8Kubernetes系列（十）如何加强 deployment 安全

Kubernetes系列（十）如何加强 deployment 安全

红帽最近的一项调查发现，超过一半的 Kubernetes 环境配置是不太规范的。更糟糕的是，大约90% 的受访者去年至少发生过一次安全事件，导致了 kubernetes 恶意软件的流行。在本文中，我们将介绍三个工具来验证和保护你的 Deployments。

使用 Kubernetes 持续部署

Kubernetes 最大的卖点在于它的声明式，也就是通过编写一个清单（manifest）文件来描述一个部署期望的状态，然后让平台解决剩余的问题。但是实际情况是这些清单很容易导致一些问题的出现。

Kubernets 的 CD的落地，只有当我们在每个阶段都进行很好的测试才能进行，比如：

1. 首先，在构建阶段检查验证你的代码。

2. 然后，构建容器镜像并且检查其结构。

3. 在部署之前检查验证清单文件（manifest）。

4. 部署之后，使用类似金丝雀或者蓝绿部署的方式在生产环境进行测试。

校验 Kubernetes 清单（manifests）

为了实现安全部署，你完全没有必要成为 Kubernetes 的权威专家，通过一些测试工具可以辅助我们来处理安全问题。不过，你必须对 Kubernetes 的工作方式有基本的认识。如果这是你第一次接触 kubernetes、pod或者容器的世界。请关注本服务号，发送 `cicd-book` ，获取相关介绍材料。

首先，你需要检查清单格式是正确的，换句话说，必须遵循 Kubernetes OpenAPI规范（ `https://github.com/kubernetes/kubernetes/tree/master/api/openapi-spec` ）。接下来我们将介绍两个工具：kubeval 和 kubeconform。

Kubeval

Kubeval(`https://www.kubeval.com/`)是一个命令行工具，同时也是一个用于验证 Kubernetes 清单的 GO 库。支持 Linux、MacOS、Docker 以及 Windows。

Kubeval 虽然只做一件事，但是它做的很好。安装之后，我们可以检查 Yaml 或者 JSON 格式的清单文件。强烈建议使用 `--strict` 参数，在发现未知属性配置时标志测试失败。

```
$ kubeval --strict deployment.yml
```

Kubeconform

受 Kubeval 启发，Kubeconform 也做了同样的事情，但是它更专注于速度以及扩展性。与Kubeval 不同，Kubeconform 可以检查自定义的资源类型（CRD）。

Kubeconform 自动从远程存储库下载 CRD 定义并进行验证测试。它同时会自动更新 Kubernetes 的最新规范。

KubeConform 支持 Docker、MacOS、Linux 以及 windows。安装之后，我们使用以下命令运行测试：

```
$ kubeconform deployment.yml
```

运行后，任何错误都将被报告。如果想要更详细的输出，可以添加 `--summary` 选项。

使用 Kube-Score 来保护 Deployments

如果没有 Kubernetes 的专家来检查每一次部署，那么一份包含合理实践的清单是最佳选择。Kube-Score（`https://github.com/zegl/kube-score`）会扫描您的部署清单，提供建议，并在我们陷入麻烦之前抛出错误。你可以在这里（`https://github.com/zegl/kube-score/blob/master/README_CHECKS.md`）看到完整的检查列表。

