Published in Contino Engineering

Jaroslav Pantsjoha Follow
Apr 28, 2020 · 9 min read · Listen

Save

Why separate your Kubernetes workload with nodepool segregation and affinity options

Kubernetes is already hard.
We know it. They know it. Steve knows it.
Then why, you'd be right to say — Why!? — is the node labelling is called "**Tainting**" and why-oh-why does the Pod need to "**Tolerate**".

Well, let's figure this one out.

If you are reading this, You're keen, inquisitive but also brave. Major Kudos!

Let's start with the **Why.**

As the average Joe, you `kubectl run nginx – image=nginx – port=80 – expose` , and you would be right to be done with it.

But we don't want to be the average Joe!

In the [any] environment in particular the intent is to get your application Deployment, your `Statefulsets` , your Pods to work as efficiently as possible, leveraging the auto-scaling and self-healing nature of the Kubernetes Cluster and orchestration mechanisms.

This means running application Pods on the correct node with the correct specification — be it with high CPU or high Memory, with or without SSD storage attached. Similarly, this also means that some microservices would be best performant, as a bigger application context piece, if they worked closer together (`affinity`), while in other cases you want to ensure that no such application will be running on the host where such application Pod is already running (`anti-affinity`).

The main intent is to segregate or co-locate workloads as appropriate, and to avoid free-for-all Pods scheduling stampede — poor resource utilisation will be detrimental to the billing budget.

Let’s break this down with visuals and examples.

There is a great set of documents already available on kubenretes.io for your leisure reading. These explain in great detail how to apply labels to nodes and how to apply Toleration tags for your Pods.

I do not wish to repeat the quality content all over again, and will rather visualise and expand upon the DevOps reasoning perspective for such implementation.

TL;DR

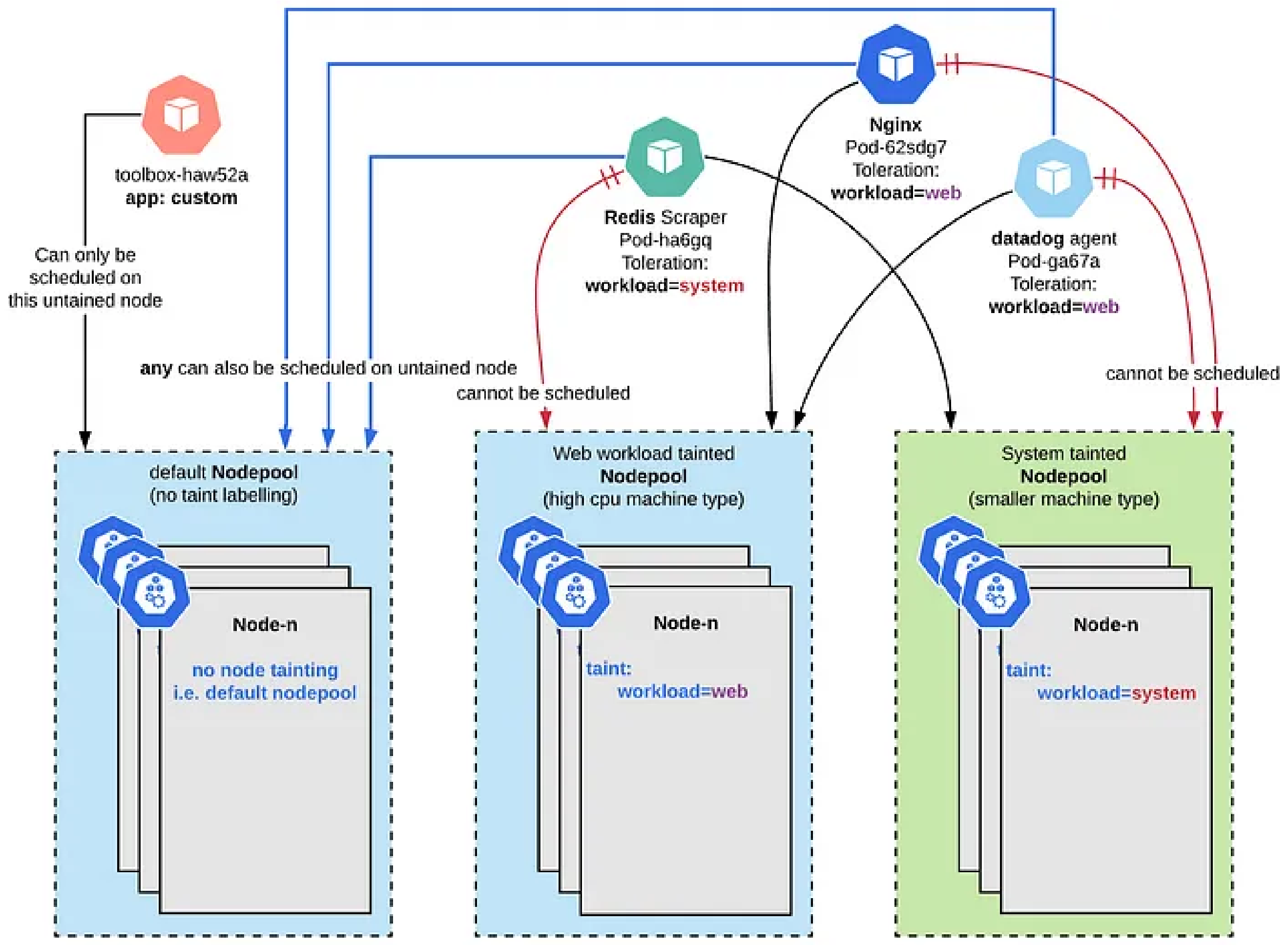
> ***Node Taints** and **Pod Tolerations** are used in combination to ensure the Pods are scheduled to run on the appropriate Node(s).*

