Secure Kubernetes Workload Deployment with Automated Enforcement of Cluster-Defined Policies

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

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

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

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

***Abstract*—Scheduling pods on separate physical nodes is a crucial strategy to isolate workloads with incompatible security requirements. In Kubernetes, this is enforced using metadata such as node selectors, affinity rules, and topology spread constraints, all manually defined by developers at resource creation. The aforementioned process is complex and prone to errors, frequently resulting in misconfigurations that expose systems to data breaches and regulatory violations.**

This paper proposes an approach to constrain scheduling using policies defined once at the cluster level and automatically evaluated by Kubernetes during each workload deployment. The advantages are (i) automatic rejection of uncompliant resource creation requests, (ii) streamlined support for executing multi-tenant workloads, and (iii) secure scheduling and deployment of workloads based on security requirements. To implement this solution, we integrate the native Kubernetes node-filtering capabilities with OPA Gatekeeper for policy enforcement. We demonstrate how this approach reliably enforces common corporate governance policies and analyze its performance advantage over isolation achieved solely through sandboxing. The experimental evaluation confirms the effectiveness of our proposal and the minimal overhead.

***Index Terms*—Kubernetes, Security, Node isolation, Scheduling, Data sovereignty, Multi-tenancy**

I. Introduction

Kubernetes has emerged as the predominant technology for managing containerized workloads [1]. Its execution model revolves around the concept of *pod*, defined as a cohesive group of interdependent containers that are co-scheduled within a shared execution context with dedicated resources. Pods significantly simplify deployment and orchestration tasks, but scheduling multiple pods on a single physical node introduces notable security challenges. When containerized applications are flawed, the shared execution context can be compromised, and in case of severe vulnerabilities all workloads running on the node can be affected, leading to widespread impact (e.g., [2]–[5]). This is also testified by the 2024 Red Hat's State of Kubernetes Security Report [6], which found that 90% of surveyed organizations experienced Kubernetes-related security incidents, and 46% reported loss of revenue or customers.

Enhancing isolation between workloads is a key strategy for mitigating vulnerabilities. The industry has proposed several ways to achieve this. For instance, Google has heavily invested in gVisor [7] to isolate Linux host systems from application containers. Amazon instead proposed to isolate multi-tenant container services through virtualization using Firecracker [8], a method also adopted by Kata Containers [9]. Although these solutions are crucial to isolate co-located workloads, they inevitably increase resource consumption and introduce performance overhead [8], [10]. Moreover, sandboxing does not fully address the broader spectrum of security-related requirements. Indeed, there are three cases that prove complex to satisfy: (i) regulations such as General Data Protection Regulation (GDPR) and California Consumer Privacy Act (CCPA) impose constraints on how data are stored and processed to protect data sovereignty, (ii) different tenants, like customers or teams, often need tailored configurations to prevent unintended data sharing within the same cluster, and (iii) workloads can require allocation based on the availability of dedicated hardware resources. Hence, precise control over workload deployment and execution is essential for achieving a solid security posture.

Kubernetes facilitates this control through *scheduling constraints* [11], rules which govern the distribution of pods across a cluster. These are embedded in the pod specification and are interpreted by the Kubernetes scheduler upon receiving a pod creation request. Unfortunately, they must be manually specified by developers for every workload, a practice that increases the risks of data breaches or regulatory violations.

Contribution. In this work, we propose a solution to automatically generate and enforce scheduling constraints based on workload security requirements. This approach enables the definition of a policy at the cluster level, ensuring automatic enforcement whenever a pod creation request is submitted. Major benefits include: (i) elimination of organizations' reliance on handwritten scheduling constraints, (ii) uniform governance policy application across all workloads, with automatic rejection of non-compliant requests, and (iii) precise and granular workload isolation based on security requirements. Building upon these features, our solution can also reproduce Workload Security Rings [12], a recent Google proposal

exclusive to Borg [13] for managing workload deployment based on sensitivity levels. The artifact of the software and the scripts that permit the reproduction of all the experiments reported in the paper are available open-source.[1]

Outline. We start with a background on native Kubernetes scheduling capabilities and the introduction of the use cases in Section II. Then, Section III presents the threat model and details the architecture of our approach. Section IV explains the validation of the design, while Sections V evaluates the scheduling overhead and the potential performance improvement over a sandboxing-only approach. Section VI discusses the related work. Finally, conclusions are drawn in Section VII.

II. Motivation

This section highlights the limitations of the native Kubernetes scheduling capabilities and then introduces the key use cases that motivate our work.

A. Kubernetes scheduling constraints

Before describing our proposal we give an overview of the native Kubernetes scheduling capabilities. Specifically, we briefly introduce (i) node labels and selectors, (ii) node affinity, and (iii) taints and tolerations.

Node labels [14] are key-value pairs assigned to nodes. Pod specifications utilize these labels through node selectors for direct matching. However, this selection mechanism lacks flexibility, as individual selectors cannot be combined with logic operators.