Kube-Score 可以在 Windows、Linux、macOS和Docker上运行，你可以在线（`https://kube-score.com/`）试用。

通过运行 `kube-score score deployment.yml` 得到下面的结果：

```
apps/v1/Deployment semaphore-demo-ruby-kubernetes
    [CRITICAL] Container Resources
        · semaphore-demo-ruby-kubernetes -> CPU limit is not set
            Resource limits are recommended to avoid resource DDOS. Set
            resources.limits.cpu
        · semaphore-demo-ruby-kubernetes -> Memory limit is not set
            Resource limits are recommended to avoid resource DDOS. Set
            resources.limits.memory
        · semaphore-demo-ruby-kubernetes -> CPU request is not set
            Resource requests are recommended to make sure that the application
            can start and run without crashing. Set resources.requests.cpu
        · semaphore-demo-ruby-kubernetes -> Memory request is not set
            Resource requests are recommended to make sure that the application
            can start and run without crashing. Set resources.requests.memory
    [CRITICAL] Container Image Pull Policy
        · semaphore-demo-ruby-kubernetes -> ImagePullPolicy is not set to Always
            It's recommended to always set the ImagePullPolicy to Always, to
            make sure that the imagePullSecrets are always correct, and to
            always get the image you want.
    [CRITICAL] Pod NetworkPolicy
        · The pod does not have a matching NetworkPolicy
            Create a NetworkPolicy that targets this pod to control who/what
            can communicate with this pod. Note, this feature needs to be
            supported by the CNI implementation used in the Kubernetes cluster
            to have an effect.
    [CRITICAL] Pod Probes
        · Container is missing a readinessProbe
            A readinessProbe should be used to indicate when the service is
            ready to receive traffic. Without it, the Pod is risking to receive
            traffic before it has booted. It's also used during rollouts, and
            can prevent downtime if a new version of the application is failing.
            More information: https://github.com/zegl/kube-score/blob/master/README_PROBES.md
    [CRITICAL] Container Security Context
        · semaphore-demo-ruby-kubernetes -> Container has no configured security context
            Set securityContext to run the container in a more secure context.
v1/Service semaphore-demo-ruby-kubernetes-lb
```

上面的评估结果不容乐观，部署有五个关键错误，是时候来修复这些问题了。

什么是镜像拉取策略?

当我们第一次在Kubernetes中部署一些东西时，kubelet 会从相应的 registry 中拉取镜像。镜像将保存在节点的缓存中以供重复使用。只要管理好镜像 Tag，缓存镜像就会一直有效。我的意思是，如果我们不重复使用同一个 tag（也不要一直使用 `latest`），那么便是可靠的。否则，将发生意想不到的事情，因为我们永远无法确定节点运行的是哪个镜像:缓存中的镜像还是 registry 中的镜像。

`ImagePullPolicy` 的值定义了镜像缓存的工作方式。默认值是 `IfNotPresent` 意味着如果本地没有缓存那么就从 registry 拉取镜像。建议将它的值设置为 `Always`，这样子缓存就失效了，始终拉取最新的镜像。

```
spec:
  template:
    spec:
      containers:
      - name: semaphore-demo-ruby-kubernetes

        # ... add this line to never cache the image
        imagePullPolicy: Always
```

