quad \forall j \in B \quad (2)$$

$$\sum_{j \in N} x_{i,j} \leq 1 \quad \forall p_i \in P: p_i.\text{priority} \leq pr \quad (3)$$

Constraints (1) and (2) impose that the sum of the requested resources by the pod with priority up to pr are within the respective node’s RAM and CPU capacities, while (3) ensures the assignment of a pod with priority up to pr to at most one node.³

³Unlike Shaw (2004), we omit the constraint that the sum of the

At Line 6 of Algorithm 1, we maximise the number of pods within the current priority by maximising of the metric

$$\sum_{p_i \in P: p_i.\text{priority} \leq pr} \sum_{n_j \in N} x_{i,j}.$$

If the solver finds an OPTIMAL solution, we update the model by adding a constraint that imposes to find future solutions with exactly the same amount of pods up to priority pr (Line 8). If the solver did not terminate within the allocated time, thus not proving that the solution is optimal but providing only a FEASIBLE solution, we add a constraint that imposes to find solutions with at least the same amount of pods up to priority pr (Line 10).

In the second stage, we minimise the disruptions caused by avoidable pod evictions. To do so, we minimise the evictions of allocated pods up to priority pr by asking the solver, at Line 14, to maximise the following metric:

$$\sum_{p_i \in P: p_i.\text{priority} \leq pr \wedge p_i.\text{where} \neq 0} \left(\sum_{n_j \in N} x_{i,j} + 2x_{i,p_i.\text{where}} \right)$$

where we give a weight of 2 to the pods that remain in place (i.e., assigned to the same node they are already allocated onto) and a weight of 1 to the pods to move to another node.

Note that, since bin-packing is an NP-hard problem, both problems solved at every iteration of the loop are NP-hard, and the solver could take a long time to find an optimum. Since we define our algorithm to find a placement for pods, we model it to provide a solution (possibly not optimal) in a reasonable amount of time. For this reason, we design it to run with a fixed wall-clock timeout T_{total} , which limits the total solving time across all priority tiers. To avoid that a single priority tier exhausts the entire time budget, thus compromising the possibility to find a solution involving also lower priorities pods, we dedicate a fraction $\alpha \in [0, 1]$ of the total time to each priority tier, leaving the remaining $(1 - \alpha)T_{\text{total}}$ time for the opportunistic termination of the solver execution. Moreover, for every priority, we split in half the reserved time between the two solving phases (maximisation of pods and minimisation of evictions). In particular, in Algorithm 1, we use function `get_timeout` at every solver invocation (Line 6 and 14) to compute the available time budget. Formally, assuming that `unused` is the amount of unreserved time yet to consume, $\alpha \in [0, 1]$

$$\text{get_timeout}() = \alpha T_{\text{total}} / (p_{\text{max}} + 1) + \text{unused}$$

Solver We implement our approach by using OR-Tools (Perron and Furnon 2024) – an open-source suite developed by Google for optimisation tasks using, among other approaches, a lazy-constraint approach. Notably, OR-Tools is the winner of the latest MiniZinc Challenge (MiniZinc Team 2025), a competition evaluating the performance

bin loads is equal to the sum of the item size, which would prevent us from computing the optimal schedule if the resources were not enough to host all the pods – our core problem is not a bin-packing one and closer to a multi-knapsack one. We have also considered adding symmetry-breaking constraints (Shaw 2004), but we found that they did not improve the solving time in our experiments.

of constraint programming solvers across diverse benchmarks. In particular, we use the OR-Tools’ CP-SAT Python API, which runs several complementary search strategies in parallel. Since, unlike popular SMT solvers (e.g., Z3), CP-SAT does not support incremental push/pop of constraints, we re-solve the model after each place/move step. To try to speed up the search, we use CP-SAT hints to warm-start the next solve by providing the current assignment.

Kubernetes Plugin We recall that our goal is to conservatively enhance Kubernetes’ scheduler by enabling optimal scheduling decisions while retaining the efficiency of the default scheduling process. Unlike prior approaches (Lebesbye et al. 2021; Luca and Erascu 2023), which strive for optimality at every decision, our design invokes the solver only periodically or when needed (e.g., via an HTTP API), allowing the default scheduler to efficiently handle most allocations.