It would seem to be that simple.

Wording it properly in line with the documentation thought would describe the taints to allow a node to repel Pods (not matching the taint with toleration). Thus any workloads/Pods which have the toleration configured to match that node taint will be able to be scheduled to run on that node. It's a negative thing.

The takeaway gotcha to be aware of, however, is that **Node**(s) without any taints can have [any] **Pods**, — **Including** Pods with Tolerations labels (if unmatched to tainted nodes), — scheduled to run on them.

As it's all easier with a handy visual, let's review and break it down.



I have taken liberty including a variety of the apps for understandable diversification of the app purposes in this example;

- **Datadog** — logging agent. To be scheduled on **workload=web**
- **Nginx** high CPU requirement web front-end application. To be scheduled on **workload=web**

- **Redis Prometheus custom scraper** (humor me). To be scheduled on **workload=system**
- **Custom Pod** — DevOps busybox **toolbox** image. To be scheduled on default nodepool.

What we have from the Kubernetes Cluster design, are the following nodepools:

- **Default Nodepool** — created on cluster creation, typically there is no node **tainting** in place, to force it to run a particular scheduled application workload(**Pods**). This nodepool will be the default location for all and any workload/**Pod(s)**, which has **no Toleration** or has **Tolerations** which do not match any node Taints.
- **Web Workload — High CPU machine type — Nodepool** — a purpose system specification Node spec, to host a specific set of workload which have such requirements. This could have been a bin-packing exercise to ensure the right number of Pods are scheduled on a node, as well as the relevant workloads/Pods that are scheduled on such machine(s). Such machine types could be expensive, particularly if they were the (High Memory types) but we do not want to schedule any other workloads on such machines. The objective is to run **Nginx application workload only** on such nodes. Also, we wish to run **datadog agents** only on such client-facing production workload hosting nodes in the nodepool as well. Logging agent (pricing is usually per node) would collect the appropriate and relevant metrics only there.
- **System nodepool** — a smaller machine type nodes to support a variety of system tools like **Redis Prometheus scraper** application. These are small Pod(s), and you may not want to have it running anywhere else but rather as in a managed manner. Irrespective of the environment, such Pod(s) will run on such dedicated system nodepool, — which would typically come with the nodepool autoscaler, to need any growing demand. This type of **system nodepool** would typically also be suited to run the **CronJobs**. The latter kubernetes object type can cause resource cost run-away if misconfiguration happens for such regular CronJob. Particularly in cases, where it's a multi-container Pod run, where non-terminating cron job run will result in the subsequent cron job run to spin out a new node, within the nodepool. This is exactly why running a separate nodepool is a good idea; **Cost Damage control**. In this particular nodepool specification,

