بخش دوم

کسرا نوربخش 810100230

هومان مشیری امین 810100210

# مقدمه

در این بخش از تمرین به کار با k8s پرداختیم.

**خب دیگه دست به کیبورد شیم!**

در ابتدا با استفاده از دو دستور curl، kind و kubectl را بر روی WSL مان نصب کردیم. سپس یک فایل yml ساده با توضیحات داده شده نوشتیم(اسم فایل کانفیگ kind-ha-cluster.yaml می باشد):

kind: Cluster

apiVersion: kind.x-k8s.io/v1alpha4

nodes:

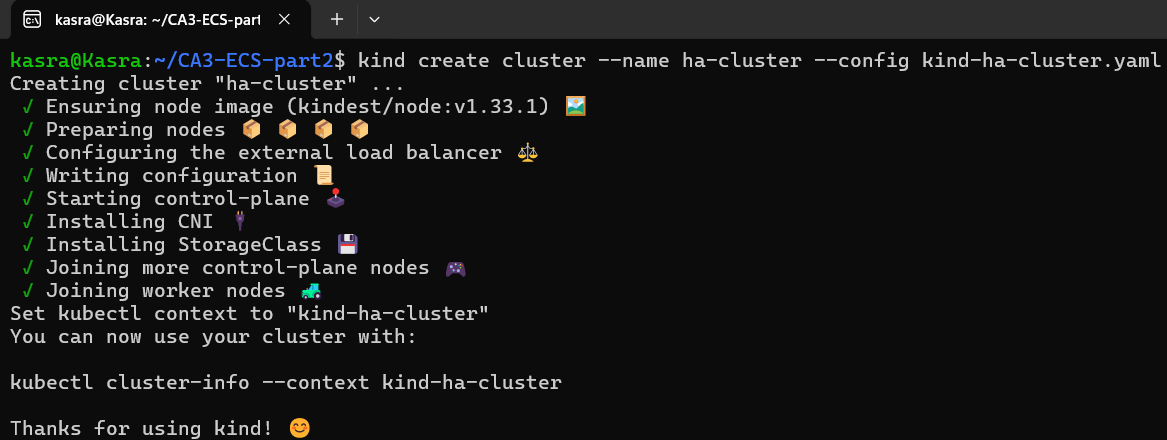
- role: control-plane

- role: control-plane

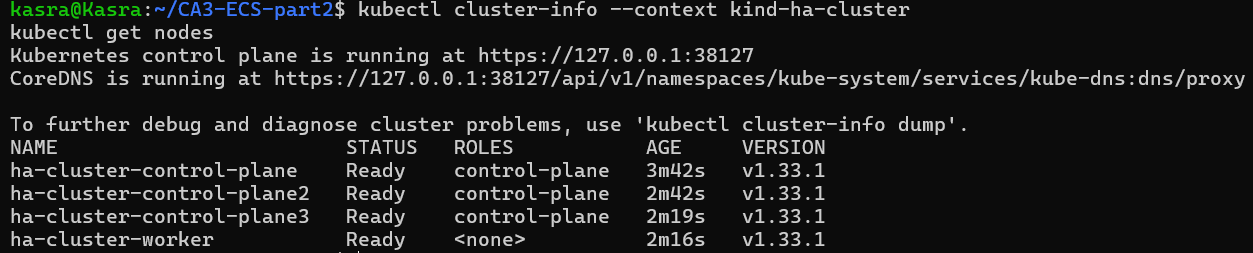
- role: control-plane

- role: worker

سپس با استفاده از دستور زیر، کلاستر را ساختیم و deploy کردیم:



اطمینان حاصل کردیم که ساخته شده باشند:



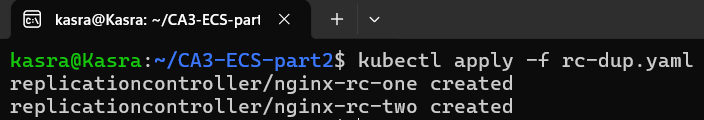
می دانیم که در هر سیستم توزیع شده ای ما نیاز به اجماع رسیدن و الگوریتم های Quorumداریم و برای این که بتوانیم مابین master ها در شرایط رخداد یک حادثه، ارور یا هر جای دیگری به نتیجه گیری برسیم، تعداد باید فرد باشد. Best practice برای تعداد نود های مختلف در کلاستر های بسیار بزرگ: مستر نود ها: 3 یا 5. به دلیل این که با داشتن 3 نود مستر تقریباً تمامی نیاز های سازمان تامین می شود اما اگر دسترس پذیری بیشتر بخواهیم به سراغ 5 می رویم. اگر بیش از 5 داشته باشیم موجب تاخیر و پیچیدگی می شود. ورکر نود ها: بسته به workload ای که داریم مقیاس می شوند و می توانند بین 10 تا بیش از 1000 نود هم باشند. ضمناً برای نگه داری وضعیت کلاستر از 3 الی 5 etcd هم استفاده می شود.