Notably, achieving global optimality requires pre-empting pods across nodes. At the moment Kubernetes pre-emption operates within a single node and API support to cross-node pod pre-emption is under consideration for development (Kubernetes 2024a,c). To implement of our proposal, we develop a custom scheduler plugin capable of evicting and re-scheduling pods according to the solver’s optimised placement, including cross-node preemption.

Given the current Kubernetes APIs, we implement the atomic binding and pre-emption of the pods that our optimiser suggests through the scheduling and eviction in separate scheduling events. We implement our plugin across five scheduler framework extension points: `PreEnqueue`, `PreFilter`, `PostFilter`, `Reserve/Unreserve`, and `PostBind` (cf. Preliminaries). Default preemption is disabled to ensure that all eviction and relocation decisions are controlled exclusively by our optimisation logic.

When the optimisation algorithm is invoked, it considers all cluster pods, both scheduled and pending. If the optimiser proposes relocating certain pods, we evict and re-schedule them in separate scheduling events, according to the computed plan. During solver execution, new pods arriving in the scheduling queue are temporarily paused. We record these pods in an internal list, which is used to re-queue them once the solver execution completes. The faster the solver, the smaller is the scheduling latency for new pods.

The plugin manages the allocation of pods affected by the optimiser’s plan at multiple extension points. First, at the `PreEnqueue` and `PreFilter` points, it assigns the affected pods to their target nodes, allowing the default scheduler to bind them accordingly. In the `PostFilter` we mark the pods that cannot be scheduled by the default scheduler. Since pod names change upon rescheduling, and we cannot rely on them for correct placement, at the `Reserve/Unreserve` point, the plugin explicitly reserves resources for the affected pods. Finally, at the `PostBind` point the plugin monitors progress and marks the plan as completed once all intended allocations are realised.

We have written our plugin in Go, which is the language used for the development of Kubernetes, and one can deploy it by extending Kubernetes’ default scheduler container. The optimisation algorithm calls a Python script, which invokes

OR-Tools, deployed within the scheduler’s container. The default scheduler, extended with our plugin, can be deployed as a container image within the cluster, replacing the original scheduler pod instance.

Evaluation

We move on to the empirical evaluation of our proposal. To the best of our knowledge, there is no standard benchmark for Kubernetes’ scheduler, nor publicly available datasets of real-world Kubernetes scheduling decision requests. Hence, to evaluate the performance of our optimisation algorithm, we generate random traces of pod requests, simulate initial allocations, and record the performance of our algorithm. To simulate the scheduling process, we use Kubernetes With-Out Kubelet (KWOK) (Kubernetes community 2025) – an open-source simulator that faithfully mirrors the behaviour of the standard Kubernetes scheduler. KWOK does not manage actual node resources, and it simulates node capacities and pod resource requests, enabling large-scale and reproducible scheduling experiments with minimal overhead.

Specifically, we generate a set of pod requests with configurable *a)* number of nodes in the cluster, *b)* average number of pods per node, *c)* workload ratio between the total amount of resources in the cluster and the ones needed by the pods, and *d)* maximal amount of pods’ priorities.

We create the pods with random values of CPU and RAM in the interval $[100, 1000]$. The total sum of these resource demands determines the node capacities together with the workload ratio. To reflect typical cloud deployments, all nodes are assumed to have identical resource capacities. Finally, we generate random ReplicaSets requests, i.e., a request to deploy a given amount of replicas of a pod. Each request requires the deployment of a random number in $[1, 4]$ of pods. The pod’s priorities are generated randomly.

We generate different datasets. Each dataset consists of 100 instances with various parameters combinations. We considered clusters with 4, 8, 16 or 32 nodes, with 4 and 8 average number of pods per node, and priority tiers in 1, 2, 4. As target usage, we considered 90%, 95%, 100%, and 105% of a cluster resources. We test the optimisation algorithm with different T_{total} timeouts: 1, 10, and 20 seconds.