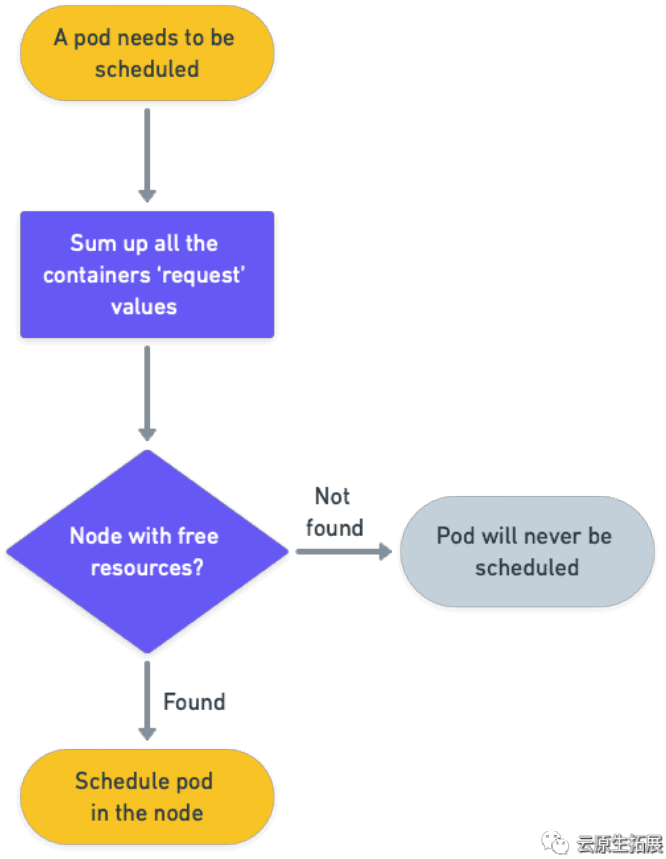
设置 CPU 以及 内存 Memory limits

Kubernetes中的容器没有任何限制。狂野而自由的它们会占用尽可能多的CPU和内存。让一个pod运行失控的容器占用节点的所有内存是没有意义的。为了控制这些，我们应该为部署中的所有POD 指定限制。

当然，limits 设置过低反而比不设限更糟糕。所以监控也是至关重要的。除非你真的需要没有限制的 pod，那你也最后将这些pod 控制在固定的节点下面，以保持集群的稳定性。

继续修复我们的 deployment。我们可以通过两种方式：requests 和 limits 来配置资源配额。

request 配置了正常运行一个容器所需的最小CPU以及内存。Kubernetes 调度器将根据此信息将 POD 调度到足够空闲资源的节点。

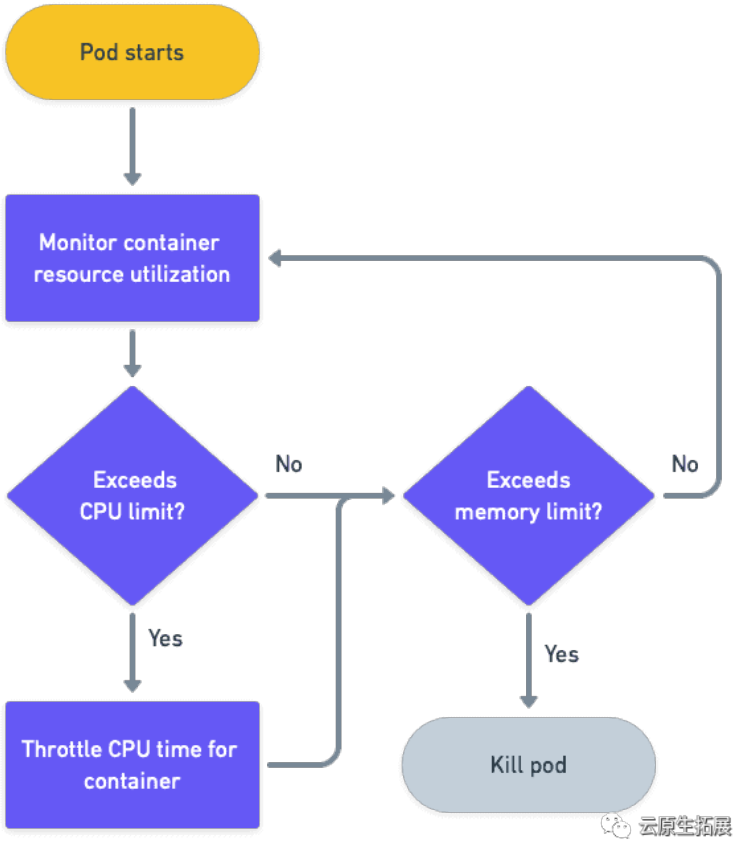


Limit 配置了一个容器允许使用的最大内存和CPU 梳理。容器允许使用超过 request 配置的值，但是不允许超过 limits 设置的值。

requests 和 limits 都可以指定如下选项：

- **CPU**：值为1时，等同于裸机上有一个超线程处理器，或虚拟机中有一个vCPU核。例如，值0.25是一个核心的四分之一，也可以写成 250m (millicpus 毫核)。容器使用的CPU时间不能超过其允许的份额，否则将面临节流的风险。容器和pod永远不会因为超出处理器配额而被杀死。

