# POSTER: Policy-driven security-aware scheduling in Kubernetes

Matthew Rossi
matthew.rossi@unibg.it
Università degli Studi di Bergamo
Bergamo, Italy

Michele Beretta
michele.beretta@unibg.it
Università degli Studi di Bergamo
Bergamo, Italy

Dario Facchinetti
dario.facchinetti@unibg.it
Università degli Studi di Bergamo
Bergamo, Italy

Stefano Paraboschi
stefano.paraboschi@unibg.it
Università degli Studi di Bergamo
Bergamo, Italy

## Abstract

Nowadays, Kubernetes is the leading platform for managing containerized application workloads. These are built of numerous *pods*, groups of one or more containers that are always co-located and co-scheduled on the same node. Given a pod, the *scheduler* performs a critical task, i.e., it finds the best possible node for its execution. This process is affected by several factors, including resource availability, hardware requirements, data processing restrictions (e.g., GDPR and CCPA), workload sensitivity, and the presence of other workloads. Developers can control the scheduling process through several methods, such as node selectors, affinity, anti-affinity, and topology spread constraints. However, this activity is cumbersome, error prone, and can easily lead to security incidents.

In this paper we propose an approach to constrain and validate pod scheduling decisions without relying on complex, handwritten node selection policies. The idea is to combine the node filtering capabilities of Kubernetes with the use of OPA Gatekeeper for automated policy enforcement. We discuss how this approach overcomes the limitation associated with existing solutions, and then describe how it is used to support corporate governance policies in common scenarios. Preliminary experiments confirm the applicability of our proposal.

## CCS Concepts

• **Security and privacy → Software and application security**; **Access control**.

## Keywords

Kubernetes, Security, Scheduling, Multi-tenancy, Data sovereignty, Workload isolation

**ACM Reference Format:**
Matthew Rossi, Michele Beretta, Dario Facchinetti, and Stefano Paraboschi. 2025. POSTER: Policy-driven security-aware scheduling in Kubernetes. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '25), August 25–29, 2025, Hanoi, Vietnam.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3708821.3735343

## 1 Introduction

Since its announcement in 2014, Kubernetes has become the industry leading solution for the orchestration of containerized workloads. According to a 2023 Cloud Native Computing Foundation's survey [1], 71% of respondents are using Kubernetes in production, and another 18% are evaluating its adoption. The reason of this success is Kubernetes' ability to automatically deploy, scale, and manage applications in a declarative way, independently of the size and complexity of the underlying cluster of nodes. To do so, it leverages the concept of *pod*, a group of containers that are tightly coupled, always co-located and co-scheduled, and share the same execution context and resources (e.g., storage, network).

However, companies that rely on Kubernetes also face some challenges. Indeed, as reported by Red Hat in the 2024 State of Kubernetes security report [2], nearly 9 in 10 organizations experienced security incidents, causing delays in application development to approximately 67% of companies, and even revenue or customer loss to 46% of all respondents. Indeed, when a vulnerable container is compromised the damage easily spreads to the entire pod due to the shared execution context, and when the vulnerability is severe and compromises the security of the node, it undermines all the co-located workloads.

Given this scenario, it is critical to provide developers and organizations with tools to improve the isolation between workloads. In particular, there are three common situations that can benefit from this: (i) *multi-tenancy*, where multiple tenants (e.g., customers or teams) run applications on the same cluster and need strong access control guarantees; (ii) *data sovereignty*, where several regulations (e.g., GDPR in Europe, and CCPA in California) require to control the geographical area where data is either stored and/or processed; and (iii) *incompatible sensitivity levels* for different workloads, as some may process personal data, implement critical services such as authentication and identity management, while others are exposed to untrusted input and/or depend on third-party code. In all these cases, it is crucial to select the execution node appropriately. While Kubernetes allows customizing the decision of the execution node during *scheduling*, this is done using verbose handwritten node selectors, affinity, anti-affinity, and topology spread constraints in the pod specification. Therefore, this process is tedious, error prone, and depends entirely on the developer.