The lower target usage levels (90% and 95%) present many instances that the default scheduler could solve trivially (thus not triggering our plugin). Thus, to focus on more challenging cases, we discard the instances where KWOK successfully places all pods, selecting the first 100 instances it fails to do so. Because the default scheduler is non-deterministic, for reproducibility, when generating the dataset instances (and only there) we force KWOK to behave deterministically by i) introducing a lightweight *Score* plugin to order nodes by their lexicographic name, ii) set *parallelism=1* to eliminate concurrency-induced non-determinism, and disable the *DefaultPreemption* plugin to prevent evicting lower-priority pods during scheduling.

After obtaining the 100 instances for each combination of parameters configuration, we evaluate our scheduling approach by running the default scheduler (as-is) in KWOK and then our optimisation algorithm, if the default scheduler failed to place all pods. We record the placements of

pods and whether the optimiser found an optimal solution or achieved a better allocation than the KWOK baseline (i.e., higher number of higher-priority pods).

We run all experiments on virtual machines with 8 vCPUs and 48 GB of RAM (Intel Xeon Gold 6130), Ubuntu Linux, and OR-Tools version 9.14.6206. The source code of the scheduler with our new plugin, including the optimisation algorithm, is available at <https://github.com/henrikdchristensen/scheduler-plugins> – the repository includes a thorough README under the directory at `pkg/mypriorityoptimizer` and the scripts to generate benchmarks, run tests, and reproduce the plots.

Results We visualise the summary of the results obtained with our optimisation algorithm in Figure 3. In the figure, the x-axis shows the number of nodes in the cluster, while the y-axis reports the percentage of instances of a given category (explained below). For every cluster size (4, 8, 16, and 32), we have three grouped stack bars reporting the different results using incremental solver timeouts (1, 10, and 20 s). In Figure 3, we compose the results to show the data related to a given priority tier (1, 2, 4) and average pods-per-node amount (4 and 8), aggregating across target usage levels.

The instance categories in the stack bars are:

- green (Better&Optimal), the optimiser found an optimal solution that was better than the one provided by the default scheduler;
- orange (Better), the solver found a better solution, but it could not terminate before the timeout, thus without proving the solution’s optimality;
- blue (KWOK Optimal), the default scheduler produced a solution that the solver proved to be optimal (i.e., no additional pod could be added);
- yellow (No Calls), the default scheduler allocated all pods and the solver was not invoked – due to the non-deterministic nature of the default scheduler;
- grey (Failures), the solver failed to produce a solution within the given time limit.

As expected, shown in Figure 3, increasing the timeout generally allows the optimiser to find more optimal solutions. For example, with 4 pods per node and two priority tiers, the share of optimal outcomes (green) visibly increases between 1 s and 10 s, with little gain reaching 20 s. We infer that most instances can be solved within a few seconds, and longer timeouts mainly help the more complex cases.

The optimisation problem becomes increasingly difficult as the cluster size grows. When moving from 8 to 32 nodes, the proportion of failed instances rises, showing that the solver often times-out and find no solution. This effect reflects the exponential search space increase, as the solver must explore a much larger number of configurations. In particular, for 32-node clusters, even a 20 s timeout slightly increases the chances to reach feasibility or optimality.

The number of pods per node also affects solver performance. With fewer pods per node, there are fewer possible placements, which makes the problem simpler. The effect of the number of priorities is also notable. As the number of priority tiers increases, the optimiser tends to find a higher