• **Memory**：定义为字节(1Mb = 10^6字节)或mebibytes (1Mi = 2^20字节)。当容器使用超过允许的内存时，它将以内存不足(OOM)错误终止，从而杀死pod。



需要注意的是，这些值是针对每个容器设置的。当POD由一个以上的容器组成时，它们是汇总的。

所以，现在我们知道 requests 和 limits 是如何工作的。像那种简单的应用程序只需要很少的资源。复杂的系统需要更多;到底要增加多少，最好通过就地监测和一些实验来回答。

```
spec:
  template:
    spec:
      containers:
      - name: semaphore-demo-ruby-kubernetes

        # ... add resource requests and limits
        resources:
          requests:
            cpu: "100m"
            memory: "64Mi"
          limits:
            cpu: "200m"
            memory: "128Mi"
```

使用 Liveness 和 Readiness 探测

一旦某个容器运行，Kubernetes 如何追踪它的健康状态? 它准备好接受用户连接了吗? 这些问题的答案涉及到使用探针（`https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/`）。探针有两种工作模式:

• **command**：在容器内运行一个命令，并时刻检查是否存在退出码.

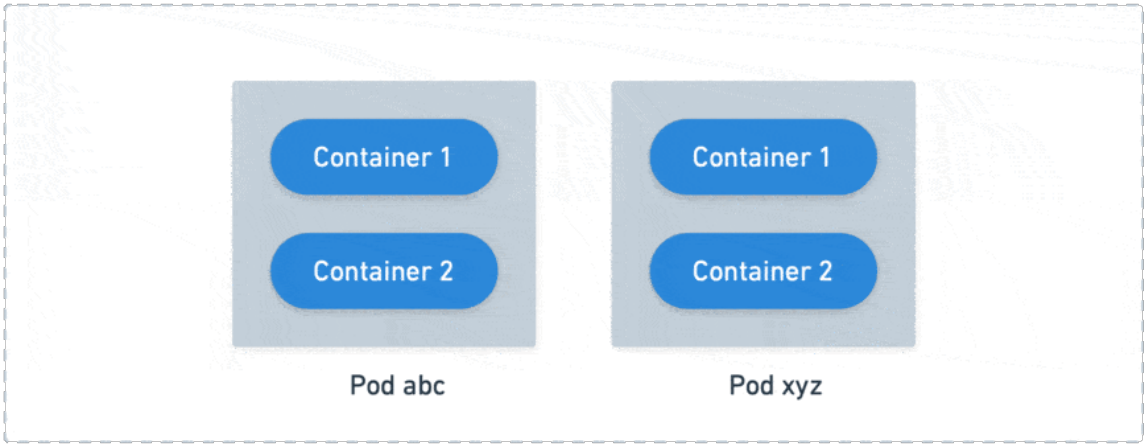
• **network**：ping 端口或者执行 http 请求，并检查结果.

下面的例子，通过检查进程状态来判断:

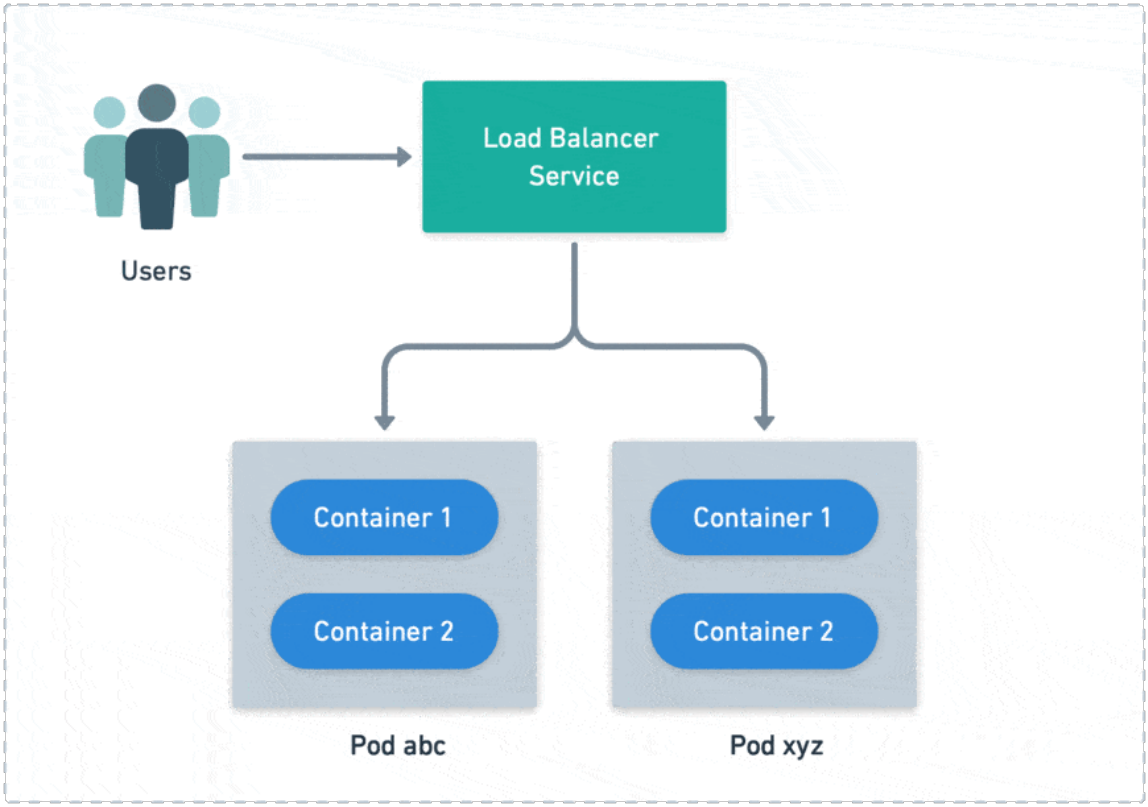
```yaml
spec:
  template:
    spec:
      containers:
      - name: semaphore-demo-ruby-kubernetes

        # ... add liveness probe
        livenessProbe:
          exec:
            command:
              - pgrep
              - -f
              - puma
          initialDelaySeconds: 5
          periodSeconds: 5
```

上面的探针被称为存活探针（liveness probe）。如果退出码为零，则认为应用程序是活动的。失败的探测将重新启动容器。



还有另一种类型的探测叫做可读性探测（readiness probe）。 它检查 POD 是否准备好接受请求连接。 失败的探测不会自动重启pod，但它会断开与负载均衡器的连接，阻止用户访问它。 readiness 探测是为了应对暂时的问题。



我们在容器级别定义探针，但Kubernetes处理的是pods而不是容器。 这意味着它的所有容器探针必须一起传递结果，以使pod被认为是活的和准备好的。

让我们添加一个就绪探针（readiness probe），它尝试在端口4567上发出GET请求。只要返回代码在200到399之间，探测就会通过。

```
spec:
  template:
    spec:
      containers:
      - name: semaphore-demo-ruby-kubernetes

        # ... and a readiness probe
        readinessProbe:
          httpGet:
            port: 4567
            path: /
          initialDelaySeconds: 5
          periodSeconds: 20
```

通过 pod 网络策略控制访问权限

除非另有说明，Kubernetes的pods被允许从任何地方发送和接收数据包。虽然在没有配置某种端点的情况下，pods确实不能接收来自外部的通信，但集群内的所有pods都允许自由通信。我们使用pod 网络策略(`https://kubernetes.io/docs/concepts/services-networking/network-policies/`)锁定他们的访问权限。

在集群中，我们可以定义跨pods和跨namespace策略来隔离pods。从外部，我们使用基于ip的策略锁定外部访问。

下面的例子，是一个允许所有范围范围的策略:

```
# pod network policies are a different resource
# add these lines at the end of deployment.yml

---

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all
spec:
  podSelector:
    matchLabels:
      app: semaphore-demo-ruby-kubernetes
  ingress:
  - {}
  egress:
  - {}
  policyTypes:
  - Ingress
  - Egress
```