Affinity [15] introduces advanced scheduling rules beyond basic selectors, distinguishing between hard requirements and soft preferences. Although preferences allow scheduling even when rules are partially satisfied, the complexity of writing and validating affinity rules often poses a challenge, especially when combining many `matchExpressions` operators (e.g., `Exists`, `DoesNotExist`, `In`, `NotIn`, `Gt`, `Lt`).

Taints [16] allow nodes to repel pods lacking corresponding tolerations. Specified via key-value pairs coupled with an effect (e.g., `NoExecute`), they commonly restrict access to nodes with specialized hardware. However, this method only operates with simple matching, and does not allow for the expression of complex constraints.

While native Kubernetes scheduling capabilities allow control over pod placement, they must be embedded in each pod specification by the developer. Errors in their definition directly leads to policy violations, such as workload executing on unauthorized nodes.

B. Use cases

Among the various practical scenarios that can benefit from automated and verified scheduling constraints, we focus on: (i) data sovereignty, which mandates how data must be processed and stored based on its origin, (ii) multi-tenancy, which regulates how applications deployed by different entities may coexist in a cluster, and (iii) workload security rings, which define the boundaries between workloads associated with different sensitivity levels. Our goal is to devise a policy-driven mechanism that generates the correct scheduling constraints based on simple metadata that classify the workload.

Data Sovereignty. Regulations such as GDPR or CCPA define a series of data protection standards to protect citizens' data. Companies that violate these laws risk significant penalties. A common and prudent approach is to prevent data from crossing jurisdictions, and this is done by deploying services based on the geographical region. This task can be automated using metadata to classify the origin, along with a company governance policy. To give an example, our solution automates this by translating a `data-sovereignty` label (e.g., `eu`) into the appropriate node affinity rules, ensuring the workload is always deployed in accordance with the company guidelines.

Multi-tenancy. In Kubernetes, multi-tenancy refers to the practice of sharing a cluster among multiple teams or customers. Dedicated node sets are commonly used to this end, reducing the risk of noisy neighbors and lateral movements. Instead of asking the developer to partition the cluster with constraints, our solution uses namespace information to automatically schedule workloads based on the tenant, enhancing isolation even in the case of shared control plane components.

Workload Security Rings. The application of the same security measures to all Kubernetes workloads can lead to increased security risks and inefficiency. For instance, co-locating untrusted applications with sensitive ones significantly expands the attack surface, while heavily constraining verified workloads introduces unnecessary overhead and slows down the development. Workload Security Rings (WSR) [12] addresses this by categorizing applications based on sensitivity, and then by implementing category-specific protection measures. With our solution this behavior is replicated by including sensitivity-related labels in the specification, which are then automatically evaluated by the cluster-level policy, simplifying development and centralizing policy enforcement.

Note that while for simplicity our description treats these use cases as separate entities, in practice they can be combined. For instance, a workload may be classified as both `data-sovereignty: eu` and `security-ring: critical`, and our solution ensures that the pod is scheduled on nodes that satisfy both conditions.

III. Our approach

This section clarifies the threat model, then illustrates our solution, discussing the advantages and the open challenges.

Threat model. The execution of workloads with incompatible security requirements on the same physical node increases the likelihood of service interruptions and security incidents [6]. Indeed, vulnerable applications that process unverified input or run untrusted third-party code can be used to target confidential or critical co-located services. Recent examples of vulnerabilities that can be leveraged for this purpose

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

include the use of buggy third-party libraries [3], incorrect management of requests in Kubernetes environments [4], and flaws in the kernel interface [5] or in the container runtime [2]. This paper aims at mitigating this issue by introducing fine-grained scheduling constraints to improve the isolation between workloads with incompatible security requirements. Sandboxing techniques and the use of namespaces can be used in conjunction with our approach.

A. Our solution

In a Kubernetes cluster, whether issued by developers or controllers, all pod creation requests are sent to the *Kubernetes API server*. By default, they undergo mandatory authentication and authorization to verify the requester's identity and their permissions to modify the cluster's state. Upon successful completion of these stages, the API server initiates the admission control process, which consists of mutation and validation phases. Our approach modifies the default admission control workflow by redirecting all requests to *OPA Gatekeeper* [17], an open source policy and governance framework. This framework leverages *Open Policy Agent* (OPA) [18], a general-purpose policy engine that enforces unified, declarative policies across systems.

As shown in Figure 1, OPA Gatekeeper receives the complete pod specification from the API server and applies mutations based on cluster-defined policies. Mutation entails the automatic injection or modification of scheduling-related fields, such as `nodeSelector`, `affinity` (including `nodeAffinity`, `podAffinity`, and `podAntiAffinity`), and `tolerations`, tailored to specific use cases. To give an example, a data sovereignty policy may translate a `data-sovereignty: eu` label into a `nodeAffinity` rule, ensuring the pod is scheduled only on nodes labeled with `topology.kubernetes.io/region` belonging to the European Union.