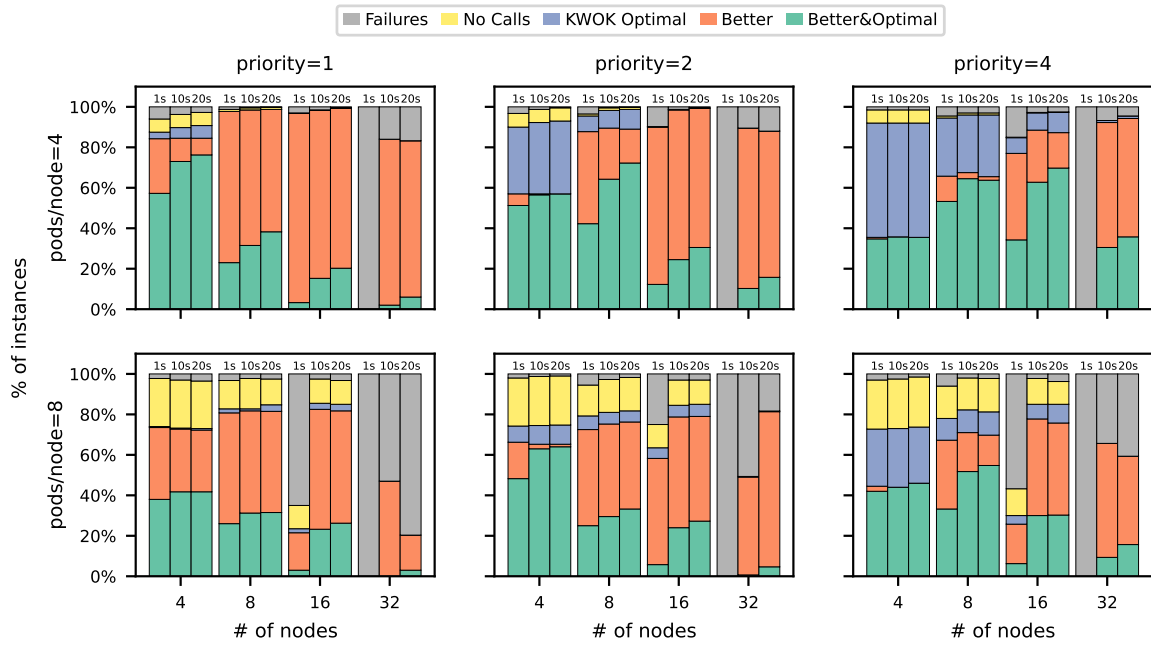


Figure 3: Distribution of solved instances by cluster size (x-axis). We collate the results by priority (columns) and pods-per-node (rows), aggregating usage levels. For each cluster size, grouped bars correspond to increasing solver timeouts (1, 10, 20 s).

proportion of optimal solutions (blue and green bars combined). This result likely comes from the fact that priorities introduce a clearer ordering between pods, guiding the solver’s search toward better solutions.

Summarising the results from Figure 3, longer timeouts improve solution quality but with diminishing returns beyond 10 s, larger clusters and higher pod densities increase solver difficulty, while additional priorities tend to guide the optimiser toward more optimal outcomes.

Figure 4 shows the target usage levels affect the optimisation performance as 3D bar plots of the percentage of solved instances, with 4 pods per node, 4 priorities, and a solver timeout of 10 s (other combinations show similar trends).

Overall, the usage level has only a moderate effect on solver performance. While higher usage levels (100–105%) slightly increases the number of failed or non-optimal outcomes, the overall trend is relatively stable across the levels. At lower usage levels (90–95%), the default scheduler frequently manages to schedule all pods (yellow), which limits the number of cases where optimisation is needed, suggesting that its benefits become more relevant as cluster resources approach full capacity.

Table 1 presents the average solver duration and the combined improvement in CPU and memory usage (Δ cpu/mem util (%)) relative to the default scheduler. We report the results for configurations with 4 and 8 pods per node, using 4 priority tiers, and a solver timeout of 10 s.

As expected, the solver duration generally increases with the number of nodes, reflecting the growing complexity of larger problems. For small cluster sizes (4–8 nodes), the solver completes within 1–2 s, while for 16 nodes, solving time rises to 5–10 s, hitting the limit of 10 s at 32 nodes.

Table 1: Solver performance metrics with 4 and 8 pods per node at different target usage levels. The results are based on 4 priorities, 10s solver timeout. The reason that the solver duration exceeds a little the timeout is that we limit the solver itself, and the time here is the total duration including extraction of the solution and I/O, which may slightly be above the solver timeout.