the machine type is small with further customisation like attached SSD storage to consider.

- By default, the default nodepool will get all the workload/Pod(s) scheduled to it, unless there are toleration matches against taints. The **busybox Toolbox** Pod as per example will be scheduled to run on such **default nodepool**.

The Benefits

The main feature of the Node `Taint` and Kubernetes Workload/Pod `Toleration` spec is to ensure that the right workload gets scheduled on the correct Node specification, with consideration to;

- **Node customisation** to match the workload. You may wish to run high-mem or high-cpu nodes which are more expensive, and thus wish to ensure only high mem/CPU requiring workload will run there. Similarly, perhaps your nodes will have larger SSD storage for your requirements, or nodepool service account (GKE) for your particular RBAC requirements.
- **Cost-damage control**. Some `cronjobs` or workloads can enter failed state, or hang with failing health checks. Things break. Your nodepool autoscaler will kick in automatically to address the demands of your workload. You may wish to keep such a prototype/risk workload separate from another production workload.
- **Maintainability**. You have an option to perform a controlled, staged nodepool upgrade to rollout another specification for your new requirements. Such Node Taints and Workload/Pod Toleration matching will ensure that only a small subset of production workloads would be impacted. The moving of application workload/Pods, the “rebalancing act” can be addressed with further scaling and Node cordon or Taint based evictions.

Pod Spec: Node and Pod Affinity — What is it?

We've talked about how great Node **Taint**-ings work, and where such applications of labels excel when coupled up with **Pod** *spec* featured **Tolerations.**

But it would be highly inappropriate not to bring it all home together and talk about **Pod Spec** *cousins* — `nodeAffinity` and `podAffinity` options. Confused yet? I hope not. That or let's review another visual to make sense of all these terms.

Affinity rules may be used in conjunction with the node taints as you can Taint multiple variations of node pools, or use taints closely to provide certain workloads to be scheduled on certain machine types, considering ephemeral/preemptive machines.