After the mutation, OPA Gatekeeper is used in the validation stage to verify that pod specifications comply with governance rules. This also permits checking requests that are not mutated (i.e., they do not need to be augmented, according to the mutation policy) or contain errors. In detail, validation policies leverage two Gatekeeper *Custom Resource Definitions*: (i) *Constraint Templates*, which encapsulate policy logic written in the Rego language [19] and configurable parameter schemas, and (ii) *Constraints*, which instantiate templates, specifying parameters and targeting resources based on criteria such as resource kind, namespace, or use case-specific labels.

When a pod specification fails validation, Gatekeeper rejects the request, providing immediate feedback to the requester. When instead validation is successful, the API server persists the pod specification to *etcd*, the distributed key-value store that maintains the cluster's state and configuration. The *kube-scheduler* then processes the persisted pod, evaluating its scheduling constraints (augmented during mutation) to select the most suitable node for deployment.

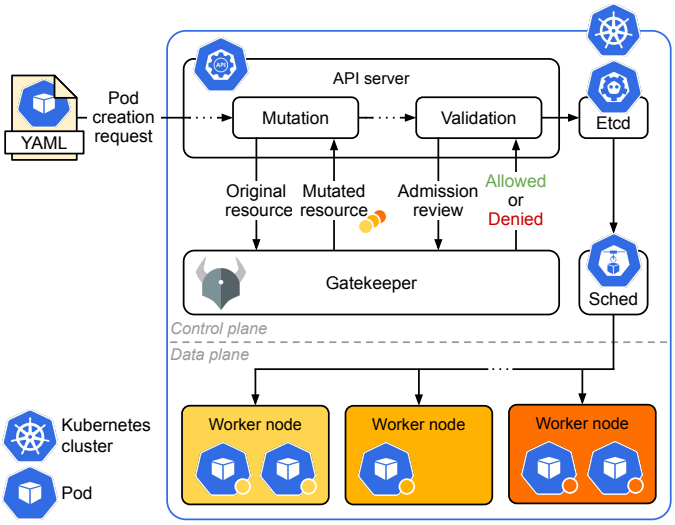


Fig. 1: Kubernetes pod creation workflow, highlighting the integration of OPA Gatekeeper for policy enforcement through the mutation and validation hooks. Our solution uses this architecture to automate cluster-wide policy enforcement that then drives the scheduling process

B. Advantages

By automating the translation of high-level security labels into specific scheduling constraints with Gatekeeper, our solution offers several key advantages.

Centralized policy. Workload isolation is enforced consistently based on a single cluster-defined policy. This significantly reduces the chances of misconfigurations and policy drift, i.e., minimizes possible discrepancies between the defined policy and the actual state of the cluster over time.

Simplified pod definition. With our solution, developers are no longer required to write complex and error-prone scheduling constraints by hand. Simple use-case-related labels are sufficient to assign a workload to a policy-compliant node.

Easier governance updates. When a central policy update is issued, Gatekeeper's validation capabilities can be directly used to detect workloads that violate the new security definitions, and therefore need to be migrated to policy-compliant nodes. Without our approach, a manual inspection is necessary.

In essence, our solution provides a powerful abstraction layer that enhances Kubernetes' native scheduling capabilities. It empowers cluster administrators to define and enforce sophisticated scheduling policies centrally, while allowing developers to focus on the security characteristics of their applications through a simplified, label-driven approach. We believe these features are crucial for managing increasingly complex multi-tenant Kubernetes environments.

C. Open challenges

The security benefits of node isolation come at the cost of reduced cluster utilization when an excessive number of security categories is used. Repurposing nodes offers a potential

TABLE I: Latency measures for mutation, validation, and scheduling phases. Mutation and validation policies are always evaluated when our solution is enabled (*constrained* and *unconstrained* cases), although no mutation is performed on unconstrained pod creation requests

Use case	Test	Latency P50 [ms]			Latency P90 [ms]			Latency P99 [ms]		
		Mutation	Validation	Scheduling	Mutation	Validation	Scheduling	Mutation	Validation	Scheduling
Data sovereignty	Baseline	—	—	0.51	—	—	0.92	—	—	1.70
	Unconstrained	0.50	0.50	0.52	0.90	0.90	0.93	0.99	0.99	1.80
	Constrained	0.50	0.50	0.58	0.90	0.90	1.40	0.99	0.99	4.30
Multi-tenancy	Baseline	—	—	0.51	—	—	0.92	—	—	1.68
	Unconstrained	0.50	0.50	0.52	0.90	0.90	0.93	0.99	0.99	1.82
	Constrained	0.50	0.50	0.55	0.90	0.90	0.98	0.99	0.99	3.69
Workload security rings	Baseline	—	—	0.51	—	—	0.92	—	—	1.75
	Unconstrained	0.50	0.50	0.52	0.90	0.90	0.93	0.99	0.99	1.87
	Constrained	0.50	0.50	0.54	0.90	0.90	0.98	0.99	0.99	3.51

solution, but introduces operational overhead. Indeed, transitioning nodes between security domains requires rigorous sanitization beyond workload draining. This includes system integrity checks to prevent persistent attacks.

Robust isolation requires accurate maintenance of node and pod labels. This activity can be supported by tools that assign labels based on hardware features and system configuration such as *Node Feature Discovery* [20] and by CI/CD pipelines.