util	metric	ppn = 4				ppn = 8			
		4	8	16	32	4	8	16	32
90%	solver duration (s)	0.9	1.2	2.4	5.9	1.0	1.1	1.8	—
	Δ cpu util (%)	3.3	3.5	3.4	4.0	3.9	2.6	1.6	—
	Δ mem util (%)	3.1	3.7	3.1	3.9	3.4	2.6	1.6	—
95%	solver duration (s)	0.9	1.7	5.0	9.8	1.1	1.5	3.9	10.1
	Δ cpu util (%)	2.4	3.3	3.6	4.3	2.6	2.9	2.5	0.2
	Δ mem util (%)	2.4	3.3	3.7	4.3	2.6	2.9	2.5	0.2
100%	solver duration (s)	0.9	1.7	5.2	10.1	1.0	5.0	10.3	10.9
	Δ cpu util (%)	2.5	2.8	3.2	3.2	1.8	3.3	3.4	2.5
	Δ mem util (%)	2.1	2.5	3.0	3.3	2.1	3.1	3.1	2.4
105%	solver duration (s)	0.9	1.6	6.0	10.2	1.2	6.2	10.3	11.0
	Δ cpu util (%)	2.1	2.6	3.2	1.9	2.5	2.3	2.6	0.6
	Δ mem util (%)	1.8	2.1	2.9	2.1	2.4	2.8	2.5	0.5

The improvement in CPU and memory usage remains almost consistently around 2–4% across usage targets. This result shows that the optimiser not only schedules more high-priority pods, but it also achieves denser and more efficient resource allocations than the default scheduler. Even when the usage level is already high, the optimiser finds better placements that slightly increase the cluster capacity. At 100–105% usage levels, improvements diminish for the largest clusters, as most pods are already placed and few optimisation opportunities remain. We also observe that higher numbers of pods per node and larger cluster sizes tend to

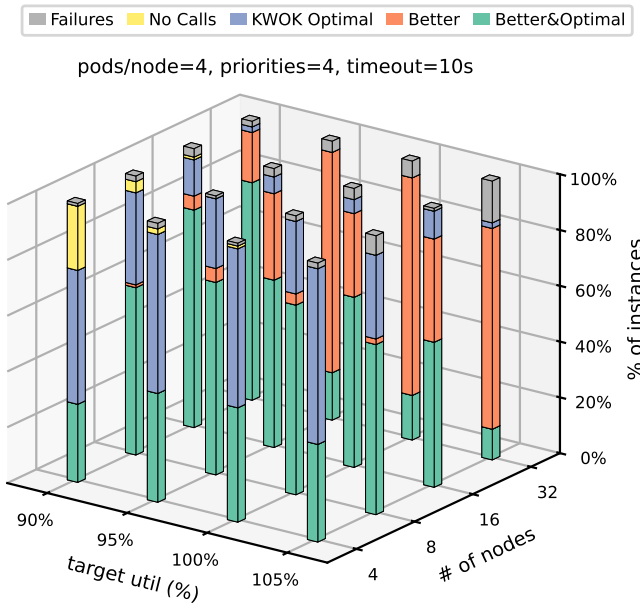


Figure 4: Distribution of solved instances for different target usage levels, with 4 pods per node, 4 priorities, and a solver timeout of 10 s.

slightly reduce usage gains, as the increased problem complexity limits the solver’s ability to find optimal solutions within the timeout. Finally, as expected, the higher number of priority tiers increases the solver duration, since we iterate over each priority tier.

Related Work

We position our work among the proposal for the optimisation of Kubernetes’ scheduler, referring the interested reader to Delorme, Iori, and Martello (2016) and Cacchiani et al. (2022) for a survey on exact algorithms for the bin-packing and knapsack problems proposed in the last fifty years.

Zhang, Ma, and Zhang (2018) and Carrión (2023) surveyed Kubernetes’ scheduler optimisations with meta-heuristics based on evolutionary and swarm-intelligence algorithms and particle-swarm optimisation. In general, these approaches are slower than the default scheduler.

Other works use machine learning techniques to improve scheduling decisions, predicting the scheduling tasks or the effects of the allocation of pods on nodes. For example, DL2 (Peng et al. 2021) uses the data of the default scheduler to train a new scheduler based on a deep neural network and relies on reinforcement learning to improve it. After an initial training time, the DL2 scheduler replaces the default scheduler. The authors of DL2 claim improvements over the default scheduler up to 44.1% in terms of average job completion time. However, a key challenge with DL2 and similar approaches arises when there are changes in workload patterns or cluster structure. The performance of machine learning can degrade significantly under such conditions, leading to suboptimal scheduling decisions.