**آپدیت ما نباید مشتری رو اذیت کنه!**

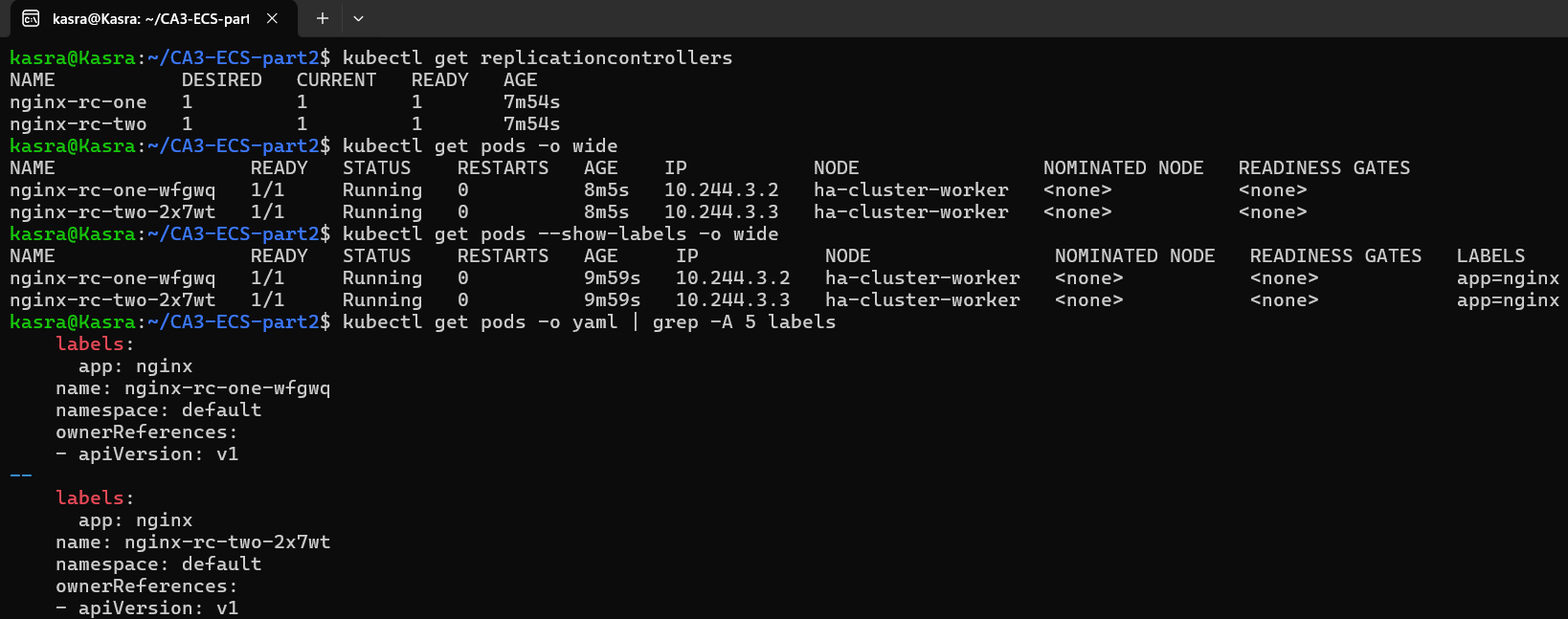
در واقع هر دو کنترل کننده هایی هستند که اطمینان حاصل می کنند که تعداد مشخصی pod در هر لحظه در حال اجراست. ReplicationController مدل قدیمی تر است که ساده تر هم می باشد و از یک single label selector برای شناسایی pod ها استفاده می کند.(بنابراین نمی تواند به صورت پیچیده میان آن ها انتخاب کند) اما ReplicaSet ها پیشرفته تر اند و از set-based selector ها استفاده می کنند که عموماً توسط کنترلر های سطح بالاتر مانند deployment ها مدیریت می شوند و در انتخاب pod ها انعطاف پذیر تر اند. ما در deployment های جدید نباید از ReplicationController استفاده کنیم.

با توجه به توضیحات داده شده، هر ReplicationController تلاش می کند که به صورت مستقل، 1 پاد را maintain کند که بتواند با selector اش آن را انتخاب کند. چون تمامی تنظیماتشان یکسان می باشد، هر دو تلاش می کنند که یک pod مشخص را انتخاب کنند، پس اولین کنترلر آن را می سازد و دومین کنترلر هم pod ای را می سازد اما مساله در این است که به صورت اتوماتیک هنگام ساختن pod ها یک pod-template-hash هم پشت صحنه اضافه می شود که از هم قابل تمییز دادن هستند.(البته اول من از hash پشت صحنه خبر نداشتم و انتظار داشتم 1 pod ساخته بشه اما 2 تا شد)

ابتدا manifest داده شده را ذخیره کردیم و با استفاده از دستور زیر آن را apply کردیم:



حال می توانیم خروجی را ببینیم: (باید دقت کرد که گرچه label ها یکسان اند اما در انتها می بینیم که hash به آن ها اضافه شده است و name ها متفاوت است)



اگر از ReplicationController داریم استفاده می کنیم چاره این است که در manifest، در قسمت image نام را از 20 به 21 تغییر بدهیم یعنی: nginx:1.21 و دوباره دستور apply رو بزنیم و به صورت دستی pod های قدیمی را پاک بکنیم که این روش بسیار بدی می باشد چون همراه با down time می باشد و به صورت دستی باید اعمال شود. اگر به سراغ ReplicaSet ها برویم، می توانیم باز دستی خط مربوط به image را تغییر دهیم و مقیاس به پایین و بالا به ترتیب داشته باشیم که down time کاهش می یابد اما باز هم دستی داریم عمل می کنیم که بد است. بهترین روش استفاده از deployment می باشد که پیش تر گفتیم ReplicaSet را هم کنترل می کند به این صورت که manifest دپلویمنت را ادیت می کنیم و apply می زنیم. در این حالت k8s به صورت اتومات rolling update را اجرا می کند به این معنا که به آرامی پاد های جدید را مقیاس می کند و و هر وقت ready شدند، پاد های قدیمی را به 0 مقیاس می کند. با این روش هم down time نداریم، هم از منابع به خوبی استفاده کرده ایم و هم rollback می توانیم بکنیم. دستور apply: kubectl apply -f deployment.yaml می باشد و فایل yaml شامل:

spec:

template:

spec:

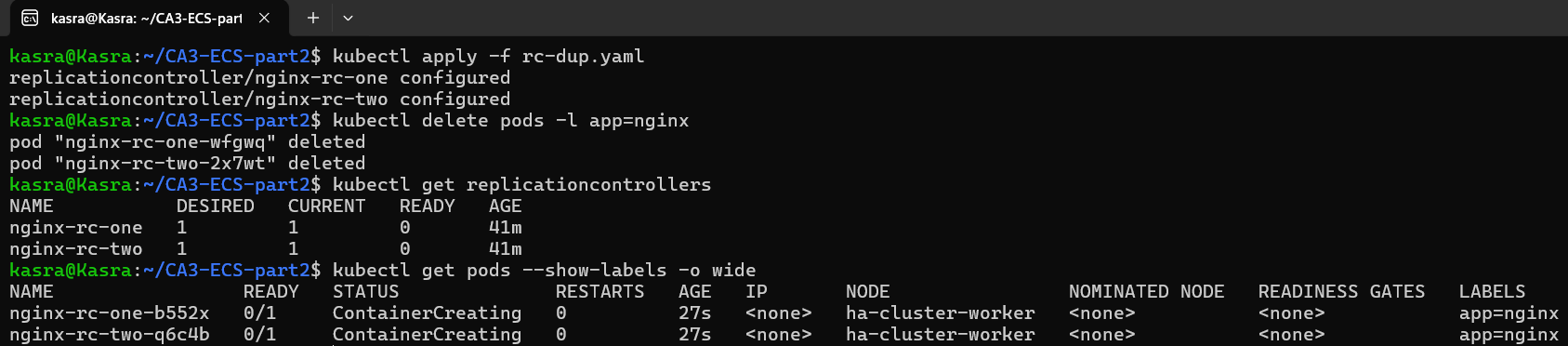
containers:

- name: nginx

image: nginx:1.21 # Update version

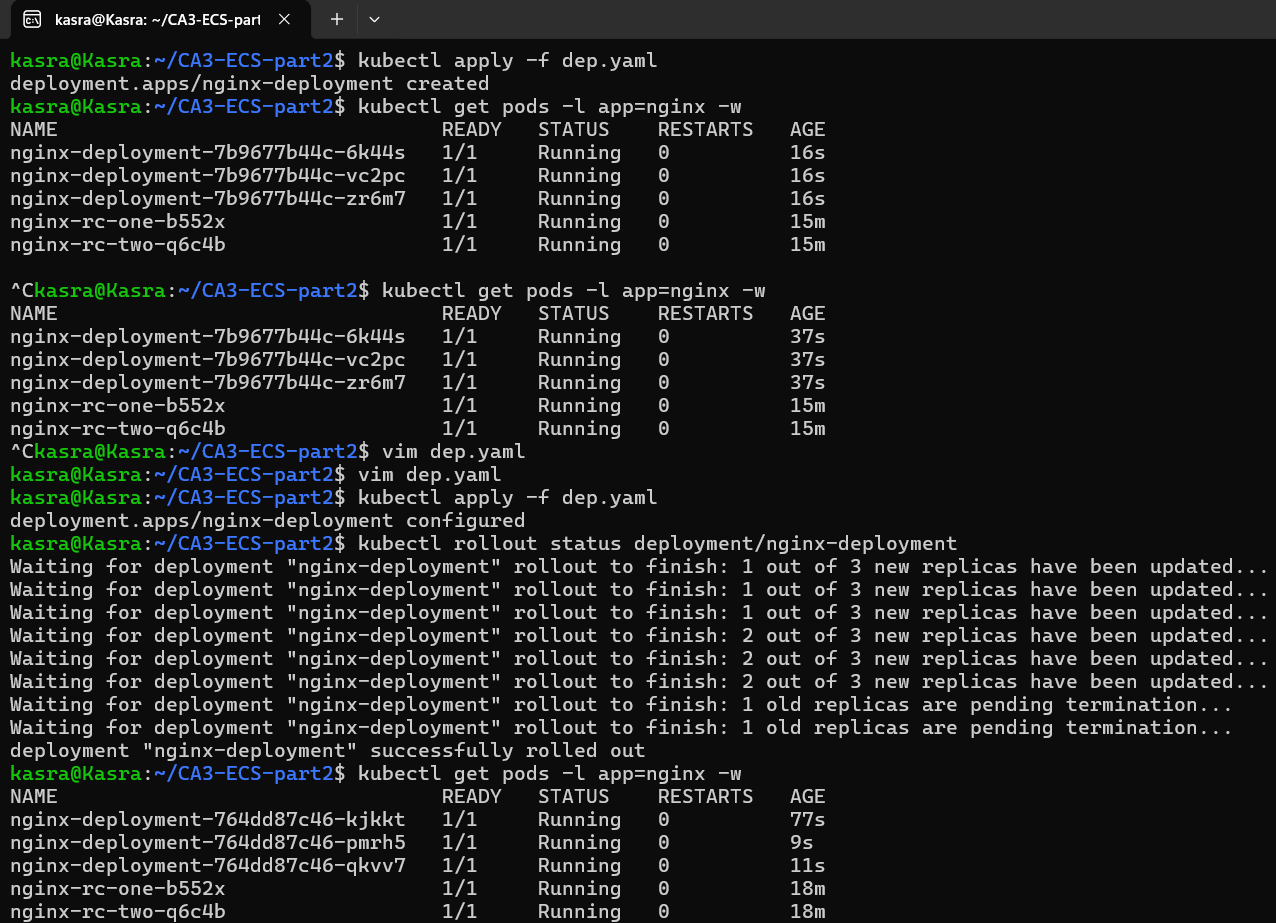
خواهد بود.