In this paper we address this limitation by proposing the use of policies that are defined at cluster level, and are evaluated automatically every time pod creation requests are submitted to Kubernetes.

These policies are tailored on the previously mentioned use cases (i.e., multi-tenancy, data sovereignty, incompatible security levels), and mutate each pod creation request to ensure the selection of a policy-compliant execution node. In the following sections, we will first look at the native Kubernetes capabilities for limiting pod assignment, and then provide a detailed explanation of our solution. Finally, we will illustrate the experimental evaluation, showing the limited overhead associated with our approach.[1]

## 2 Native node filtering capabilities

As mentioned in Section 1, Kubernetes assigns pods to nodes based on many factors like the availability of resources in the cluster, the pod's resource requirements, and the distribution of the workloads. It also provides native methods to customize this process. We briefly explain the advantages and drawbacks associated with each of them.

*Node labels and selectors.* Node labels [4] are key-value pairs attached to nodes. When paired with the definition of node selectors in the pod specification, they allow to restrict the group of nodes eligible for execution to the ones matching all required node labels. While this is a viable solution, it has limited flexibility, hence it is only suitable for simple use cases.

*Node affinity.* Node affinity [3] complements node selectors creating a set of additional rules. These rules can express requirements (i.e., `requiredDuringSchedulingIgnoredDuringExecution`) and preferences (i.e., `preferredDuringSchedulingIgnoredDuringExecution`), allowing the scheduler to assign a pod even when it cannot find a node that satisfies all the constraints. Node affinity rules can quickly become complex to write and hard to validate, especially when using multiple `matchExpressions` clauses (e.g., `In`, `NotIn`, `Exists`, `DoesNotExist`, `Gt`, `Lt`).

*Taints and tolerations.* While node selection and affinity attract pods to a set of nodes, taints [7] allow nodes to repel pods that do not specify the corresponding taint tolerations. This capability is useful when a hardware feature is present only on few nodes in the cluster, and needs to be reserved for a specific set of pods that benefit from it. Taints are represented with simple key-value pairs and their resulting effect (e.g., `NoSchedule`). Therefore, they lack flexibility and do not allow the definition of complex expressions.

Although these features offer a solid starting point, it is essential to acknowledge their limitations. Indeed, they require all developers operating on the cluster to manually introduce a compliant and effective set of constraints. Moreover, since this activity is cumbersome and error prone, it can easily lead to security incidents due to lack of training, unclear company guidelines, or misconfigurations.

## 3 Our approach

The primary goal of this work is to improve workload isolation by automatically enforcing governance guidelines at the cluster level, without relying solely on handwritten policy constraints. In the following we clarify the threat model and illustrate our approach.

*Threat model.* In a Kubernetes cluster there is a risk of security incidents when workloads with incompatible security requirements and/or sensitivity levels are deployed on the same nodes. Indeed, workloads that process untrusted input can be compromised by an attacker, potentially gaining code execution at node level. When this happens, all workloads running on the node are exposed to the attacker, hence applications can suffer disruptions and failures, user data can be lost or leaked, and the company can be held liable and suffer financial and reputational damage.

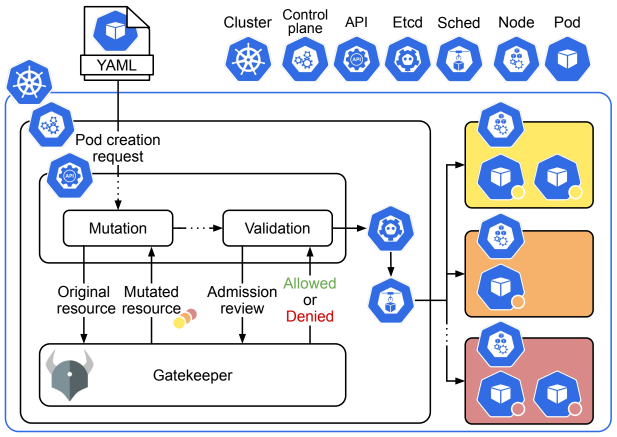

**Figure 1: The architecture of our solution: Kubernetes scheduling capabilities are enhanced by OPA Gatekeeper, which automatically mutates and validates pod creation requests to enforce cluster-defined policies**