IV. Design validation

We validated the design of our proposal by showcasing its ability to support the use cases described in Section II-B. First, we prepared a small local High Availability (HA) cluster with *kind* 2.0.4 with three control plane nodes and five workers nodes, and integrated OPA Gatekeeper 1.0.10 as explained in Section III. Then, we authored and deployed a cluster policy to model each use case.

To test the effectiveness of the mutation and validation stages, we prepared a test suite with several pod specifications covering multiple cases. For each of them, we issued a pod creation request to the Kubernetes API server and observed the mutation and validation decisions as well as the resulting state of the cluster.

The correctness of the mutation stage was tested by using the following criteria:

- M_1 Pod definitions not covered by the use cases must be left unaltered.
- M_2 Pod definitions covered by the use cases that already report scheduling constraints relevant to the use case must not be modified.
- M_3 All other pod definitions covered by the use cases must be mutated with the injection of the required scheduling constraints.

On the other hand, the effectiveness of the validation stage was evaluated as follows:

- V_1 Validation must not be applied to pod specifications unaffected by the use cases.
- V_2 Validation must reject pod specifications that are covered by the use cases but lack any scheduling constraints required by the use case.
- V_3 Validation must reject pod specifications that are covered by the use cases and include a misconfiguration of the scheduling constraints.
- V_4 All other pod specifications covered by the use cases must be admitted for deployment.

Rather than overwriting malformed pod specifications, we opted to return a clear message reporting the error, along with directives on how to fix it. Our experiments confirm that scheduling constraints are added to pod specifications as expected, and their validation did not report any deviations from the expected behavior.

V. Performance Evaluation

To assess the practical applicability of our solution, we conducted a series of experiments to evaluate the scheduling overhead introduced by our solution, as well as the runtime performance of workloads deployed in a cluster with our solution enabled. The tests were executed on a Linux server with kernel 6.15, a 12 core AMD Ryzen 8 7900X CPU, 64 GiB of RAM, and a 2 TB SSD. We used *kind* 2.0.4 to run Kubernetes locally, and *containerd* 2.0.4 as the container runtime. ClusterLoader2 [21] was used to issue pod creation requests, while test metrics were collected with Prometheus 2.25.0.

A. Scheduling overhead

The changes introduced by our proposal to the default Kubernetes admission process are contained, but we still expect to introduce some overhead during pod creation. So, in this section we evaluate the scheduling overhead introduced by our solution, which is the sum of the overhead introduced by the mutation and validation phases of the admission process, as well as the scheduling latency introduced by the scheduling algorithm itself.

The first challenge we faced was to accurately reproduce a large cluster (e.g., 1k nodes) on our test machine. To this end, we conducted a preliminary experiment comparing two cluster configurations, one with real worker nodes and one with simulated worker nodes (i.e., nodes that behave like real ones but do not execute the containerized application). Considering the limited computing resources at our disposal,

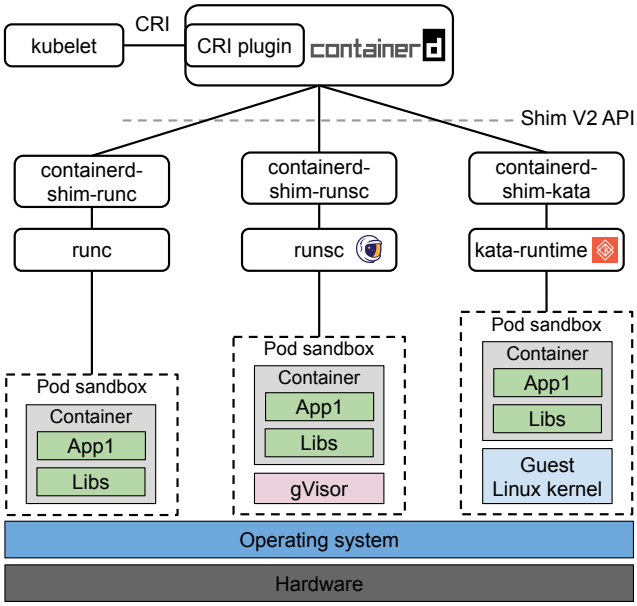


Fig. 2: Runtime overhead test environments

this test was conducted with a small HA cluster with three control plane nodes and five worker nodes. The emulation of worker nodes in the simulated configuration was performed with Kubernetes WithOut Kubelet (KWOK) v0.6.1 [22]. As expected, there was no significant difference between the two cluster configurations in terms of pod admission and scheduling latency, as these operations are performed by control plane nodes (i.e., not by workers). We therefore scaled the number of simulated worker nodes to 1k for the next experiment.

In the second experiment we measured the admission and scheduling latency when pods are deployed in a default Kubernetes cluster (baseline), and when pods are created in a cluster where our solution is enabled. In this second case, we also highlighted the difference between the overhead associated with pod creation requests that are mutated and validated by OPA Gatekeeper (constrained pods), as well as requests that are checked and validated but not mutated (unconstrained pods). The 50th, 90th, and 99th percentiles of latency measures are reported in Table I.