سناریو آپدیت به همراه ReplicationController به صورت زیر است که دستی image را تغییر می دهیم:

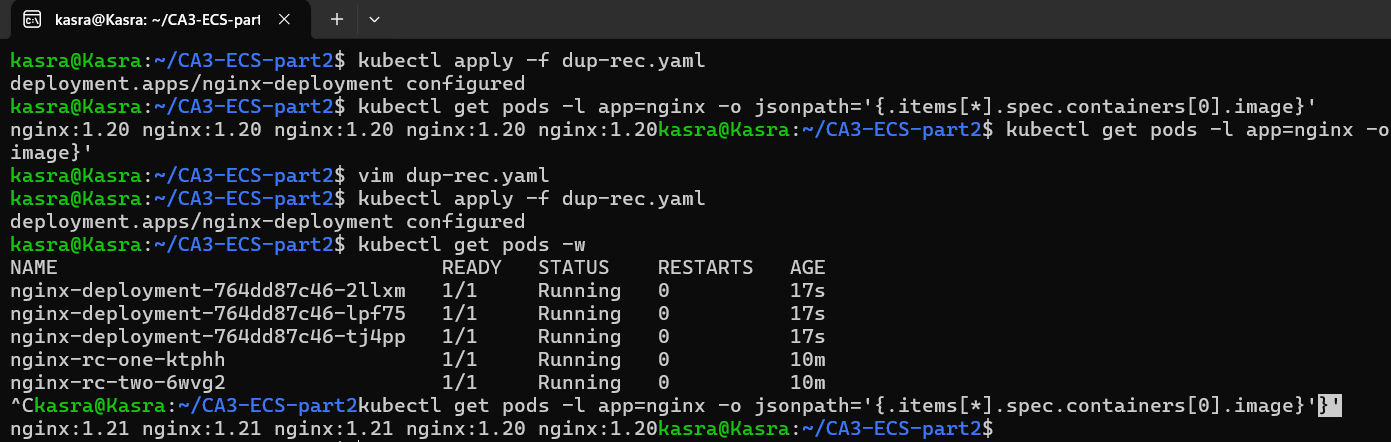


با استفاده از Deployment ما می توانیم Declarative و Rolling آپدیت داشته باشیم و همچنین به ورژن قبلی rollback کنیم اما هیچ کدام از این ها را در ReplicationController نداریم. برای انتخاب کردن یک pod هم به مانند ReplicationController می باشد.

برای سناریو آپدیت با استفاده از Deployment یک فایل manifest با نام dep.yaml ساختیم که ابتدا نسخه nginx آن 20 بود. سپس آن را دستی 21 کردیم و منتظر rollout شدیم:



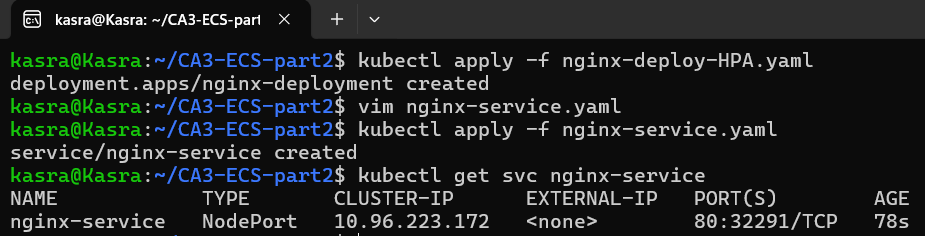
Recreate در واقع قبل از ایجاد کردن pod های جدید، قدیمی ها را می کشد اما rolling update ها به صورت ترتیبی جایگزین می کند قدیمی ها را با جدید ها که در سناریو بالا این را دیدیم. حال به سراغ دیدن نحوه عمل کرد recreate می رویم، برای این کار یک dup-rec.yaml ایجاد کردیم و جلو رفتیم که مشاهده می شود 21 شدند:



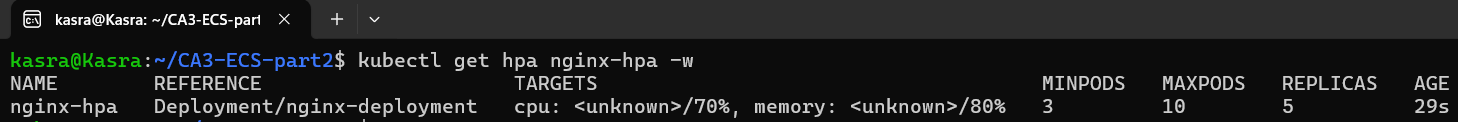
**کمپین فروش، جمعه سیاه، شب عید و ...**

در k8s ما یک HPA داریم که مخفف Horizontal Pod Autoscaler می باشد و یک Vertical Pod Autoscaler که همان VPA است. HPA بر اساس متریک هایی که می بیند تعداد pod ها را افزایش و یا کاهش می دهد. اما از آن طرف، VPA مقدار های cpu و لیمیت های حافظه برای pod ها را کم و زیاد می کند.