*Our solution.* We propose to strengthen Kubernetes scheduling decisions using OPA Gatekeeper [6], an open source policy and governance framework for Kubernetes built on the Open Policy Agent (OPA) engine. In detail, Kubernetes delegates runtime policy decisions to the Gatekeeper admission controller. Gatekeeper acts as a bridge between the Kubernetes API server and OPA, fetching the relevant information associated with the pod resource definition, and invoking OPA to evaluate a set of cluster policies. Policies operate at the cluster level, and implement the use cases presented in Section 1. OPA Gatekeeper employs them in two stages: mutation and validation. During mutation Gatekeeper modifies the pod resource definition to include elements from a policy template, supporting the developer in the correct definition of pod scheduling constraints. During validation instead, OPA ensures pod creation requests comply with the cluster policy guidelines, allowing to admit (or reject) pod creation requests at runtime.

This approach does not require to modify extensively the architecture of an existing cluster (Figure 1). We only assume that (i) nodes in the cluster have already been labelled meaningfully (e.g., with information about tenant, geographical location, and sensitivity level), and (ii) the developer can leverage a comprehensive set of labels to classify pods based on their security needs (e.g., a pod processes sensitive data, it must be deployed in given region).

Our solution brings several advantages: (i) policy enforcement is automated by Gatekeeper, this avoids manually replicating error-prone node filtering definitions; (ii) policies are decoupled from the pod specification, simplifying pod definition and improving developer productivity; and most importantly, (iii) Gatekeeper ensures policies are applied consistently over the cluster, independently of the tenant that implements or deploys the workload.

[1]The code is available at https://github.com/matthewrossi/k8s-secure-scheduling

**Table 1: 90th percentiles for the evaluation of policies regarding data sovereignty (DS), multi-tenancy (MT), and incompatible security levels (ISL) scenarios. Mutation does not introduce scheduling changes in unconstrained use cases**

| Scenario | Latency [$\mu s$] | | |
|---|---|---|---|
| | Mutation | Validation | Scheduling |
| DS (EEA) | 900.09 | 900.73 | 942.77 |
| DS (US) | 900.12 | 900.57 | 982.35 |
| DS (unconstrained) | 900.17 | 900.76 | 925.56 |
| MT (tenant-A) | 900.23 | 900.32 | 951.11 |
| MT (tenant-B) | 900.21 | 900.41 | 944.19 |
| MT (unconstrained) | 900.07 | 900.14 | 926.24 |
| ISL (sensitive) | 900.12 | 900.50 | 942.34 |
| ISL (unhardened) | 900.30 | 900.54 | 936.39 |
| ISL (unconstrained) | 900.21 | 900.24 | 916.22 |

## 4 Evaluation

We performed a set of experiments to measure the time associated with the mutation and validation stages of pod creation requests, along with their subsequent pod scheduling latency.

The experiments have been performed on a server with Ubuntu 24.04 with kernel 6.8.0, a 128 cores AMD 7985WX CPU, 256 GiB of DDR5 RAM, and 2 TB SSD. To setup the Kubernetes cluster we used `kind` 0.27.0, Docker 28.0.1 for control plane nodes, the `containerd` 1.7.25 container runtime, KWOK[2] v0.6.1 to simulate worker nodes, and ClusterLoader2[3] to run performance tests. OPA Gatekeeper 3.18.2 was used to mutate and validate scheduling requests, and Prometheus 2.25.0 to gather the test metrics.

To assess the accuracy of our test environment in measuring the impact of our solution on control plane components, we initially run a small-scale test with 3 control plane nodes, comparing the performance associated with the use of 10 real versus 10 simulated worker nodes. As expected, simulating workers still provides accurate control plane performance measures. So, we proceeded with the creation of 3 control plane nodes, on which our solution is run, and a total of 1k simulated worker nodes for the creation of 30k pods (with 500 qps). Table 1 shows the results for each use case. The measures report the 90th percentile of the latency introduced to mutate and validate pod creation requests, as well as the pod scheduling time. In all use cases these steps are completed within 1 *ms*. Finally, we monitored the API availability and confirmed that our solution does not affect the normal functioning of the cluster, as availability always remains at 100% even under heavy load.