不幸的是，基于ip的策略有一些缺点。首先，它的实现依赖于集群上启用的网络插件。在许多情况下，策略可能会被完全忽略，因为不是每个云都以相同的方式支持它们。

使用 security context 来保护容器

虽然网络策略通过控制通信来保护pods，但安全上下文（`https://kubernetes.io/docs/tasks/configure-pod-container/security-context/`）定义了pods和容器在运行时拥有的特权。安全上下文允许我们运行具有更多(或更少)功能的pods，授予或拒绝访问内部文件，或控制允许哪些系统调用。

这是一个需要对Linux和容器内部有相当深入的理解才能很好地使用的主题。我们将创建一个基本的安全上下文，将文件系统标记为只读，并使用比较大的用户ID运行容器，这样应该能通过 Kube-Score 的检查了。

```
spec:
  template:
    spec:
      containers:
      - name: semaphore-demo-ruby-kubernetes

        # add a security context for the container
        securityContext:
          runAsUser: 10001
          runAsGroup: 10001
          readOnlyRootFilesystem: true
```

再次运行 Kube-Score 以确保不再出现错误。您可以使用 `--enable-optional-test TEST_ID` 启用可选测试，或者使用 `--ignore-test TEST_ID` 禁用强制测试。

完成后，将文件提交到存储库，这样就可以进行下面的部分:使用Semaphore CI/CD自动化测试。

```
apps/v1/Deployment semaphore-demo-ruby-kubernetes                    
networking.k8s.io/v1/NetworkPolicy allow-all                         
v1/Service semaphore-demo-ruby-kubernetes-lb                         
```

作为参考，清单的最终版本是这样的:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: semaphore-demo-ruby-kubernetes
spec:
  replicas: 1
  selector:
```

```
    matchLabels:
      app: semaphore-demo-ruby-kubernetes
  template:
    metadata:
      labels:
        app: semaphore-demo-ruby-kubernetes
    spec:
      imagePullSecrets:
      - name: dockerhub
      containers:
      - name: semaphore-demo-ruby-kubernetes
        image: $DOCKER_USERNAME/semaphore-demo-ruby-kubernetes:$SEMAPHORE_WORKFLOW_ID

        # don't cache images
        imagePullPolicy: Always

        # resource quotas
        resources:
          requests:
            cpu: "100m"
            memory: "64Mi"
          limits:
            cpu: "200m"
            memory: "128Mi"

        # liveness probe
        livenessProbe:
          exec:
            command:
              - pgrep
              - -f
              - puma
          initialDelaySeconds: 5
          periodSeconds: 5

        # readiness probe
        readinessProbe:
          httpGet:
            port: 4567
            path: /
          initialDelaySeconds: 5
          periodSeconds: 20

        # security context for the container
        securityContext:
          runAsUser: 10001
          runAsGroup: 10001
          readOnlyRootFilesystem: true

---

apiVersion: v1
kind: Service
metadata:
  name: semaphore-demo-ruby-kubernetes-lb
spec:
  selector:
    app: semaphore-demo-ruby-kubernetes
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: 4567