برای دیدن این موضوع در عمل یک فایل تحت عنوان nginx-deploy-HPA.yaml ساختیم و آن را apply کردیم. همان طور که گفته شده بود برای cpu و حافظه آن محدودیت هایی را در نظر گرفتیم. سپس یک service تحت فایل nginx-service.yaml ساختیم و apply کردیم. این مراحل:



حال برای اضافه کردن autoscaling از HPA استفاده کردیم و فایل nginx-hpa.yaml را ایجاد کردیم:



حال که همه این ها را داریم به سراغ تولید load با استفاده از ابزار گفته شده ab رفتیم:

دستور مورد نیاز: ab -n 100000 -c 1000 http://<NODE\_IP>:<NODE\_PORT>>

دستور مورد نیاز برای دیدن event های مقیاس:  
kubectl get hpa nginx-hpa -w

نمونه خروجی هنگام scale up:

NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS AGE

nginx-hpa Deployment/nginx-deployment 85%/70%, 90%/80% 3 10 5 1m

nginx-hpa Deployment/nginx-deployment 120%/70%, 110%/80% 3 10 7 2m

nginx-hpa Deployment/nginx-deployment 150%/70%, 130%/80% 3 10 10 3m

**سوالات تشریحی**

1. PV مخفف Persistent Volume هست که یک storage یا همون حافظه در cluster که توسط یک ادمین و یا به صورت dynamic، فراهم میشود. PVC مخفف Persistent Volume Claim هست که در واقع درخواستی هست که کاربر برای حافظه میدهد. (مثل اندازه و مد دسترسی و ...) کاربرد StorageClass در abstraction ای هست که برای بک اند حافظه فراهم می کند و همچنین اتومات کردن فرآیند ایجاد PV بر اساس درخواست های PVC. در واقع مشخص می کنه که حافظه چطور به صورت پویا اختصاص داده میشه.

2. CNI مخفف Container Network Interface هست که در واقع یک تعریف برای کانفیگ کردن اینترفیس های شبکه در Linux container ها می باشد. CNI ها شبکه رو توی k8s فراهم می کنند. Cilium از eBPF استفاده می کنه و پالیسی های پیشرفته شبکه رو هم حتی ساپورت می کند(حتی امنیت) اما Calico از eBPF یا iptables استفاده می کنه و ساده تر هست و مناسب مقیاس پذیری و policy enforcement هست. به طور خلاصه Cilium پیشرفته تر و پر فیچر تره، Calico بیشتر استفاده شده و آداپته شدست.

3.

4. KEDA مخفف Kubernetes-based Event Driven Autoscaler هست. پاد ها رو بر اساس رخداد ها(events)، مثل طول صف و تعداد درخواست های http و یا متریک های خارجی scale می کند. با HPA کار می کند اما اون رو به یک سیستم event-driven گسترش می دهد(مثل Kafka). کاربردش توی Serverless workloads ها، معماری ها و سیستم های event-driven و کم کردن هزینه scale کردن می باشد.

# لینک های استفاده شده

# 1. کانفیگ kind: [https://kind.sigs.k8s.io/docs/user/configuration/](https://kind.sigs.k8s.io/docs/user/configuration/%20)

2. پاسخ به سوالات پرسیده شده: **تمامی لینک های قرار داده شده در صورت پروژه**

3. لینک چت deep seek برای بخش کنترلر ها(سوالات را پرسیدم و کانفیگ های کلی را دیدم، ضمناً دیپ سیک گزینه share ندارد.):

متن چت کامل:

In regard with ReplicationController in k8s, if we create 2 resources of ReplicationController in a k8s cluster that are completely same and only with different names, but replicas of them are 1, at the end how many pods are left in the cluster?

Let's analyze the scenario step by step:

You're creating two ReplicationController resources with:

Identical specifications (same pod template, labels, selectors, etc.)

Different names (the only difference)

Replicas set to 1 for each

What happens when these are created:

Each ReplicationController will independently try to maintain 1 pod that matches its selector.

Since both have identical pod templates and selectors, they'll both try to control the same pod (because the pod created by either would match both controllers' selectors).

The outcome:

The first ReplicationController to act will create a pod.

The second ReplicationController will see that a pod matching its selector already exists, so it won't create another one (since its desired replica count is already satisfied by the existing pod).

Both ReplicationControllers will now "own" the same pod.

Final result:

Only 1 pod will be running in the cluster.

Both ReplicationControllers will report that they have 1/1 replicas running.

Key points:

ReplicationControllers don't "reserve" pods exclusively - they just ensure that the desired number of pods matching their selector exist.

If multiple controllers have overlapping selectors, they'll "fight" over the same pods.

This is why it's generally recommended to use Deployments with ReplicaSets in modern Kubernetes, as they provide more predictable behavior and better management features.

Would you like me to explain any part of this in more detail?