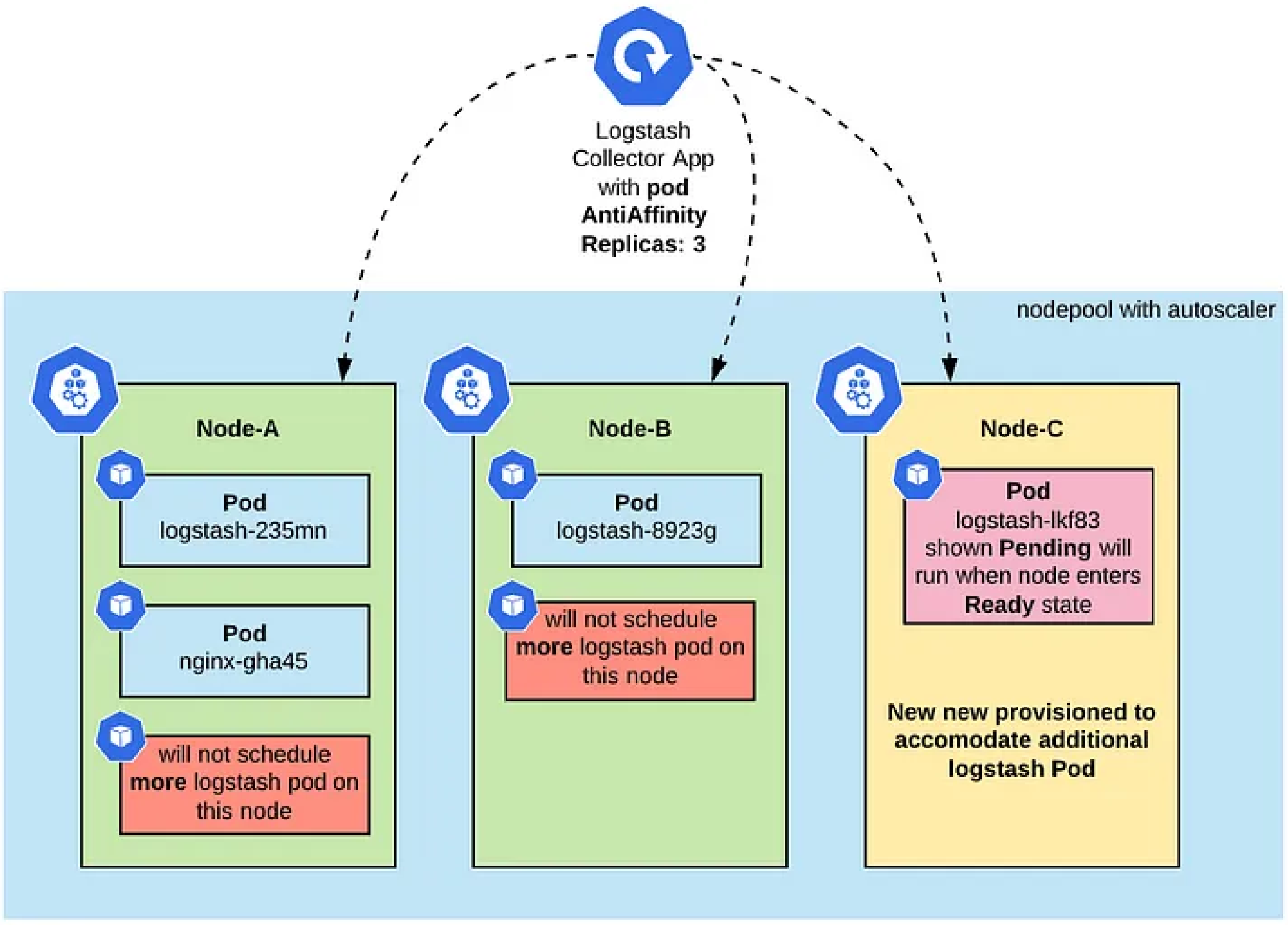
There is a fine documentation resource you can choose to dive right into here.

Pod (workload) [Anti/]Affinity

This is part of the Pod Spec Yaml configuration. This option allows application developers to ensure certain types of distinct workloads get to be scheduled for runtime on the same host or distinct hosts. This could be a consideration for the disk IOPS for certain similar types of production workload — where fanning these Pods out across a number of nodes would reduce contention. Another consideration is to ensure application resiliency in case of node or zonal failure (for even sets)

There are two natural affinity options:

- **Affinity** — Co-locate the label matching workloads on the same node(s) for availability or performance requirement. Consider network performance for workload communication to take place on the same host. Or,
- **Anti-Affinity** — Ensure certain workload types do not get mixed on the same host due to (IOPS/Memory/CPU) performance requirement. Ensure resource isolation — similar to the dedicated nodepool. Ensure workload gets to be spread out over larger availability zones rather than getting scheduled for runtime on the same host, in the same zone. Remember that opting for the larger scale of your deployment/statefulset will not necessarily guarantee that the actual workload is spread over multiple availability zones.