---

# pod network policy
```

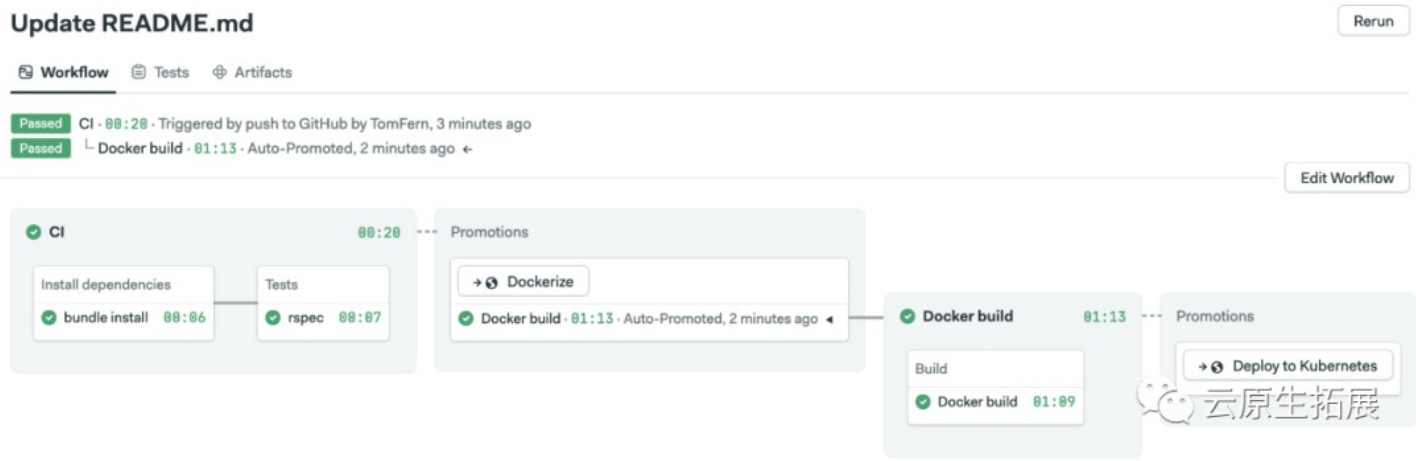
```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all
spec:
  podSelector:
    matchLabels:
      app: semaphore-demo-ruby-kubernetes
  ingress:
  - {}
  egress:
  - {}
  policyTypes:
  - Ingress
  - Egress
```

通过 CI/CD 集成清单测试

到目前为止我们的学习都是为现在做准备。我们现在的目标是将这些工具合并到CI/CD管道中。

您可以将以下步骤与Docker容器结构测试中配置的管道相结合（`https://semaphoreci.com/blog/structure-testing-for-docker-containers`），该管道展示了如何在部署之前测试容器。我假设您对Semaphore的工作原理有一定的了解。如果不是这样，请查看我们的初学者指南（`https://docs.semaphoreci.com/guided-tour/getting-started/`）。

我们现在要做的，就是构建Docker镜像、测试它并将其部署到Kubernetes的管道。演示代码可以在这里找到:



清单验证应该在部署之前进行。 第一条管道进行持续集成，第二条管道构建 Docker镜像。 部署发生在最后一个管道中，因此我们将在部署到Kubernetes之前添加清单验证。

打开 worfflow 编辑器，并将你自己置于部署管道中。

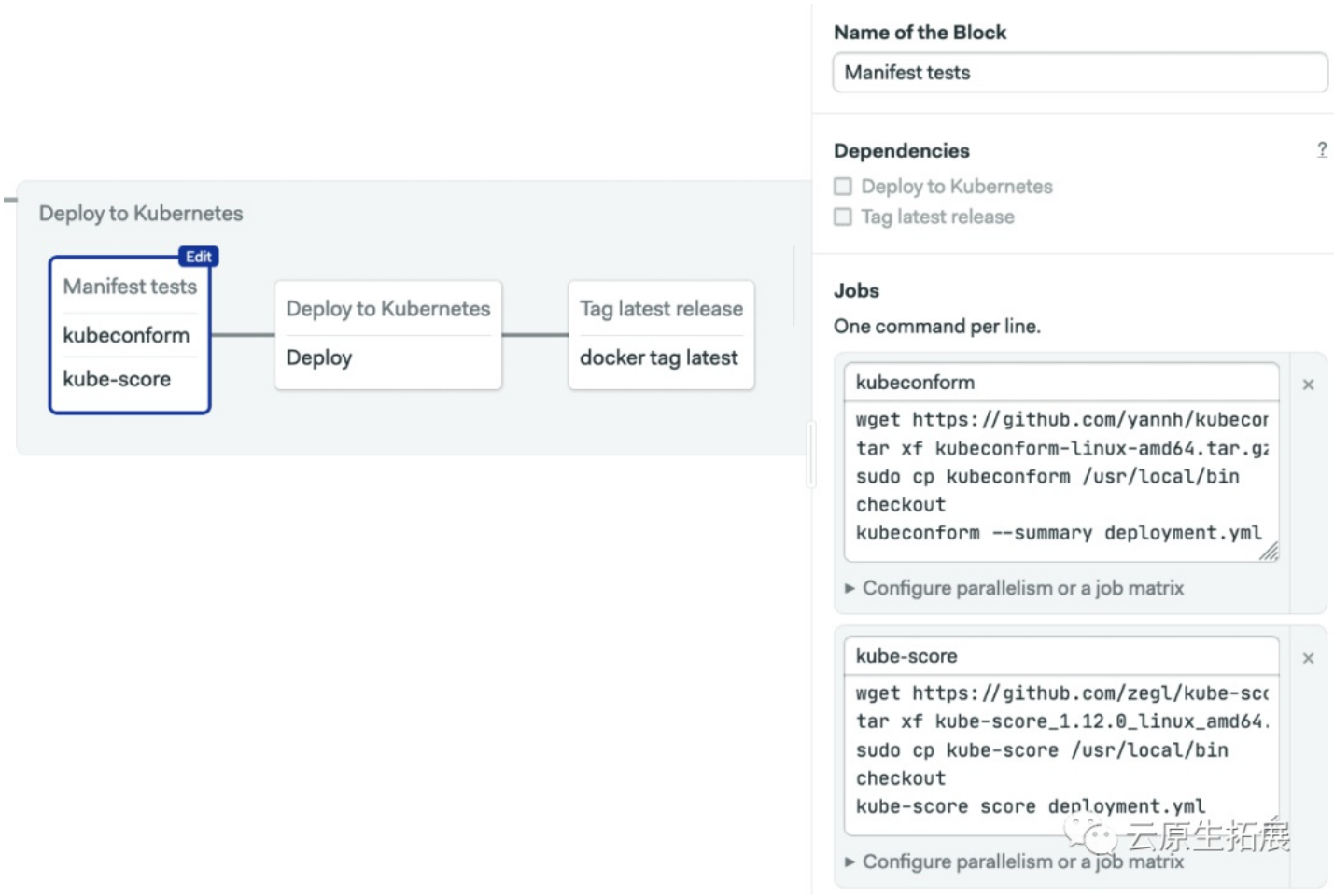
创建一个新的区块并删除它的所有依赖项。我们将配置两个 job。第一个安装Kubeconform，通过clone 存储库，然后运行测试。