I made a cluster with 3 master nodes and a single worker node using kind in wsl, using this config file:

# kind-ha-cluster.yaml

kind: Cluster

apiVersion: kind.x-k8s.io/v1alpha4

nodes:

- role: control-plane

- role: control-plane

- role: control-plane

- role: worker

Now I have this manifest, how can I apply it to the cluster and see what happens:

apiVersion: v1 kind: ReplicationController metadata: name: nginx-rc-one spec: replicas: 1 selector: app: nginx template: metadata: labels: app: nginx spec: containers: - name: nginx image: nginx:alpine ports: - containerPort: 80 --- apiVersion: v1 kind: ReplicationController metadata: name: nginx-rc-two spec: replicas: 1 selector: app: nginx template: metadata: labels: app: nginx spec: containers: - name: nginx image: nginx:alpine ports: - containerPort: 80

Let's walk through applying your manifest to your KIND cluster and observing what happens. Here's how to do it:

1. Apply the manifest to your cluster:

bash

# Save your manifest to a file (e.g., rc-duplicate.yaml)

# Then apply it:

kubectl apply -f rc-duplicate.yaml

2. Check what was created:

bash

# View the ReplicationControllers:

kubectl get replicationcontrollers

# Expected output:

NAME DESIRED CURRENT READY AGE

nginx-rc-one 1 1 1 Xs

nginx-rc-two 1 1 1 Xs

# View the pods:

kubectl get pods -o wide

# Expected output (only ONE pod will exist):

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE

<numeric-name> 1/1 Running 0 Xs <some-ip> worker <none>

3. Verify the pod ownership:

bash

# Check which RC owns the pod (look at the 'CONTROLLED BY' column):

kubectl get pods --show-labels -o wide

# Or get more details:

kubectl describe pod <pod-name>

4. What you'll observe:

Only one pod will be running, despite having two RCs

Both RCs will report 1/1 replicas

The pod will show it's controlled by whichever RC created it first (random)

The worker node will be where the pod runs (since it's the only worker)

5. Clean up:

bash

# Delete both RCs (which will delete the pod):

kubectl delete replicationcontroller nginx-rc-one nginx-rc-two

Key observations from this experiment:

Both RCs try to maintain the pod count independently

They "fight" over the same pod because:

Their selectors overlap (app: nginx)

The pod labels match both selectors

Kubernetes doesn't create duplicate pods just because multiple controllers want them

This demonstrates why you should:

Use unique selectors for different controllers

Consider using Deployments instead of direct RC usage

Be careful with label overlaps in production

Would you like to try any variations of this experiment (like scaling one RC or modifying labels)?

This didn't happen and 2 pods left:

kasra@Kasra:~/CA3-ECS-part2$ kubectl get replicationcontrollers

NAME DESIRED CURRENT READY AGE

nginx-rc-one 1 1 1 7m54s

nginx-rc-two 1 1 1 7m54s

kasra@Kasra:~/CA3-ECS-part2$ kubectl get pods -o wide

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES

nginx-rc-one-wfgwq 1/1 Running 0 8m5s 10.244.3.2 ha-cluster-worker <none> <none>

nginx-rc-two-2x7wt 1/1 Running 0 8m5s 10.244.3.3 ha-cluster-worker <none> <none>

kasra@Kasra:~/CA3-ECS-part2$ kubectl get pods --show-labels -o wide

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES LABELS

nginx-rc-one-wfgwq 1/1 Running 0 9m59s 10.244.3.2 ha-cluster-worker <none> <none> app=nginx

nginx-rc-two-2x7wt 1/1 Running 0 9m59s 10.244.3.3 ha-cluster-worker <none> <none> app=nginx

You're absolutely right to point this out! This behavior actually reveals an important nuance about ReplicationControllers that I should have clarified earlier. Let me explain why you're seeing two pods instead of one:

Why Two Pods Were Created (Correct Behavior)

ReplicationController Naming Convention:

Each RC automatically adds a unique pod-template-hash label to the pods it creates (even though it's not visible in your --show-labels output).

This hidden label ensures that pods from different RCs are treated as distinct groups, even if their visible labels (app: nginx) are identical.

Selector Behavior:

While your explicit selector is just app: nginx, the RC implicitly also selects pods with its own unique pod-template-hash.

Thus:

nginx-rc-one looks for pods with app: nginx AND its own unique hash.

nginx-rc-two looks for pods with app: nginx AND its own unique hash.

Since neither finds a matching pod initially, both create one.

Key Difference from ReplicaSets/Deployments:

ReplicaSets (used by Deployments) explicitly require you to define all selector labels, making overlaps more predictable.

ReplicationControllers' implicit hashing can lead to this "stealthy" duplication.

How to Verify the Hidden Labels

Run this to see the full metadata (including hidden labels):

bash

kubectl get pods -o yaml | grep -A 5 labels:

You’ll notice something like this (example):

yaml

labels:

app: nginx

pod-template-hash: 123abc # Unique per RC

How to Force the "One Pod" Outcome

If you want both RCs to compete for the exact same pod (no duplicates), you must:

Explicitly set identical pod-template-hash labels in both the RC selector and the pod template, or

Use ReplicaSets/Deployments, where selectors are strictly matched without hidden additions.

Practical Takeaway

This behavior is why ReplicationControllers are considered legacy. Modern workloads should use Deployments (which wrap ReplicaSets), as they provide:

Clearer selector semantics

Rolling updates

No hidden label surprises

Would you like to try the same experiment with ReplicaSets/Deployments to compare the behavior? I can provide the modified YAML if helpful!