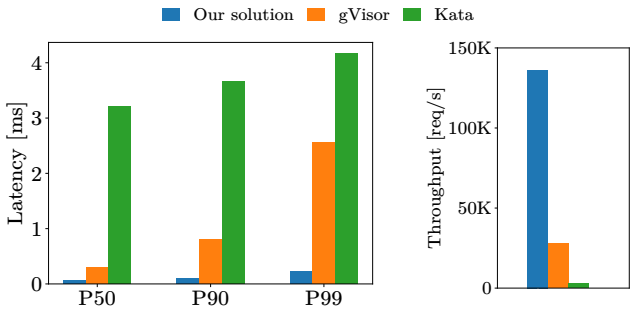
Comparing the baseline and a cluster with our solution, for all the use cases, we measured a total admission overhead (i.e., both mutation and validation) that ranges between 1.00 ms and 1.98 ms. Looking carefully at the mutation and validation percentiles, we can notice that there are no visible differences between the use cases, this is also true for constrained requests, suggesting that the longest time is spent invoking OPA Gatekeer rather than evaluating the different use case policies. Turning our attention to the scheduling time, we can notice an interesting behavior. First, for all the use cases, there is a minor variation between the scheduling time of the baseline and the unconstrained case (consistently less than a fraction of ms). This is also true for the constrained case, except for the P99. We investigated the reason and found out that a small number of constrained scheduling requests experienced high overhead due to unbalances in the partitioning of nodes that negatively affected the default scheduling algorithm. Also, we can notice that the latency distribution is characterized by a long tail (i.e., P99 is significantly different compared to P50 and P90). As a consequence, a fraction of requests are associated with high latency. In this particular case this was due to the high number of requests compared to the available control plane nodes.

Despite all these differences, we would like to point out that even in the worst case represented by P99, the total overhead experienced by a constrained request was 4.58 ms compared to the baseline. These results confirm the applicability of our proposal for large clusters, as the scheduling overhead is negligible especially when amortized by the workload runtime (i.e., a generic workload can live for minutes, hours, or days).

B. Runtime overhead

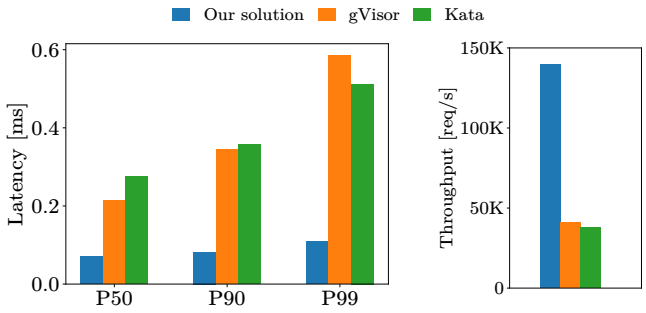
While we do not expect our solution to introduce runtime overhead, in this section we investigate the runtime performance benefits of employing host containers compared to relying solely on sandboxing technologies. The rationale is that, thanks to the automated policy-driven scheduler, we can ensure workloads with equivalent security boundaries are co-located on dedicated nodes, so in some practical scenarios (e.g., verified services not processing untrusted data) an additional sandboxing layer, beyond what is already provided by containers, can be redundant. To this end, we collected metrics associated with the execution of common software in: (i) host containers scheduled by our solution, (ii) containers isolated from the host by a user space kernel like gVisor [10], and (iii) containers deployed in dedicated QEMU virtual machines by Kata [9]. The architecture of these environments is summarized in Figure 2.

We first analyzed the impact of sandboxing with two common web servers, i.e., Lighttpd and Nginx. Both servers were deployed in the described test environments, and exposed on the network through a Kubernetes Service [23]. To collect the results, we employed *wrk* [24] on the host to repeatedly download the default web server page for 2 minutes over 10 concurrent HTTP connections. The results are shown in Figure 3 and 4. Looking at the measures, both gVisor and Kata Containers introduce a significant latency overhead. Indeed, taking the P99 measures as a reference for comparison, systematically applying sandboxing to Nginx rather than carefully co-schedule pods with compatible security requirements is associated with a roughly 10.9x latency increase with gVisor and roughly 17.7x with Kata (roughly 5.4x and 4.7x for Lighttpd, respectively). The throughput reduction was also significant, as the performance drop was at least 70.6% for the two web servers in all sandboxed configurations. These results were expected, as short-lived requests are particularly affected by both sandboxing methods. Indeed, gVisor intercepts and re-implements system calls in user space, while with Kata the HTTP requests are dispatched to a separate virtualized environment before being served, hence they are penalized by a fixed delay. It is worth mentioning that Nginx's architecture



