Received 22 June 2026, accepted 15 September 2026, date of publication 30 September 2026
www.computingonline.net / computing@computingonline.net

Print ISSN 1727-6209
Online ISSN 2312-5381
DOI 10.47839/ijc.25.3.4858

A Formal Model of the Kubernetes Scheduler Filtering Phase for Dependability Analysis of Pod Placement

MYKOLA BURIAK[1], GEORGE VOROBETS[2], STEPAN MELNYCHUK[2]
[1]Department of Computer Systems Software, Yuriy Fedkovych Chernivtsi National University, Chernivtsi, Ukraine
[2]Department of Computer Systems and Networks, Yuriy Fedkovych Chernivtsi National University, Chernivtsi, Ukraine
Corresponding author: Mykola Buriak (e-mail: buriak.mykola@chnu.edu.ua).

ABSTRACT This paper studies the filtering phase of the Kubernetes scheduler (kube-scheduler) and how this phase affects the dependability of pod placement in a cluster. The goal is to build a formal mathematical model of the filtering phase that makes it possible to analyze the risks to availability and fault tolerance of pod placement in advance, before deployment. The tasks are: to formalize the five main filtering predicates (node affinity, taints and tolerations, pod affinity, pod anti-affinity, and resource sufficiency) within a single formal framework; to combine them into a complex feasibility predicate; and, on this basis, to derive conditions for detecting unschedulable configurations and for verifying the correctness of replica spreading. The work uses mathematical logic (predicates, indicator functions, quantifiers) and set theory. The results. It is shown that the proposed model makes it possible to formally detect configurations in which a pod cannot be placed on any node (the risk of a Pending state), and to check whether the anti-affinity mechanism will indeed spread replicas across different failure domains. The use of the model is illustrated by a worked example of a small cluster. Conclusions. The model provides simple conditions, computable directly from the configuration and state of the cluster, that reveal dependability defects which the standard Kubernetes scheduler detects only at runtime. The scientific novelty lies in the application of a formal abstraction of selected filtering predicates to the analysis of availability and fault tolerance of pod placement before deployment.

KEYWORDS kubernetes; resources optimization; containerization; analysis of resources usage; clustering; distributed systems; cloud computing; modeling; filtering; queuing systems

I. INTRODUCTION

Today, Kubernetes is the most widely used platform for orchestrating containerized applications in cloud and edge environments. The decision about which node to run a particular pod on is made by the kube-scheduler, and it is these decisions that directly determine the dependability of the system – its availability and fault tolerance.

The first phase of the scheduler's work is filtering, which screens out the nodes on which a pod cannot be run. Configuration errors at this stage have direct consequences for reliability: an incorrectly defined anti-affinity rule may place all replicas of a service on a single node, creating a single point of failure; overly strict constraints may leave a pod unschedulable on every node, so that it remains in the Pending state, which means the service is effectively unavailable; and insufficient resource checking may overload a node.

In practice, such problems are mostly discovered after the fact – once the workload has already been deployed – through trial and error or through incidents in operation. There is a lack of a formal framework that would make it possible to analyze the risks to availability and fault tolerance of a placement before deployment and directly from the configuration of the pods and nodes. This justifies building a formal model of the filtering phase oriented toward dependability analysis.

II. STATE OF THE ART

Kubernetes has become the de facto standard for running containerized applications, building on the experience of Google's earlier cluster management systems, Borg and Omega [1]. The decision about which node to run a pod on is made by the kube-scheduler. It works in two steps: first, filtering removes the nodes where the pod cannot run, and then scoring selects the best node among the remaining ones [2]. Placement can be controlled through several mechanisms – node and pod affinity and anti-affinity, node protection through taints/tolerations [3], even distribution of pods by topology (topology spread constraints) [4], and specifying the required resources through requests and limits [5]. The availability and fault tolerance of the system depend directly on these settings: correct anti-affinity spreads the replicas of

a service across different failure domains, while a configuration mistake can place all replicas on a single node (a single point of failure) or even leave the pod in the Pending state. A general review of container technologies and their orchestration is given in [6], a detailed classification of scheduling methods in Kubernetes in [7], and the development directions of cloud computing with a focus on reliability in [8]. These works show that interest in the topic is growing quickly, and at the same time they outline the main directions from which our work differs.

Many studies treat the placement of containers and virtual machines as an optimization or constraint satisfaction problem. Placement is formulated as an integer linear programming problem with several objectives [9], different algorithms are compared by energy consumption, cost, and node utilization [10], or they are summarized in surveys of resource allocation algorithms [11]. What these works have in common is that they look for the best placement according to certain numerical metrics, but they do not check whether the pod can be placed anywhere at all under the given constraints.

The second notable direction is the use of machine learning and reinforcement learning for custom schedulers. Schedulers based on deep reinforcement learning have been proposed for microservices [12] and for compute-intensive pods [13], as well as algorithms that maximize resource utilization [14]. Such solutions choose a scheduling policy by "observing" the cluster during operation, and they do not provide any formal guarantee that placement is possible at all before deployment. This direction also includes works that improve the scoring step and the rapid autoscaling of pods based on metrics collected at runtime [15], [16].