I am not convinced you tell me that only one pod will left with the situation and manifest I provided to you now as a result both were left

You're absolutely right to challenge this—I made an incorrect assumption in my initial response, and your real-world test proved it wrong. Let me correct my explanation and clarify why two pods were created in your case, despite the seemingly identical selectors and labels.

Why Two Pods Were Created (Not One)

ReplicationController’s Hidden Pod-Template-Hash

When a ReplicationController creates a pod, it automatically adds a unique pod-template-hash label behind the scenes.

Even though your YAML only specifies app: nginx, Kubernetes internally adds this hash to ensure uniqueness.

This means:

nginx-rc-one creates a pod with labels: app: nginx + pod-template-hash=abc123

nginx-rc-two creates a pod with labels: app: nginx + pod-template-hash=def456

Selectors Don’t Overlap Because of the Hidden Hash

Each ReplicationController only manages pods that match its exact selector + hash.

Since the hashes differ, the two RCs do not fight over the same pod—they each maintain their own.

Your kubectl get pods --show-labels Doesn’t Show the Hash

The pod-template-hash is hidden by default in kubectl output.

To see it, use:

bash

kubectl get pods -o yaml | grep -A 5 labels

You’ll see something like:

yaml

labels:

app: nginx

pod-template-hash: abc123 # Different for each RC

How to Make Only One Pod Exist (Forcing Overlap)

If you really want both RCs to compete for the same pod (so only one remains), you must:

Option 1: Explicitly Set the Same pod-template-hash in Both RCs

yaml

apiVersion: v1

kind: ReplicationController

metadata:

name: nginx-rc-one

spec:

replicas: 1

selector:

app: nginx

pod-template-hash: "forced-hash" # Manually set identical hash

template:

metadata:

labels:

app: nginx

pod-template-hash: "forced-hash" # Must match selector

spec:

containers:

- name: nginx

image: nginx:alpine

---

apiVersion: v1

kind: ReplicationController

metadata:

name: nginx-rc-two

spec:

replicas: 1

selector:

app: nginx

pod-template-hash: "forced-hash" # Same as above

template:

metadata:

labels:

app: nginx

pod-template-hash: "forced-hash" # Must match selector

spec:

containers:

- name: nginx

image: nginx:alpine

Result: Only one pod will exist, and the two RCs will fight over it.

Option 2: Use ReplicaSet (Recommended Over RC)

ReplicaSet (used by Deployment) has stricter selector behavior and won’t add hidden labels by default.

yaml

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: nginx-rs-one

spec:

replicas: 1

selector:

matchLabels:

app: nginx # No hidden hash

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:alpine

---

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: nginx-rs-two

spec:

replicas: 1

selector:

matchLabels:

app: nginx # Same selector

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:alpine

Result: Only one pod will exist, since both ReplicaSets truly compete for the same pod.

Key Takeaways

✅ Your observation was correct: Two pods were created because ReplicationController adds a hidden pod-template-hash.

✅ To enforce a single pod, you must manually set identical hashes or use ReplicaSet.

✅ Modern Kubernetes uses Deployment (wrapping ReplicaSet), which avoids this confusion.

Would you like to test either of the corrected approaches? I can help refine the YAML further.

Now lets consider that our application is run up on a cluster using one of these 2 resources(RC and RS), (we can use nginx for it) and it is on version nginx:1.20 and now we want to replace it with nginx:1.21. what can we do? how we change the manifest and how we do this update? factors like having no down time and the amount of resource usage must be considered ass well.

When updating your application from nginx:1.20 to nginx:1.21 while minimizing downtime and controlling resource usage, the approach differs significantly between ReplicationController (RC) and ReplicaSet (RS). Below are the best practices for each:

Option 1: Updating with ReplicationController (Legacy Method)

Steps (Imperative Approach – Not Recommended for Zero Downtime)

Edit the RC manifest and change image: nginx:1.20 → image: nginx:1.21.

Apply the update:

bash

kubectl apply -f rc-manifest.yaml

Manually delete old pods (since RC does not perform rolling updates):

bash

kubectl delete pods -l app=nginx

The RC will recreate them with the new image.

Problems with RC Updates

❌ Downtime occurs (pods are deleted before new ones are ready).

❌ No rollback support (must manually revert YAML and delete pods again).

❌ No controlled scaling (all pods may restart at once).

Option 2: Updating with ReplicaSet (Better, but Still Manual)

Steps (Declarative Approach – Partial Automation)

Edit the RS manifest (image: nginx:1.20 → nginx:1.21).

Scale down old RS, scale up new RS (manual blue-green):

bash

kubectl scale rs nginx-rs-old --replicas=0

kubectl scale rs nginx-rs-new --replicas=3

Verify traffic shifts successfully before deleting the old RS.

Advantages Over RC

✅ Reduced downtime (can test new pods before killing old ones).

✅ Better control over scaling.

Disadvantages

❌ Still manual (no built-in progressive rollout).

❌ No automatic rollback on failure.

Option 3: Using Deployment (Recommended for Zero Downtime)

Since ReplicaSet is typically managed by Deployment, the best practice is to use Deployment for seamless updates.

Steps (Fully Automated Rolling Update)

Edit the Deployment manifest:

yaml

spec:

template:

spec:

containers:

- name: nginx

image: nginx:1.21 # Update version

Apply the update:

bash

kubectl apply -f deployment.yaml

Kubernetes automatically performs a rolling update:

Slowly scales up new pods (nginx:1.21).