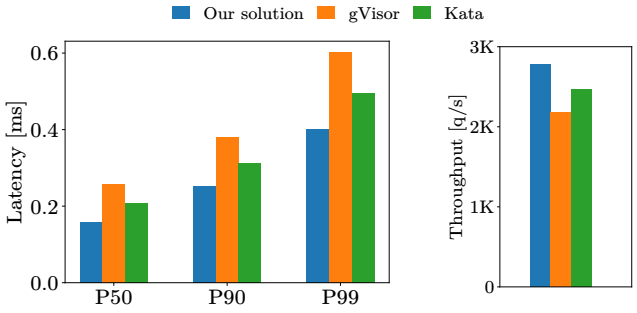
(a) Latency (b) Throughput

Fig. 3: Latency and average throughput of Nginx



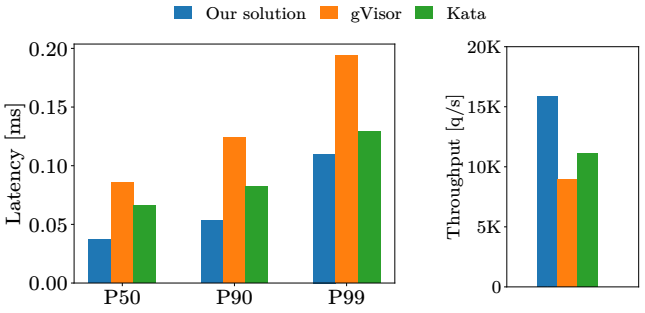
(a) Latency (b) Throughput

Fig. 4: Latency and average throughput of Lighttpd



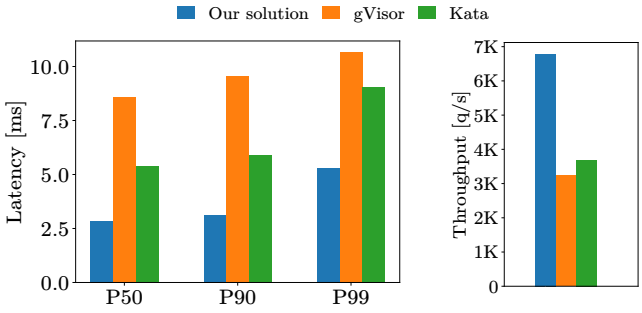
(a) Latency (b) Throughput

Fig. 5: Latency and average throughput of MongoDB



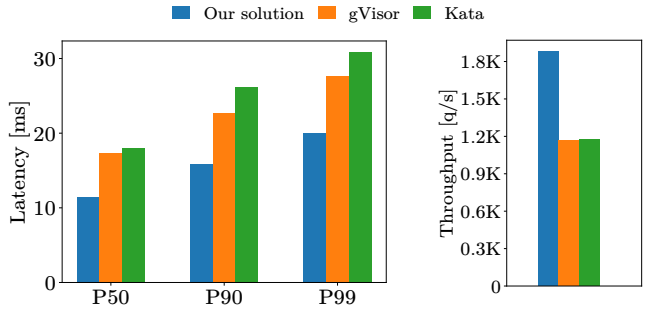
(a) Latency (b) Throughput

Fig. 6: Latency and average throughput of Redis



(a) Latency (b) Throughput

Fig. 7: Latency and average throughput of PostgreSQL



(a) Latency (b) Throughput

Fig. 8: Latency and average throughput of MySQL

is significantly more complex than Lighttpd's, and its high-performance I/O and frequent system calls are more sensitive to Kata's hypervisor and virtualized network stack.

Nonetheless, we would like to point out that, despite the reduced throughput, both gVisor and Kata Containers provide strong guarantees that are necessary for the execution of unverified or third-party workloads.

For the second test, we analyzed two NoSQL DBMSs, i.e., MongoDB and Redis. After deploying them, we ran the open source Yahoo! Cloud Serving Benchmark (YCSB) [25] to assess their performance through a 50% read and 50% write workload. The results are shown in Figure 5 and 6. Comparing theses results with the ones obtained for web servers, gVisor and Kata Containers introduced a significantly smaller (albeit non-negligible) overhead. Indeed, the drop in throughput never exceeded 43.8% for both DBMSs. It is worth noticing that Kata Containers showcased slightly better performance than gVisor, the reason being that gVisor employs a filesystem proxy [26], hence imposing a larger overhead on applications that access the disk frequently.[2]

Finally, to complement the previous category, we evaluated the performance of two relational database management systems, i.e., PostgreSQL and MySQL. In this case, the performance was assessed with the open source *sysbench* suite [27]

[2]https://gvisor.dev/docs/user_guide/filesystem/

with a 77% read and 23% write workload, as implemented in the standard `oltp_read_write.lua` script provided by the sysbench community, for 2 minutes on a 10-MiB indexed table with 10'000 rows. The results are reported in Figure 7 and 8. Again, gVisor and Kata introduced a non-negligible overhead. Interestingly, while PostgreSQL showed better performance with Kata than gVisor, in MySQL the two sandbox showcased comparable results.