A separate direction is the formal modeling of Kubernetes. A model of the container environment written in Real-Time ABS makes it possible to study the behavior of services before deployment [17], and its continuation is aimed at predicting CPU and memory consumption [18]. Another group of researchers uses Petri nets to model resource management and scheduler performance [19], [20], as well as to characterize and schedule applications in the cluster [21], [22]. These models mostly describe how resource consumption and system performance change over time, while the node selection at the filtering phase is largely left outside the formal description.

In edge and fog environments, hybrid autoscaling mechanisms for serverless platforms are common, and they manage resources already during the operation of the system [23]. The dependability of cloud systems is usually assessed through state models – Markov chains and stochastic reward nets: they are used to compute the availability of large IaaS clouds [24], [25], to generalize the methodology of availability assessment itself [26], and to study how different parameters affect availability [27]. Such models describe the system through failure and recovery rates and calculate the probable downtime during operation, but they do not check whether the placement configuration itself is correct. Similar in name but not in essence are works on "proactive" resource management [28], "safe" and optimal management of microservices [29], and resource management through neural networks [30]: here proactivity means forecasting the load, not checking the very possibility of placing a pod. Earlier, the authors proposed a method of granular analysis of shared resource utilization in a cluster, where each resource is divided into states – allocated, reserved, used, and free [31]; this method will serve as the basis for further integration with the formal model of filtering.

In practice, placement can also be checked by dry run tools such as cluster-capacity [32] and kube-scheduler-simulator [33]. These tools run the scheduler code in a separate environment on a snapshot of the cluster state and give the answer for one concrete pod and one concrete state. The model in this paper describes the filtering predicates as set operations over data already present in the cluster API. Conditions can therefore be evaluated in the working cluster itself, for example in the admission path, without running the scheduler. The model also checks the fault tolerance and the placement margin of the result, which these tools do not compute.

Thus, despite the significant amount of work done, the existing studies solve other parts of the problem: some look for the best placement according to numerical metrics [9-11], others choose a scheduling policy through learning [12-16], and others assess the dependability and resource behavior of already deployed systems [23-27], while the formal models of Kubernetes itself focus on performance and resource consumption [17-22]. What is still missing is an approach that checks the dependability of a placement before deployment, directly from the configuration of the pods and the nodes. Such an approach has to describe the predicates of the filtering phase formally and then derive checkable conditions from them. These conditions have to detect configurations in which a pod cannot be placed anywhere and to name the predicate that caused this, to check the spread of replicas across failure domains, and to estimate the placement margin. It is exactly this, previously unsolved, part of the problem that this paper addresses.

III. OBJECTIVES AND TASKS

The aim of the work is to build a formal mathematical model of the filtering phase of the kube-scheduler that makes it possible to analyze the risks to availability and fault tolerance of pod placement in a cluster in advance. To achieve this aim, the following objectives have been addressed:

- to introduce a single formal framework (sets of nodes and pods, labels, and the general form of a filtering predicate);
- to formalize the five main filtering predicates as boolean functions of a (pod, node) pair;
- to combine the predicates into a complex feasibility predicate and to define the set of feasible nodes;
- to derive from the model the conditions for verifying the dependability of a placement – detecting unschedulable configurations and checking replica spreading;
- to demonstrate the applicability of the model on the example of a cluster.

The rest of the paper is organized as follows. Section IV builds the formal model of the filtering phase. Section V derives the dependability conditions from the model. Section

VI presents a worked example of a cluster. Section VII contains the conclusions and directions for further research.

IV. FORMAL MODEL OF THE FILTERING PHASE

This section builds the formal model of the filtering phase. First, the scope of the model is stated, then the basic sets and notation are introduced, then the predicates of the five filtering mechanisms are defined, which are finally combined into a complex filter. Each predicate takes a (pod, node) pair and returns 1 if the placement is feasible and 0 otherwise.

A. SCOPE AND ASSUMPTIONS

The model is a partial abstraction of the filtering phase of kube-scheduler. It covers the filter plugins NodeAffinity, TaintToleration, InterPodAffinity, and NodeResourcesFit for CPU and memory. The other filter plugins (NodeName, NodePorts, NodeUnschedulable, VolumeBinding, VolumeRestrictions, VolumeZone, NodeVolumeLimits, and PodTopologySpread) are outside the scope. The model describes one scheduling moment for a fixed set of nodes N. Cluster autoscaling, preemption, eviction, and rescheduling are not modeled. Only required affinity rules are modeled. Node affinity is modeled as one node selector term with match expressions. Pod affinity and anti-affinity consider only the pods in the namespace of the scheduled pod and use one topology key per rule. Only taints with the NoSchedule effect are modeled. The extension to preferred rules, several selector terms, namespace selectors, and the NoExecute effect is left for future work. The semantics of the modeled mechanisms follows the Kubernetes documentation [2], [3], [5] and was checked against Kubernetes v1.35.

B. BASE SETS AND NOTATION

Let N be a finite set of the cluster's compute nodes (1). For each node, $P(n)$ denotes the set of pods already running on it, and the set of all pods is their union (2). Each node and pod is characterized by a set of labels, that is, key–value pairs (3, 4). The general filtering predicate is a boolean function of a (pod, node) pair (5).

$$N = \{n_1, n_2, \ldots, n_k\} \tag{1}$$

$$P = \bigcup_{n \in N} P(n) \tag{2}$$

$$L(n) = \{(k_{n1}, v_{n1}), (k_{n2}, v_{n2}), \ldots, (k_{nm}, v_{nm})\} \tag{3}$$

$$L(p) = \{(k_{p1}, v_{p1}), (k_{p2}, v_{p2}), \ldots, (k_{pl}, v_{pl})\} \tag{4}$$

$$f: P \times N \longrightarrow \{0,1\} \tag{5}$$

where n – node, p – pod, (k_i, v_i) – tuple consisting of a label's key and value.