Another direction regards the usage of mathematical mod-

elling to compute the optimal scheduler decision. Attempts in this area include SAGE (Luca and Erascu 2023) and BOREAS (Lebesbye et al. 2021). SAGE is a utility that, given a set of pods and deployment requirements, determines what set of VMs is optimal to deploy the pods, creates a Kubernetes cluster on those VMs, and schedules the pods on the nodes. BOREAS is a scheduler that collects pod requests and uses a configurator optimiser to allocate the pods on the remaining available nodes of the cluster. Both approaches do not consider priorities and pre-emption and rely on solving a bin-packing problem hinged on the assumption that node resources are enough to accommodate all the pods.

Santos et al. (2019) present a solution that relies on run-time information on the cluster and implement an extension of the default scheduling mechanism for Kubernetes, enabling it to make resource provisioning decisions based on the current status of the network infrastructure. Similarly, Nguyen et al. (2020) collect information on the distribution of network traffic and allocate resources proportionally to it.

Kaur et al. (2020) and James and Schien (2019) optimise the scheduling capabilities of Kubernetes based on power consumption and usage of green energy. These are interesting extensions that, differently from our proposal, consider alternative policies to the maximisation of allocated pods.

Conclusions

We present an improvement to Kubernetes’ scheduler that makes optimal scheduling decisions when pod requests cannot be fulfilled and become pending. We show that, by using off-the-shelf optimisation solvers, it is possible to improve the default scheduler allocation for Kubernetes clusters of small and medium size (up to 32 nodes) and with a varying number of pods, resource demands, and priority levels.

We deem our work a first step for future research in understanding the algorithmic trade-offs of performing multi-node pre-emption in Kubernetes. One promising future direction is to explore the usage of other solvers (e.g., MIP solvers like Gurobi (Gurobi Optimization, LLC 2024), CP solvers like Gecode (Gecode Team 2024) or bin-packing solvers of the BPPLib library (Delorme, Iori, and Martello 2024)), possibly using them in a portfolio approach (Kotthoff 2014) to improve the chances of finding better solutions. In particular, we envision exploring the possibility of combining heuristic approaches (Munien and Ezugwu 2021) meant to find good solutions quickly with complete solvers that can prove the solution optimal, if more time is available.

Another evolution regards supporting Kubernetes’ scheduler rules and cluster zones (i.e., requirements like node/pod (anti-)affinity that constraint pods deployment). This work would require adding constraints to further control the allocation of pods, possibly making the problem easier to solve thanks to a reduction of the search space.

Acknowledgements

We thank Simone Canova, Alessandro Neri, and Matteo Trentin, who contributed to preliminary groundwork for the techniques presented herein. Their initial explorations and suggestions helped the foundation behind this research.