VI. Related Work

The research community has extensively studied scheduling techniques in cloud environments [28]–[30], with the primary goal of increasing the efficiency and reducing costs [31]–[35]. Recent proposals have also focused on energy-efficient scheduling in Kubernetes. Indeed, Piontek et al. [36] propose a CO_2-aware workload scheduler, which is able to shift non-critical jobs in time based on their predicted carbon emissions, hence reducing data center emissions while guaranteeing Quality of Service. Rao et al. [37] introduce an energy-aware scheduling algorithm based on Service Level Agreement to reduce the energy consumption of Kubernetes microservices, reducing it by at least 5% in a typical cloud environment. On the other hand, Anouar et al. [38] propose a theoretical framework that enhances the Kubernetes scheduler through reinforcement learning in order to improve decision-making processes. Their architecture is able to adapt to dynamically changing clusters, allowing for potential benefits in resource allocation efficiency.

A promising research line [39]–[44] explores in more detail security-aware scheduling, with the goal of mitigating the impact of possible vulnerabilities, and therefore reducing the attack surface in the cluster. Our proposal complements these works by providing a way to schedule workloads based on a cluster-defined policy, which prevents the deployment of sensitive or critical workloads to untrusted nodes.

Sandboxing is a complementary technique that can be used alongside our solution to increase the isolation between workloads running on the same physical node. It allows for higher cluster utilization, but it also introduces a non-negligible overhead (Section V-B). Sandboxing can be introduced with several frameworks. For instance, gVisor [10] intercepts pods' system calls and re-implements them in user space, offering strong security guarantees at the expense of a higher overhead and possible compatibility issues (not all Linux system calls are available). Virtual machines (e.g., [8], [9], [45]) run each container in a dedicated kernel, dramatically increasing security and reducing the likelihood of a container escape, but they are characterized by higher resource utilization and slower bootstrap time. To overcome these issues, other solutions (e.g., [46]–[51]) propose to use in-kernel sandboxing technologies such as Landlock and eBPF, however, they also expose a wider attack surface due to the shared kernel.

Isolation of vulnerable nodes has also been researched by Google with Workload Security Rings [12], in which the risk of lateral movement is mitigated preventing the co-scheduling of sensitive and untrusted workloads. This solution is unfortunately only available in Borg [13], Google's cluster management system. Our solution can replicate this behavior in Kubernetes.

VII. Conclusions

In this paper we presented an approach to enforce scheduling policies defined once at the cluster level and automatically applied whenever a pod creation request is submitted to Kubernetes. We evaluated the effectiveness of this solution by demonstrating its application to common governance policies including data sovereignty, multi-tenant workloads, and workload security rings. Our approach introduced only minimal overhead during pod creation. Future work will (i) explore the use of the experimental Common Expression Language, i.e., expressions that are evaluated directly within the API server providing an efficient alternative to webhooks, for the mutation and validation constraints; and (ii) assess the approach in multi-cluster environments.

Acknowledgments

This work was supported in part by the European Commission under project GLACIATION (01070141), by the Italian Ministry of University and Research (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] "CNCF Annual Survey," https://www.cncf.io/reports/cncf-annual-survey-2023/, 2023.
[2] "CVE-2024-21626," https://nvd.nist.gov/vuln/detail/cve-2024-21626.
[3] "CVE-2021-44228," https://nvd.nist.gov/vuln/detail/CVE-2021-44228.
[4] N. Ohfeld, R. Shustin, S. Tzadik and H. Ben-Sasson, "IngressNightmare: 9.8 Critical Unauthenticated Remote Code Execution Vulnerabilities in Ingress NGINX," https://www.wiz.io/blog/ingress-nginx-kubernetes-vulnerabilities, 2025.
[5] "CVE-2022-0185," https://nvd.nist.gov/vuln/detail/CVE-2022-0185.
[6] "Kubernetes adoption, security, and market trends report," https://www.redhat.com/en/resources/kubernetes-adoption-security-market-trends-overview, 2024.
[7] N. Lacasse, "Open-sourcing gVisor, a sandboxed container runtime," https://cloud.google.com/blog/products/identity-security/open-sourcing-gvisor-a-sandboxed-container-runtime.
[8] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D. Popa, "Firecracker: Lightweight Virtualization for Serverless Applications," in *NSDI*, 2020.
[9] "Kata Containers," https://katacontainers.io/, 2025.
[10] E. G. Young, P. Zhu, T. Caraza-Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "The true cost of containing: a gVisor case study," in *HotCloud*, 2019.
[11] "Pod Topology Spread Constraints," 2025.
[12] M. Czapiński and R. Wolafka, "Workload Security Rings," https://www.usenix.org/publications/loginmisc/workload-security-rings, 2023.
[13] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *EuroSys*, 2015.
[14] "Assign Pods to Nodes with Node labels," https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/, 2025.
[15] "Assign Pods to Nodes using Affinity," https://kubernetes.io/docs/tasks/configure-pod-container/assign-pods-nodes-using-node-affinity/, 2025.
[16] "Taints and Tolerations – Kubernetes," https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/, 2025.
[17] "OPA Gatekeeper," https://open-policy-agent.github.io/gatekeeper/website/, 2025.
[18] "Open Policy Agent," https://www.openpolicyagent.org/, 2025.

[19] "Policy Language," https://www.openpolicyagent.org/docs/latest/policy-language/, 2025.