C. NODE AFFINITY

Node affinity restricts the set of feasible nodes according to the values of their labels. The requirements of a pod p form a set $S(p)$, where each requirement is a tuple consisting of a target key, a comparison operator, and a set of values (6).

$$(k_S, o_S, V_S), o_S \in \{In, NotIn, Exists, DoesNotExist\} \tag{6}$$

where k_S – the target key of a node's label, o_S – the comparison operator, V_S – the set of values of a node's labels. The matching of a node's labels against a single requirement is defined by an auxiliary predicate $match$, whose semantics are determined by the operator (7). A pod is feasible under this criterion if and only if the node satisfies all the requirements simultaneously (8).

$$match(L(n), k_S, o_S, V_S) \Leftrightarrow \begin{cases} \exists (k_n, v_n) \in L(n): (k_n = k_S \wedge v_n \in V_S), & \text{if } o_S = In \\ \forall (k_n, v_n) \in L(n): (k_n = k_S \Longrightarrow v_n \notin V_S), & \text{if } o_S = NotIn \\ \exists (k_n, v_n) \in L(n): (k_n = k_S), & \text{if } o_S = Exists \\ \neg\, \exists (k_n, v_n) \in L(n): (k_n = k_S), & \text{if } o_S = DoesNotExist \end{cases} \tag{7}$$

$$f_{NodeAffinity}(p, n) = \mathbb{1}\{\forall (k_S, o_S, V_S) \in S(p): match(L(n), k_S, o_S, V_S)\} \tag{8}$$

where p – the pod being scheduled, n – the candidate node, $S(p)$ – the set of tuples of requirements for node labels specified in the specification of pod p, $L(n)$ – the set of tuples of labels of node n, $match$ – the auxiliary predicate for validating a node's labels.

D. TAINTS AND TOLERATIONS

This mechanism implements the reverse logic: a node repels pods that do not have the corresponding toleration. A node's taint is a tuple (9), a pod's toleration is a tuple (10), and their compatibility is defined by the predicate $tolerates$ (11). A node is feasible if every taint with the NoSchedule effect is covered by at least one toleration (12).

$$(k_T, v_T, e_T) \tag{9}$$

$$(k_{Tol}, v_{Tol}, o_{Tol}, e_{Tol}), o_{Tol} \in \{Equal, Exists\} \tag{10}$$

$$tolerates(k_T, v_T, e_T, k_{Tol}, v_{Tol}, o_{Tol}, e_{Tol}) \Leftrightarrow (k_{Tol} = k_T) \wedge (e_{Tol} = e_T \vee e_{Tol} = \emptyset) \wedge \big((o_{Tol} = Exists) \vee (o_{Tol} = Equal \wedge v_{Tol} = v_T)\big) \tag{11}$$

$$f_{Taint}(p,n) = \mathbb{1}\{\forall(k_T, v_T, e_T) \in T(n): e_T = NoSchedule \Longrightarrow \exists(k_{Tol}, v_{Tol}, o_{Tol}, e_{Tol}) \in Tol(p): tolerates(k_T, v_T, e_T, k_{Tol}, v_{Tol}, o_{Tol}, e_{Tol})\} \quad (12)$$

where k_T – the taint's key, v_T – the taint's value, e_T – the taint effect, k_{Tol} – the toleration's key, v_{Tol} – the toleration's value, o_{Tol} – the comparison operator, e_{Tol} – the toleration's effect, which must either match the taint's effect or be empty, $T(n)$ – the set of tuples of taints on the node, $Tol(p)$ – the set of tuples of a pod's tolerations, $tolerates$ – the auxiliary predicate for compatibility.

E. POD AFFINITY

Pod affinity requires a pod to be placed together with other pods within a topology domain. The function $meta$ reads the value of a topology label (13); two nodes belong to the same domain if these values coincide (14). The predicate $eval$ checks whether the labels of a running pod correspond to the selectors $S(p)$ under the same operator semantics as in (7) (15). A pod is feasible if there exists a node in the same domain on which a pod satisfying the selectors is running (16).

$$meta(n,k) = v, \quad \exists!\,(k_n, v_n) \in L(n): k_n = k \quad (13)$$

$$meta(n,k) = meta(n',k) \quad (14)$$

$$eval\big(L(p'), S(p)\big) \Leftrightarrow \forall(k_S, o_S, V_S) \in S(p): match(L(p'), k_S, o_S, V_S) \quad (15)$$

$$f_{PodAffinity}(p,n) = \mathbb{1}\Big\{\exists n' \in N: \big(meta(n,k) = meta(n',k)\big) \wedge \Big(\exists p' \in P(n'): eval\big(L(p'), S(p)\big)\Big)\Big\} \quad (16)$$

where n' – a node in the same topology domain as n, $L(n)$ – the set of tuples of the topology labels of node n, k_n – the key of a topology label on the node, v_n – the value of a topology label on the node, k – the key of a topology label, v – the value of a topology label, $P(n')$ – the set of pods placed on node n'.