```
wget https://github.com/yannh/kubeconform/releases/download/v0.4.12/kubeconform-linux-amd64.tar.gz
tar xf kubeconform-linux-amd64.tar.gz
sudo cp kubeconform /usr/local/bin
checkout
kubeconform --summary deployment.yml
```

第二个通过 Kube-Score 做同样的事情:

```
wget https://github.com/zegl/kube-score/releases/download/v1.12.0/kube-score_1.12.0_linux_amd64.tar.gz
tar xf kube-score_1.12.0_linux_amd64.tar.gz
sudo cp kube-score /usr/local/bin
checkout
kube-score score deployment.yml
```

现在，如果其中一个作业失败，我们希望停止部署。单击“Deploy to Kubernetes”块，并添加新的测试块作为依赖项。



在试用管道之前，请检查Kubernetes部署作业。 关于它如何工作的完整细节可以在我们的Kubernetes持续部署指南中(`https://semaphoreci.com/blog/guide-continuous-deployment-kubernetes`)找到。

单击Run the workflow和Start。等到CI和Docker构建管道结束后再尝试部署。

单击Deploy to Kubernetes以启动管道。

清单校验可以阻止不安全的部署。

结论

能够在一周中的任何一天安全部署是一件非常强大的事情。但要自信地完成它，需要在每个阶段都进行良好的测试。

欢迎关注我的公众号“ **云原生拓展** ”，原创技术文章第一时间推送。