Logstash Collector App with pod AntiAffinity Replicas: 3
nodepool with autoscaler
Node-A
Pod logstash-235mn
Pod nginx-gha45
will not schedule more logstash pod on this node
Node-B
Pod logstash-8923g
will not schedule more logstash pod on this node
Node-C
Pod logstash-lkf83 shown Pending will run when node enters Ready state
New new provisioned to accomodate additional logstash Pod

With another example of the Redis cache deployment, for resiliency, you may not wish the Pods to be scheduled on the same node.

```
spec:
  affinity:
```

```
podAntiAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
  - labelSelector:
      matchExpressions:
      - key: app
        operator: In
        values:
        - store
    topologyKey: "kubernetes.io/hostname"
```

Node and Pod [Anti/]Affinity Rules

This is configured within the Pod specification YAML file itself. It specifies which Node will be effectively selected for the scheduling process.

There are **two main** rules to be aware of;

- **requiredDuringSchedulingIgnoredDuringExecution — Hard Affinity**
  "Only schedule this workload to run in this availability zone", for example. This is a **hard** rule and if there are no such machines available, your Pods **will simply not be scheduled** and able to run. This could be a compliance or performance requirement.

- **preferredDuringSchedulingIgnoredDuringExecution — Soft Affinity**
  "Try to run this workload in this availability zone", — as **soft** rule, and if such preference is not met, the workload will be scheduled to run anywhere else. This is a handy cost optimization exercise.

Just to ensure we cover the key distinct differences between **Pod** and **Node affinity** options again;

— With **Node Affinity**, with Pod spec config, you're able to just select the same node again, and

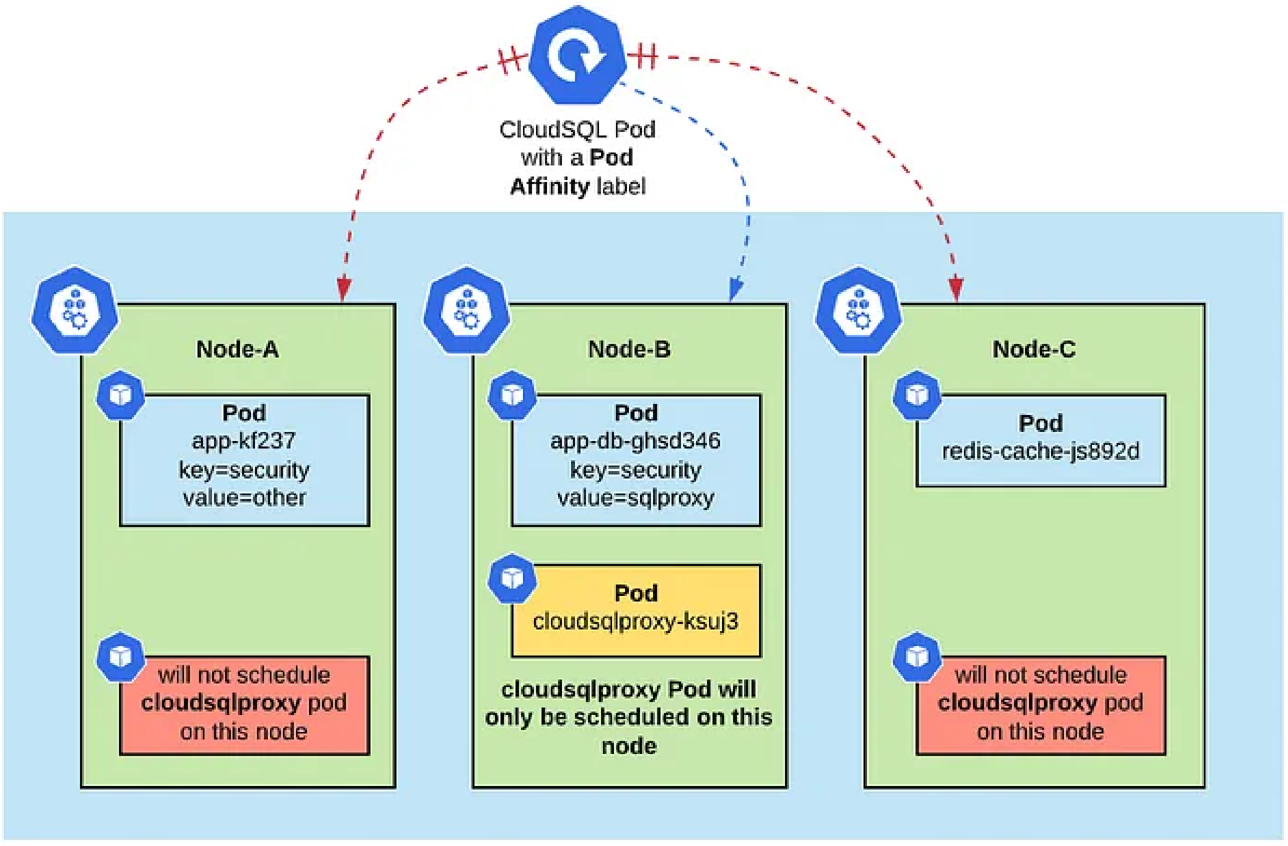
— While with **Pod Affinity**, you are able to select the same **Node** hosting the specific (label selected) Pod.