F. POD ANTI-AFFINITY

Pod anti-affinity is the logical opposite of affinity: no pod in the same domain must satisfy the pod's selectors (the negation of (16)). It is precisely this predicate (17) that is responsible for spreading replicas for fault tolerance, which is key to the dependability analysis in Section V.

$$f_{PodAntiAffinity}(p,n) = \mathbb{1}\Big\{\neg\,\exists n' \in N: \big(meta(n,k) = meta(n',k)\big) \wedge \Big(\exists p' \in P(n'): eval\big(L(p'), S(p)\big)\Big)\Big\} \quad (17)$$

G. RESOURCE CAPACITY

For each resource type, the scheduler checks whether there are enough free resources on the node for the request of a new pod. The free remainder is the difference between the available amount and the sum of the requests of the pods already placed (18). A pod is feasible under the resource criterion if, for each resource type, the free remainder is not less than the pod's request (19).

$$F_r = A_r(n) - \sum_{p'=1}^{|P(n)|} R_r(p'), \quad p' \in P(n), \quad r \in \{cpu, ram\} \quad (18)$$

$$f_{Capacity}(p,n) = \mathbb{1}\left\{\left(A_{cpu}(n) - \sum_{p'=1}^{|P(n)|} R_{cpu}(p') \geq R_{cpu}(p)\right) \wedge \left(A_{ram}(n) - \sum_{p'=1}^{|P(n)|} R_{ram}(p') \geq R_{ram}(p)\right)\right\} \quad (19)$$

where $A_r(n)$ – the amount of resource r available on node n, $R_r(p)$ – the request of pod p, $A_{cpu}(n), A_{ram}(n)$ – the available CPU and memory resources on node n, $R_{cpu}(p), R_{ram}(p)$ – the request for CPU and memory resources of the pod p being scheduled.

The model generalizes to an arbitrary number of resource types by adding analogous inequalities.

H. COMPLEX CLUSTER FILTER

The scheduler combines all the predicates by conjunction: a node is feasible only when every predicate returns 1 (20). The result of the filtering phase is the set of nodes that have passed the complex filter, that is, the space of feasible solutions (21).

$$f_{Feasible}(p,n) = f_{NodeAffinity}(p,n) \wedge f_{Taint}(p,n) \wedge f_{PodAffinity}(p,n) \wedge f_{PodAntiAffinity}(p,n) \wedge f_{Capacity}(p,n) \quad (20)$$

$$N_{Feasible}(p) = \{n \in N \mid f_{Feasible}(p,n) = 1\} \quad (21)$$

If $N_{Feasible}(p) = \emptyset$, the pod cannot be placed on any node and remains in the Pending state. This condition is the starting point for the dependability analysis in the next section.

V. DEPENDABILITY ANALYSIS USING THE MODEL

In this section, the model is used not to describe the scheduler, but as a tool for the analysis of pod placement dependability in advance. Three applications are considered: detecting unschedulable configurations (the risk of unavailability), checking replica spreading (fault tolerance), and assessing placement flexibility (resilience to node failures).

A. DETECTION OF UNSCHEDULABLE CONFIGURATIONS

From the definition of the set of feasible nodes (21), the schedulability criterion follows (22): a pod is schedulable if and only if its set of feasible nodes is non-empty.

$$Schedulable(p) \iff N_{Feasible}(p) \neq \emptyset \quad (22)$$

A violation of this criterion gives an indicator of the risk of a Pending state, computable directly from the configuration (23).

$$U(p) = \mathbb{1}\{N_{Feasible}(p) = \emptyset\} \quad (23)$$

To diagnose the cause, the complex filter is decomposed into the sets of feasible nodes of the individual predicates (24), the overall set is their intersection (25), and the set of blocking predicates (26) localizes the root of the problem.

$$N_X(p) = \{n \in N \mid f_X(p,n) = 1\}, \quad X \in \{NodeAffinity, Taint, PodAffinity, PodAntiAffinity, Capacity\} \quad (24)$$

$$N_{Feasible}(p) = \bigcap_X N_X(p), \quad X \in \{NodeAffinity, Taint, PodAffinity, PodAntiAffinity, Capacity\} \quad (25)$$

$$B(p) = \{X \mid N_X(p) = \emptyset\} \quad (26)$$

where $U(p)$ – the unschedulability indicator, $B(p)$ – the set of predicates that make placement impossible.

If $B(p) \neq \emptyset$, the cause is an individual predicate; if $B(p) = \emptyset$ while $N_{Feasible}(p) = \emptyset$, the configuration contains a conflict between predicates.

B. VERIFICATION OF REPLICA SPREADING

Consider a service with r replicas $R(p) = \{p_1, p_2, \ldots, p_r\}$, and a topology key k. The values of the topology label define the set of failure domains (27). Let a be the placement of the replicas over the nodes. Then (28) gives the number of replicas in a domain d. A configuration has no single point of failure when the replicas occupy at least two domains (29). If at least q replicas out of r have to stay available after the failure of any one domain, it is enough to limit the largest number of replicas in one domain (30).

$$D_k = \{meta(n,k) \mid n \in N\} \quad (27)$$

$$cnt(d) = |\{p_i \in R \mid meta(a(p_i),k) = d\}| \quad (28)$$

$$|\{d \in D_k \mid cnt(d) > 0\}| \geq 2 \quad (29)$$

$$max\{cnt(d) : d \in D_k\} \leq r - q \quad (30)$$

where R – the set of replicas, k – the topology key of a failure domain, D_k – the set of domains, q – the minimum acceptable number of available replicas after a domain failure.