## 5 Related Work

Several approaches improve the isolation in a Kubernetes cluster.

*Kubernetes namespaces.* Namespaces [5] provide a mechanism for isolating groups of API resources within a single cluster. They are useful to avoid name clashes and play a significant role in the definition of resource quotas (e.g., memory, CPU). However, namespaces are not meant to influence scheduling decisions, so a privileged pod or a container breakout can affect workloads in other namespaces on the same node.

*Sandboxing.* In Kubernetes, workloads can benefit from the isolation provided by sandboxed runtimes, such as gVisor [16] and Firecracker [11], as well as take advantage of common kernel-based sandboxing solutions [8–10, 13, 14]. While all these techniques offer strong security guarantees, they also increase complexity, and inevitably introduce higher overhead and resource utilization.

*Node isolation.* With node isolation, a set of nodes is dedicated to running pods having a particular security profile. In Workload Security Rings [12], a proposal by Google, this technique is used to mitigate the risk of lateral movement, as sensitive workloads are never co-located with the ones that process untrusted data. However, their approach is only available in Borg [15], while our solution can replicate this behavior on Kubernetes.

## 6 Conclusions and future work

The results achieved by our approach are promising: not only it permits to automatically apply cluster-level policies during pod creation without significant architectural changes, but it is also associated with a negligible performance impact. Future work includes the exploration of real-time threat-informed policy adaptation and the extension of our solution to multi-cluster deployments.

## Acknowledgments

This work was supported in part by the EC under project GLACIATION (01070141), by the Italian MUR under PRIN project POLAR (2022LA8XBH), and by projects SERICS (PE00000014) and GRINS (PE00000018) in the NRRP MUR program funded by the EU–NGEU.

## References

[1] 2023. *CNCF Annual Survey.* https://cncf.io/reports/cncf-annual-survey-2023/
[2] 2024. *Kubernetes adoption, security, and market trends report.* https://www.redhat.com/en/resources/kubernetes-adoption-security-market-trends-overview
[3] 2025. *Assign Pods to Nodes using Affinity.* https://kubernetes.io/docs/tasks/configure-pod-container/assign-pods-nodes-using-node-affinity/
[4] 2025. *Assign Pods to Nodes with Node labels.* https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#built-in-node-labels
[5] 2025. *Namespaces | Kubernetes.* https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/
[6] 2025. *OPA Gatekeeper.* https://open-policy-agent.github.io/gatekeeper/website/
[7] 2025. *Taints and Tolerations – Kubernetes.* https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/
[8] M. Abbadini, M. Beretta, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi. 2023. Lightweight Cloud Application Sandboxing. In *CLOUDCOM*.
[9] M. Abbadini, M. Beretta, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi. 2023. POSTER: Leveraging eBPF to enhance sandboxing of WebAssembly runtimes. In *ASIACCS*.
[10] M. Abbadini, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi. 2023. NatiSand: Native Code Sandboxing for JavaScript Runtimes. In *RAID*.
[11] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D. Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *NSDI*.
[12] M. Czapiński and R. Wolafka. 2023. Workload Security Rings. https://www.usenix.org/publications/loginonline/workload-security-rings
[13] M. Rossi, M. Beretta, D. Facchinetti, and S. Paraboschi. 2025. POSTER: Transparent Temporally-Specialized System Call Filters. In *ASIACCS*.
[14] Y. Sun, D. Safford, M. Zohar, D. Pendarakis, Z. G., and T. Jaeger. 2018. Security Namespace: Making Linux Security Frameworks Available to Containers. In *USENIX Security*.
[15] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. 2015. Large-scale cluster management at Google with Borg. In *EuroSys*.
[16] E. G. Young, P. Zhu, T. Caraza-Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. 2019. The true cost of containing: a gVisor case study. In *HotCloud*.

[2]https://github.com/kubernetes-sigs/kwok
[3]https://github.com/kubernetes/perf-tests/tree/master/clusterloader2