[20] "Node feature discovery," https://github.com/kubernetes-sigs/node-feature-discovery, 2025.

[21] "ClusterLoader2," https://github.com/kubernetes/perf-tests/tree/master/clusterloader2, 2025.

[22] "KWOK," https://github.com/kubernetes-sigs/kwok, 2025.

[23] "Service," https://kubernetes.io/docs/concepts/services-networking/service/, 2025.

[24] W. Glozer, "wrk: Modern HTTP Benchmarking Tool," https://github.com/wg/wrk.

[25] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *SoCC*, 2010.

[26] "Filesystem," https://gvisor.dev/docs/user_guide/filesystem/, 2025.

[27] A. Kopytov, "sysbench: Scriptable database and system performance benchmark," https://github.com/akopytov/sysbench.

[28] C. Carrión, "Kubernetes Scheduling: Taxonomy, Ongoing Issues and Challenges," *ACM Computing Surveys*, 2022.

[29] Z. Rejiba and J. Chamanara, "Custom Scheduling in Kubernetes: A Survey on Common Problems and Solution Approaches," *ACM Computing Surveys*, 2022.

[30] M. Rossi, M. Beretta, D. Facchinetti, and S. Paraboschi, "POSTER: Policy-driven security-aware scheduling in Kubernetes," in *ASIACCS*, 2025.

[31] G. Vallée, C. Morin, J.-Y. Berthou, and L. Rilling, "A new approach to configurable dynamic scheduling in clusters based on single system image technologies," in *IPDPS*, 2003.

[32] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: scalable and coordinated scheduling for cloud-scale computing," in *USENIX OSDI*, 2014.

[33] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni, "Graphene: packing and dependency-aware scheduling for data-parallel clusters," in *USENIX OSDI*, 2016.

[34] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner, "Memory buddies: exploiting page sharing for smart colocation in virtualized data centers," *SIGOPS Operating Systems Review*, 2009.

[35] A. C. Caminero and R. Muñoz-Mansilla, "Quality of Service Provision in Fog Computing: Network-Aware Scheduling of Containers," *Sensors*, 2021.

[36] T. Piontek, K. Haghshenas, and M. Aiello, "Carbon emission-aware job scheduling for kubernetes deployments," *Journal of supercomputing*, vol. 80, pp. 549–569, 2023.

[37] W. Rao and H. Li, "Energy-aware scheduling algorithm for microservices in kubernetes clouds," *Journal of Grid Computing*, vol. 23, no. 1, p. 2, 2025.

[38] H. Anouar, H. Hatim, and E. A. Zineb, "Proposing a theoretical energy aware framework for kubernetes scheduling using reinforcement learning," in *International Conference on Advanced Intelligent Systems for Sustainable Development*. Springer, 2024, pp. 849–857.

[39] T. Xie and X. Qin, "Scheduling security-critical real-time applications on clusters," *IEEE Transactions on Computers*, 2006.

[40] T. Xiaoyong, K. Li, Z. Zeng, and B. Veeravalli, "A Novel Security-Driven Scheduling Algorithm for Precedence-Constrained Tasks in Heterogeneous Distributed Systems," *IEEE Transactions on Computers*, 2011.

[41] J. Han, W. Zang, S. Chen, and M. Yu, "Reducing Security Risks of Clouds Through Virtual Machine Placement," in *DBSec*, 2017.

[42] Y. Han, J. Chan, T. Alpcan, and C. Leckie, "Using Virtual Machine Allocation Policies to Defend against Co-Resident Attacks in Cloud Computing," *IEEE Transactions on Dependable and Secure Computing*, 2017.

[43] M. Bahrami, A. Malvankar, K. K. Budhraja, C. Kundu, M. Singhal, and A. Kundu, "Compliance-Aware Provisioning of Containers on Cloud," in *IEEE CLOUD*, 2017.

[44] M. V. Le, S. Ahmed, D. Williams, and H. Jamjoom, "Securing Container-based Clouds with Syscall-aware Scheduling," in *ASIA CCS*, 2023.

[45] "QEMU," www.qemu.org, 2025.

[46] M. Abbadini, M. Beretta, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi, "Lightweight Cloud Application Sandboxing," in *CLOUDCOM*, 2023.

[47] M. Rossi, M. Beretta, D. Facchinetti, and S. Paraboschi, "POSTER: Transparent Temporally-Specialized System Call Filters," in *ASIACCS*, 2025.

[48] M. Abbadini, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi, "Cage4Deno: A Fine-Grained Sandbox for Deno Subprocesses," in *ASIACCS*, 2023.

[49] Y. Sun, D. Safford, M. Zohar, D. Pendarakis, Z. G., and T. Jaeger, "Security Namespace: Making Linux Security Frameworks Available to Containers," in *USENIX Security*, 2018.

[50] M. Abbadini, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi, "NatiSand: Native Code Sandboxing for JavaScript Runtimes," in *RAID*, 2023.

[51] M. Abbadini, M. Beretta, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi, "POSTER: Leveraging eBPF to enhance sandboxing of WebAssembly runtimes," in *ASIACCS*, 2023.