The key observation is that acceptance of a configuration by the scheduler – that is, $f_{PodAntiAffinity} = 1$ for every replica (17) – does not guarantee conditions (29)–(30): for example, when anti-affinity is specified as preferred rather than required, or when there are fewer domains than replicas. The model makes it possible to verify fault tolerance directly, and not merely the validity of a configuration. Under required anti-affinity with key k, at most one replica is placed in each domain, so all replicas are schedulable only under condition (31); otherwise at least (32) replicas stay unschedulable, and this links the result to Section V.A.

$$r \leq |D_k| \quad (31)$$

$$r - |D_k| \quad (32)$$

C. ASSESSMENT OF THE PLACEMENT SPACE REDUCTION

The third application is a quantitative assessment of how much the constraints narrow the placement space. The placement margin is defined as the fraction of feasible nodes (33). A value approaching 0 indicates a fragile configuration: few alternatives and a high risk of Pending in the event of failures or planned node draining; a value of 0 corresponds to unschedulability (Section V.A). In particular, a pod survives the failure of any single one of its feasible nodes and remains schedulable if and only if (34) holds.

$$\mu(p) = |N_{Feasible}(p)| \,/\, |N| \quad (33)$$

$$|N_{Feasible}(p)| \geq 2 \quad (34)$$

where $\mu(p) \in [0,1]$ – the placement margin, $|N|$ – the total number of nodes.

The metric μ makes it possible to compare configurations and to assess their resilience to node failures before deployment.

D. CONNECTION TO THE RESOURCE MODEL

The sets of feasible nodes depend on the current workload through the resource-sufficiency predicate (19): as the load on the nodes changes, $N_{Feasible}(p)$ changes, and therefore so does the margin $\mu(p)$. Integrating the model with the method of granular analysis of cluster resource usage [31] makes it possible to predict the dynamics of $N_{Feasible}(p)$ over time and to detect in advance the moments when the risk of Pending grows. A detailed treatment of this direction is left for further research.

VI. CASE STUDY

This section illustrates the application of the model on a small cluster. A typical scenario is considered: the deployment of a web service with three replicas that, by design, are meant to be spread across availability zones for fault tolerance. It is shown that the model detects – in advance and from the configuration alone – that this design intent is not fulfilled.

A. INPUT DATA: CLUSTER AND SERVICE CONFIGURATION

Consider a cluster of four nodes in two availability zones (topology key $k = zone$, domains $D_k = \{A, B\}$). The node characteristics are given in Table 1.

Table 1. Cluster nodes configuration

Node	Zone	Disk	CPU (free)	RAM (free)	Taint
n_1	A	SSD	4 cores	8 Gi	n/a
n_2	A	HDD	4 cores	8 Gi	n/a
n_3	B	SSD	2 cores	4 Gi	dedicated=gpu:NoSchedule
n_4	B	HDD	4 cores	8 Gi	n/a

The pod specification of the web service, identical for all $r = 3$ replicas, is given in Table 2.

Table 2. Web-service pod configuration

Parameter	Value
Pod label	app = web
Node affinity	disk in {SSD}
Resource requests	CPU = 2 cores, RAM = 4 Gi
Pod anti-affinity	topologyKey = zone, selector app = web
Tolerations	n/a
Replicas	3

B. MODEL RUN FOR THE FIRST REPLICA

Initially the cluster contains no $app = web$ pods. The feasible sets of each predicate (24) and their intersection (25) for the first replica p_1, and for the second replica p_2 after p_1 is placed, are shown in Table 3.

Table 3. Feasible node sets by predicate

Predicate	Feasible for p_1	For p_2 after p_1
Node affinity	$\{n_1, n_3\}$	$\{n_1, n_3\}$
Taint	$\{n_1, n_2, n_4\}$	$\{n_1, n_2, n_4\}$
Capacity	$\{n_1, \dots, n_4\}$	$\{n_1, \dots, n_4\}$
Pod anti-affinity	$\{n_1, \dots, n_4\}$	$\{n_3, n_4\}$

The intersection yields a single feasible node, so the first replica is placed on n_1 (zone A) (35).

$$N_{Feasible}(p_1) = \{n_1, n_3\} \cap \{n_1, n_2, n_4\} \cap \{n_1 \dots n_4\} \cap \{n_1 \dots n_4\} = \{n_1\} \quad (35)$$

The only SSD nodes are n_1 and n_3; n_3 carries a taint without a matching toleration, so the placement space is already reduced to a single node at this step.

C. MODEL RUN FOR THE SECOND AND THIRD REPLICAS: PENDING DETECTION

After p_1 is placed in zone A, the required anti-affinity (17) with key zone excludes all nodes of zone A, so for p_2 the anti-affinity set narrows to $\{n_3, n_4\}$. The intersection with the remaining predicates is empty (36).

$$N_{Feasible}(p_2) = \{n_1, n_3\} \cap \{n_1, n_2, n_4\} \cap \{n_1 \dots n_4\} \cap \{n_3, n_4\} = \emptyset \quad (36)$$

By criterion (22)–(23), pod p_2 is unschedulable, $U(p_2) = 1$, and remains Pending. The set of blocking predicates (26) diagnoses the root cause (37).

$$B(p_2) = \emptyset, N_{Feasible}(p_2) = \emptyset \quad (37)$$

Each individual set $N_X(p_2)$ is non-empty, but their intersection is empty – an over-constrained configuration: the only SSD node in a still-permitted zone (n_3) is blocked by a taint, while the second SSD node (n_1) is already occupied.