References

- Cacchiani, V.; Iori, M.; Locatelli, A.; and Martello, S. 2022. Knapsack problems - An overview of recent advances. Part II: Multiple, multidimensional, and quadratic knapsack problems. *Comput. Oper. Res.*, 143: 105693.
- Carrión, C. 2023. Kubernetes Scheduling: Taxonomy, Ongoing Issues and Challenges. *ACM Computing Surveys*, 55(7): 1–37.
- Cast AI. 2025. 2025 Kubernetes Cost Benchmark Report. Technical report, Cast AI.
- ClickHouse. 2024a. Saving Millions of Dollars by Bin-Packing ClickHouse Pods in AWS EKS. <https://clickhouse.com/blog/packing-kubernetes-pods-more-efficiently-saving-money>.
- ClickHouse. 2024b. Saving Millions of Dollars by Bin-Packing ClickHouse Pods in AWS EKS.
- Delorme, M.; Iori, M.; and Martello, S. 2016. Bin packing and cutting stock problems: Mathematical models and exact algorithms. *Eur. J. Oper. Res.*, 255(1): 1–20.
- Delorme, M.; Iori, M.; and Martello, S. 2024. BPPLib: A Bin Packing Problem Library.
- Dragoni, N.; Giallorenzo, S.; Lluch-Lafuente, A.; Mazzara, M.; Montesi, F.; Mustafin, R.; and Safina, L. 2017. Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering*, 195–216. Springer.
- Garey, M. R.; and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman. ISBN 0-7167-1044-7.
- Gecode Team. 2024. Gecode: Generic Constraint Development Environment.
- Gurobi Optimization, LLC. 2024. Gurobi Optimizer.
- James, A.; and Schien, D. 2019. A Low Carbon Kubernetes Scheduler. In *ICT4S*, volume 2382 of *CEUR Workshop Proceedings*. CEUR-WS.org.
- Jawarneh, I. M. A.; Bellavista, P.; Bosi, F.; Foschini, L.; Martuscelli, G.; Montanari, R.; and Palopoli, A. 2019. Container Orchestration Engines: A Thorough Functional and Performance Comparison. In *IEEE International Conference on Communications, ICC*, 1–6. IEEE.
- Jonas, E.; Schleier-Smith, J.; Sreekanti, V.; Tsai, C.; Khandelwal, A.; Pu, Q.; Shankar, V.; Carreira, J.; Krauth, K.; Yadwadkar, N. J.; Gonzalez, J. E.; Popa, R. A.; Stoica, I.; and Patterson, D. A. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR*, abs/1902.03383.
- Kaur, K.; Garg, S.; Kaddoum, G.; Ahmed, S. H.; and Atiquz-zaman, M. 2020. KEIDS: Kubernetes-Based Energy and Interference Driven Scheduler for Industrial IoT in Edge-Cloud Ecosystem. *IEEE Internet of Things Journal*, 7(5): 4228–4237.
- Kotthoff, L. 2014. Algorithm Selection for Combinatorial Search Problems: A Survey. *AI Mag.*, 35(3): 48–60.
- Kubernetes. 2024a. Cross node preemption. <https://github.com/kubernetes-sigs/scheduler-plugins/tree/master/pkg/crossnodepreemption>.
- Kubernetes. 2024b. Kubernetes Components. <https://kubernetes.io/docs/concepts/overview/components/>.
- Kubernetes. 2024c. Pod Priority and Preemption. <https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/>.
- Kubernetes. 2024d. Scheduling Framework. <https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/>.
- Kubernetes community. 2025. Kubernetes WithOut Kubelet. <https://kwok.sigs.k8s.io/>.
- Lebesbye, T.; Mauro, J.; Turin, G.; and Yu, I. C. 2021. Boreas - A Service Scheduler for Optimal Kubernetes Deployment. In *ICSOC*, volume 13121 of *Lecture Notes in Computer Science*, 221–237. Springer.
- Luca, V.; and Erascu, M. 2023. SAGE - A Tool for Optimal Deployments in Kubernetes Clusters. In *CloudCom*, 10–17. IEEE.
- Lukša, M. 2018. *Kubernetes in action*. Manning Publications.
- Merkel, D. 2014. Docker: lightweight Linux containers for consistent development and deployment. *Linux J.*, 2014(239).
- MiniZinc Team. 2025. MiniZinc Challenge. <https://www.minizinc.org/challenge/>.
- Munien, C.; and Ezugwu, A. E. 2021. Metaheuristic algorithms for one-dimensional bin-packing problems: A survey of recent advances and applications. *J. Intell. Syst.*, 30(1): 636–663.
- Nguyen, N. D.; Phan, L.; Park, D.; Kim, S.; and Kim, T. 2020. ElasticFog: Elastic Resource Provisioning in Container-Based Fog Computing. *IEEE Access*, 8: 183879–183890.
- Peng, Y.; Bao, Y.; Chen, Y.; Wu, C.; Meng, C.; and Lin, W. 2021. DL2: A Deep Learning-Driven Scheduler for Deep Learning Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 32(8): 1947–1960.
- Perron, L.; and Furnon, V. 2024. OR-Tools. <https://developers.google.com/optimization/>.
- Santos, J.; Wauters, T.; Volckaert, B.; and Turck, F. D. 2019. Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing Applications. In *NetSoft*, 351–359. IEEE.
- Shaw, P. 2004. A Constraint for Bin Packing. In *CP*, volume 3258 of *Lecture Notes in Computer Science*, 648–662. Springer.
- Zhang, W.; Ma, X.; and Zhang, J. 2018. Research on Kubernetes’ Resource Scheduling Scheme. In *ICCNS*, 144–148. ACM.