Let's review the Pod Affinity configuration, with another example below. Here you specify the podAffinity to schedule the Pod to be run on another node where there is another Pod running which **must** (requiredDuringSchedulingIgnoredDuringExecution) **match** execution pre-requisite

— the existence of correctly matching Label on a specific Pod, otherwise it will not be scheduled.

```
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: security
            operator: In
            values:
            - sqlproxy
        topologyKey: failure-domain.beta.kubernetes.io/zone
```

This adds a degree of workload dependency and cost optimisation for your workload, whereby you refrain from consuming resources unless there is an underlying dependency being met — i.e. that other **Pod** with security label and value sqlproxy exist [on any node] beforehand. Given the obvious complexity, this is a day n+1 job. The onus is to keep all labels updated, consistent and workload(s) organised first to make these affinity rules work from get-go. Failure to do so will result in hours troubleshooting the dependencies and the 'stuck' Pods. Been there, done that, got many t-shirts.


CloudSQL Pod with a Pod Affinity label
Node-A
Pod app-kf237 key=security value=other
will not schedule cloudsqlproxy pod on this node
Node-B
Pod app-db-ghsd346 key=security value=sqlproxy
Pod cloudsqlproxy-ksuj3
cloudsqlproxy Pod will only be scheduled on this node
Node-C
Pod redis-cache-js892d
will not schedule cloudsqlproxy pod on this node

In Summary

There. How was that for a rundown.

If you’re still here and managed to sit through this in one go, bravo. It’s a tough one. There are many options available and the full deep dive is available reading the official Kubernetes Docs

In yet-another-short summary**: Kubernetes is hard, but don’t let that put you off.** There are a lot of options, and it is easy to get confused.

The best way to think about this is to start considering your own resilient containerised application. What would be your application requirements?

- **Dedicated isolated node-pool**? Then Tolerations and Taints are a way to go. Relatively easy to get started with since default nodepool without taints will act as catch-all.
- **Smart logic for your app to be forcefully distributed among** all **availability zones**? Then Node anti-affinity is what you are after
- If you want to **co-locate some apps for performance or avoid placing similar resource-churning apps on the same host**, then Pod affinity and anti-affinity is a way to go.

Right, I think that does it.

Hope you enjoyed this. Enjoying the read?

Like, Clap and Share this post along!

Join the conversation. See you in **#Kubernetes-users** Kubernetes slack group

PS There are quite a number of exciting Kubernetes projects taking place at **Contino.** If you are looking to work on the latest-greatest infrastructure stack or looking for a challenge, — Get in touch! We’re hiring, looking for bright minds at every level. At **Contino**, we pride ourselves on delivering the best practices cloud transformation projects, for medium-sized businesses to large enterprises.

Node Taint Pod Toleration Kubernetes Affinity Nodepool

About Help Terms Privacy

Open in app ↗ Sign up Sign In