The third replica p_3 is analogous to p_2 and is likewise unschedulable, remaining in the Pending state. Thus, the cause is the interaction of node affinity, the taint, and anti-affinity, rather than a single predicate.

D. INTERPRETATION OVER DEPENDABILITY METRICS

Let us apply the conditions of Section V. The single placed replica p_1 is fragile, since its placement margin is minimal and there is no reserve node for rescheduling (38), (39).

$$r = 3 > |D_k| = 2 \;\Rightarrow\; \text{condition (31) is violated} \tag{38}$$

$$\mu(p_1) = |\{n_1\}|/4 = 0.25, \quad |N_{Feasible}(p_1)| = 1 < 2 \;\Rightarrow\; \text{condition (34) is violated} \tag{39}$$

Thus, the model detects three facts in advance:

- only one of the three replicas will be placed, namely on n_1, and the other two will be unschedulable under the current state of the cluster;
- even this replica is not fault-tolerant, because it cannot be rescheduled in the event of the failure or decommissioning of n_1;
- the root of the problem is automatically localized as a conflict between node affinity (SSD), the taint on n_3, and zonal anti-affinity, rather than as the failure of a single mechanism.

E. DISCUSSION

In this scenario, Kubernetes would accept the manifest without any errors: the specification is written correctly, each constraint can be satisfied on its own, and the API server has no way to check how these constraints work together across all the replicas. The problem would appear only at runtime. The first replica would be placed on a node, while the second and third replicas would stay in the Pending state for as long as the cluster remains in this state – that is, until a node is added, a taint is removed, or the constraints are relaxed. An operator would notice this only after deployment and would then have to find the cause by hand: look at the events of the pending pods, compare the node labels with the affinity rule, check the taint on the blocked node, and connect all of this with the zonal anti-affinity. This kind of diagnosis takes a lot of time and is easy to get wrong, because the failure comes from several constraints working together, not from a single one.

The proposed model removes this manual step, because it finds the conflict in advance. Condition (22) shows that the second replica cannot be scheduled even before it is sent to the cluster, and the set of blocking predicates (26) points directly to the causes – here, the combination of node affinity, the taint on the blocked node, and the zonal anti-affinity. It also separates two cases: when a single predicate is too strict, and when the configuration is over-constrained, that is, when every predicate can be satisfied on its own but their intersection is still empty. In addition, the fault-tolerance conditions (29-31) and the placement-margin metric (33-34) reveal a second, less obvious problem that Kubernetes does not report at all: even the single replica that is placed is not safe, because there is no other node to move it to, and the service cannot survive the loss of its only zone.

This is what makes our approach different from two well-known directions. Works that optimize the scoring phase treat placement as a ranking problem and simply assume that at least one node is feasible; our model instead studies feasibility and fault tolerance themselves, which are the questions that come before scoring. General approaches to the formal verification of orchestrators usually build a full model of all the states the system can reach, and therefore run into the problem of state-space explosion. Our model avoids this: it provides simple conditions – (22), (29), (31), (34) – that can be computed directly from the configuration and the current state of the cluster using set operations over the feasible-node sets, without listing all global states. This is a deliberate trade-off. The model looks at a single scheduling moment rather than at the full behavior of the cluster over time. This keeps the checks light enough to run before deployment – for example, inside an admission controller – but it also means that dynamic effects such as preemption or rescheduling are not covered, which is left for future work.

VII. CONCLUSIONS AND FUTURE WORK

A formal model of the filtering phase of the kube-scheduler has been built, oriented toward the analysis of pod placement dependability in advance. In accordance with the objectives set, the following results have been obtained. First, a single formal framework has been introduced (sets of nodes and pods, labels, and indicator predicates) that uniformly describes the heterogeneous filtering mechanisms. Second, the five main filtering predicates – node affinity, taints and tolerations, pod affinity and anti-affinity, and resource sufficiency – have been formalized as boolean functions of a (pod, node) pair. Third, the predicates have been combined into a complex filter, and the result of the filtering phase has been presented as the set of feasible nodes, that is, the scheduler's solution space. Fourth, checkable dependability conditions have been derived from the model: a criterion for the unschedulability of a configuration (the risk of a Pending state) with the localization of the culprit predicates, conditions for the correct spreading of replicas across failure domains, and a placement-margin metric for assessing resilience to node failures. Fifth, the applicability of the model has been illustrated by a worked example of a cluster: it has been shown that the model detects – in advance and from the configuration alone – a hidden loss of fault tolerance that the standard Kubernetes scheduler would detect only at runtime.

The practical significance lies in the ability to verify placement dependability before deployment and to automatically localize the causes of unschedulability. The scientific novelty lies in the model of a formal abstraction of selected filtering predicates to the analysis of availability and fault tolerance of pod placement before deployment: the filtering phase is expressed as set operations over the feasible-node sets, from which checkable conditions (22), (29), (31), and (34) follow without constructing the state space of the cluster.

Further research is directed toward: generalizing the resource-sufficiency predicate to heterogeneous environments with nodes and resources of different types; extending the model to the scoring phase, which will make it possible to analyze not only feasibility but also the quality of a placement; integration with the method of granular analysis of cluster resource usage for predicting the dynamics of the

set of feasible nodes over time; a software implementation of the checks (in particular, as an admission controller) that computes conditions (22), (29), (31), and (34) before deployment; as well as expanding the set of predicates (pod topology spread constraints, preferred rules) and validation on clusters of a larger scale.

References

[1] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," Queue, vol. 14, no. 1, pp. 70–93, 2016. https://doi.org/10.1145/2898442.2898444

[2] "Kubernetes Scheduler." Kubernetes. [Online]. Available at: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

[3] "Assigning Pods to Nodes." Kubernetes. [Online]. Available at: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/.

[4] "Pod Topology Spread Constraints." Kubernetes. [Online]. Available at: https://kubernetes.io/docs/concepts/scheduling-eviction/topology-spread-constraints/.

[5] "Resource Management for Pods and Containers." Kubernetes. [Online]. Available at: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

[6] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud Container Technologies: A State-of-the-Art Review," *IEEE Trans. Cloud Comput.*, vol. 7, no. 3, pp. 677–692, Jul. 2019. https://doi.org/10.1109/tcc.2017.2702586

[7] C. Carrión, "Kubernetes Scheduling: Taxonomy, ongoing issues and challenges," *ACM Comput. Surv.*, Jun. 2022. https://doi.org/10.1145/3539606

[8] R. Buyya *et al.*, "A Manifesto for Future Generation Cloud Computing," *ACM Comput. Surv.*, vol. 51, no. 5, pp. 1–38, Jan. 2019. https://doi.org/10.1145/3241737

[9] R. Regaieg, M. Koubàa, Z. Ales, and T. Aguili, "Multi-objective optimization for VM placement in homogeneous and heterogeneous cloud service provider data centers," *Computing*, vol. 103, no. 6, pp. 1255–1279, Feb. 2021. https://doi.org/10.1007/s00607-021-00915-z

[10] A. Katal, T. Choudhury, and S. Dahiya, "Comparison and Analysis of Container Placement Algorithms in Cloud Data Center," in *Emerging Trends in Expert Applications and Security*, vol. 682, V. Rathore, V. Piuri, R. Babo, and M. Ferreira, Eds. Singapore: Springer, 2023. https://doi.org/10.1007/978-981-99-1946-8_22.

[11] J. Zhang, H. Huang, and X. Wang, "Resource provision algorithms in cloud computing: A survey," *J. Netw. Comput. Appl.*, vol. 64, pp. 23–42, Apr. 2016. https://doi.org/10.1016/j.jnca.2015.12.018.

[12] Z. Jian *et al.*, "DRS: A deep reinforcement learning enhanced Kubernetes scheduler for microservice-based system," *Softw.: Pract. Experience*, Oct. 2023. https://doi.org/10.1002/spe.3284

[13] H. Zhou, H. Y. Chan, S. Y. Zhang, M. E. Lin, and J. Ni, "A Kubernetes custom scheduler based on reinforcement learning for compute-intensive pods," in *CCIOT 2025: 2025 10th Int. Conf. Cloud Comput. Internet Things*, Ho Chi Minh , Vietnam. New York, NY, USA: ACM, 2025, pp. 82–91. https://doi.org/10.1145/3785520.3785532.

[14] V. Sutaria and D. Bhatti, "An Optimized Container Scheduling Algorithm for Kubernetes using Maximization of Resource Utilization," *Int. J. Eng. Trends Technol.*, vol. 74, no. 3, pp. 129–140, Mar. 2026. https://doi.org/10.14445/22315381/ijett-v74i3p110.

[15] J.-Y. Roh, S.-H. Choi, and K.-W. Park, "OOSP: Opportunistic Optimization Scheme for Pod Deployment Enhanced with Multilayered Sensing," *Sensors*, vol. 24, no. 19, p. 6244, Sep. 2024. https://doi.org/10.3390/s24196244.

[16] B. Zhou, S. K. Mondal, Y. Cheng, and H. M. D. Kabir, "On Optimized Scheduling Scheme for Rapid Pod Autoscaling in Kubernetes," *Appl. Sci.*, vol. 16, no. 5, p. 2481, Mar. 2026.: https://doi.org/10.3390/app16052481.

[17] G. Turin, A. Borgarelli, S. Donetti, E. Johnsen, S. Tapia Tarifa, and F. Damiani, "A Formal Model of the Kubernetes Container Framework," in *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles*, vol. 12476, T. Margaria and B. Steffen, Eds. Cham: Springer, 2020. Available: https://doi.org/10.1007/978-3-030-61362-4_32.

[18] G. Turin, A. Borgarelli, S. Donetti, F. Damiani, E. B. Johnsen, and S. L. T. Tarifa, "Predicting resource consumption of Kubernetes container systems using resource models," *J. Syst. Softw.*, p. 111750, May 2023. https://doi.org/10.1016/j.jss.2023.111750.

[19] V. Medel, O. Rana, J. á. Bañares, and U. Arronategui, "Modelling performance & resource management in kubernetes," in *UCC '16: 9th Int. Conf. Utility Cloud Comput.*, Shanghai China. New York, NY, USA: ACM, 2016. https://doi.org/10.1145/2996890.3007869.

[20] V. Medel, R. Tolosana-Calasanz, J. Á. Bañares, U. Arronategui, and O. F. Rana, "Characterising resource management performance in Kubernetes," *Comput. & Elect. Eng.*, vol. 68, pp. 286–297, May 2018. https://doi.org/10.1016/j.compeleceng.2018.03.041.

[21] V. Medel, U. Arronategui, J. Bañares, R. Tolosana, and O. Rana, "Modeling, Characterising and Scheduling Applications in Kubernetes," in *Economics of Grids, Clouds, Systems, and Services*, vol. 11819, K. Djemame, J. Altmann, J. Bañares, O. Agmon Ben-Yehuda, and M. Naldi, Eds. Springer, Cham, 2019. https://doi.org/10.1007/978-3-030-36027-6_26.

[22] V. Medel, C. Tolón, U. Arronategui, R. Tolosana-Calasanz, J. Bañares, and O. Rana, "Client-Side Scheduling Based on Application Characterization on Kubernetes," in *Economics of Grids, Clouds, Systems, and Services*, vol. 10537, C. Pham, J. Altmann, and J. Bañares, Eds. Cham: Springer, 2017. https://doi.org/10.1007/978-3-319-68066-8_13.

[23] M.-N. Tran and Y. Kim, "Optimized resource usage with hybrid auto-scaling system for knative serverless edge computing," *Future Gener. Comput. Syst.*, 2023. https://doi.org/10.1016/j.future.2023.11.010.

[24] R. Ghosh, F. Longo, V. K. Naik, and K. S. Trivedi, "Modeling and performance analysis of large scale IaaS Clouds," *Future Gener. Comput. Syst.*, vol. 29, no. 5, pp. 1216–1234, 2013. https://doi.org/10.1016/j.future.2012.06.005.

[25] R. Ghosh, F. Longo, F. Frattini, S. Russo, and K. S. Trivedi, "Scalable Analytics for IaaS Cloud Availability," *IEEE Trans. Cloud Comput.*, vol. 2, no. 1, pp. 57–70, 2014. https://doi.org/10.1109/tcc.2014.2310737.

[26] K. S. Trivedi, D.-S. Kim, and R. Ghosh, "System availability assessment using stochastic models," *Appl. Stochastic Models Bus. Industry*, vol. 29, no. 2, pp. 94–109, 2012. https://doi.org/10.1002/asmb.951.

[27] B. Liu, X. Chang, Z. Han, K. Trivedi, and R. J. Rodríguez, "Model-based sensitivity analysis of IaaS cloud availability," *Future Gener. Comput. Syst.*, vol. 83, pp. 1–13, 2018. https://doi.org/10.1016/j.future.2017.12.062.

[28] G. Marques, C. Senna, S. Sargento, L. Carvalho, L. Pereira, and R. Matos, "Proactive resource management for cloud of services environments," *Future Gener. Comput. Syst.*, 2023. https://doi.org/10.1016/j.future.2023.08.005.

[29] H. Guo, H. Cao, J. He, X. Liu, and Y. Shi, "POBO: Safe and optimal resource management for cloud microservices," *Perform. Eval.*, vol. 162, p. 102376, 2023. https://doi.org/10.1016/j.peva.2023.102376.

[30] B. Jeong, S. Baek, S. Park, J. Jeon, and Y.-S. Jeong, "Stable and efficient resource management using deep neural network on cloud computing," *Neurocomputing*, vol. 521, pp. 99–112, 2023. https://doi.org/10.1016/j.neucom.2022.11.089.

[31] M. Buriak, "Method of Granular Analysis of Shared Resource Utilization in a Kubernetes Cluster," *Secur. Infocommunication Syst. Internet Things*, vol. 3, no. 2, p. 02001, 2025. https://doi.org/10.31861/sisiot2025.2.02001.

[32] "cluster-capacity." Kubernetes SIGs, GitHub. [Online]. Available at: https://github.com/kubernetes-sigs/cluster-capacity.

[33] "kube-scheduler-simulator." Kubernetes SIGs, GitHub. [Online]. Available at: https://github.com/kubernetes-sigs/kube-scheduler-simulator.

MYKOLA BURIAK, *PhD student at Computer Systems Software Department, Yuriy Fedkovych Chernivtsi National University. Practicing DevOps Engineer with over 7 years of experience. Master of Computer Sciences.*
https://orcid.org/0009-0006-7887-0091

GEORGE VOROBETS, *Candidate of Physical and Mathematical Sciences, Associate Professor, Head of the Department of Computer Systems and Networks, Yuriy Fedkovych Chernivtsi National University. Research interests: Internet of Things technologies and cyber-physical systems; embedded and distributed reconfigurable computer devices, tools and systems; physical principles and technology of laser correction of semiconductor sensor elements and structures for information-measurement and telecommunication systems, automation and control; principles of modeling, analysis and synthesis of bio-inspired and bio-integrated computerized devices and systems.*
https://orcid.org/0000-0001-8125-2047

STEPAN MELNYCHUK, *Doctor of Physical and Mathematical Sciences. Professor of the Department of Computer Systems and Networks, Yuriy Fedkovych Chernivtsi National University. Research interests: theoretical physics, physics of semiconductors and dielectrics, signal processing theory, and the application of IT technologies to solving physical problems.*
https://orcid.org/0009-0001-4325-2342