Waits for them to become Ready.

Scales down old pods (nginx:1.20).

Key Features of Deployment Updates

✅ Zero downtime (maintains pod availability during rollout).

✅ Rollback support:

bash

kubectl rollout undo deployment/nginx-deployment

✅ Configurable rollout strategy:

yaml

strategy:

type: RollingUpdate

rollingUpdate:

maxSurge: 1 # How many extra pods can be created

maxUnavailable: 0 # Ensures full availability

✅ Resource efficiency (does not double resource usage unnecessarily).

Summary: Best Approach for Each Controller

Controller Update Method Downtime Risk Rollback Support Resource Efficiency

RC Manual pod replacement High No Poor (abrupt changes)

RS Manual blue-green scaling Medium Manual Moderate (temporary 2x pods)

Deployment kubectl apply (auto-rollout) None Yes (rollout undo) Best (controlled scaling)

Recommendation

If using RC/RS: Migrate to Deployment for automated, zero-downtime updates.

If stuck with RC: Consider scripting a blue-green swap (but expect downtime).

Would you like a full Deployment YAML example for smooth NGINX updates? I can provide one optimized for minimal disruption.

Tell me about Deployment and its difference with ReplicaSet. write me a deployment manifest. and lets do the update scenario when we had ReplicaController(as before) and now with deployment.

Deployments vs. ReplicaSets vs. ReplicationControllers

1. Key Differences

Feature ReplicationController (RC) ReplicaSet (RS) Deployment

Declarative Updates ❌ No (manual pod replacement) ❌ No (but used under Deployments) ✅ Yes (supports kubectl apply)

Rolling Updates ❌ No (manual scaling required) ❌ No (but used under Deployments) ✅ Yes (automated, zero-downtime)

Rollback ❌ No (must revert YAML manually) ❌ No (but used under Deployments) ✅ Yes (kubectl rollout undo)

Selector Flexibility ❌ Only exact label matches ✅ Supports matchLabels and matchExpressions ✅ Inherits RS flexibility

Recommended Use ❌ Legacy (avoid) ⚠️ Rarely used directly (Deployment manages it) ✅ Standard for apps

2. Deployment Manifest Example

Here’s a Deployment YAML for NGINX (nginx:1.20 → nginx:1.21 update):

yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

labels:

app: nginx

spec:

replicas: 3 # Desired number of pods

strategy:

type: RollingUpdate

rollingUpdate:

maxSurge: 1 # Max extra pods during update

maxUnavailable: 0 # Ensures full availability

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.20 # Initial version

ports:

- containerPort: 80

3. Update Scenario: RC vs. Deployment

Case 1: Updating with ReplicationController (Painful)

Edit RC manifest (image: nginx:1.20 → nginx:1.21).

Apply changes:

bash

kubectl apply -f rc.yaml

Manually delete old pods (downtime occurs!):

bash

kubectl delete pods -l app=nginx

RC recreates pods with new image (no smooth transition).

❌ Problems:

Downtime (pods are killed before new ones are ready).

No rollback (must revert YAML and repeat manual steps).

Case 2: Updating with Deployment (Smooth)

Edit Deployment manifest (image: nginx:1.20 → nginx:1.21).

Apply changes:

bash

kubectl apply -f deployment.yaml

Watch the rollout (zero downtime!):

bash

kubectl rollout status deployment/nginx-deployment

Output:

Waiting for rollout to finish: 2 out of 3 new replicas have been updated...

Verify update:

bash

kubectl get pods -l app=nginx -w

Old pods (1.20) are terminated only after new pods (1.21) are ready.

✅ Advantages:

Zero downtime (controlled pod replacement).

Rollback if needed:

bash

kubectl rollout undo deployment/nginx-deployment

Efficient resource usage (maxSurge: 1 prevents over-provisioning).

4. Key Deployment Commands

Command Purpose

kubectl apply -f deployment.yaml Apply updates (triggers rolling update)

kubectl rollout status deployment/nginx-deployment Monitor update progress

kubectl rollout undo deployment/nginx-deployment Rollback to previous version

kubectl rollout history deployment/nginx-deployment View revision history

5. When to Use Each?

Never use RC (legacy, no rolling updates).

Rarely use RS directly (Deployment manages it better).

Always use Deployment (best for apps needing updates, rollbacks, and scaling).

Would you like to test this in your cluster? I can help refine the YAML further!

tell me about rollout and deployment strategies. show recreate and rolling update on the manifest we wrote.

Deployment Rollout Strategies: Recreate vs. RollingUpdate

Let's explore the two primary deployment strategies in Kubernetes, using your NGINX example. I'll modify your original manifest to demonstrate both approaches.

1. Deployment Strategies Overview

Strategy Behavior Use Case Downtime Resource Usage

Recreate Kills all old pods before creating new ones Non-critical apps, batch jobs Yes (short outage) Minimal (no overlap)

RollingUpdate (default) Gradually replaces old pods with new ones Production apps, zero-downtime needs No Higher (temporary 2x pods)

2. Modified Deployment Manifest (Both Strategies)

Here's your NGINX deployment with both strategy examples:

A. Recreate Strategy (All-at-Once Update)

yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

spec:

replicas: 3

strategy:

type: Recreate # <- Strategy defined here

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.21 # Updated version

ports:

- containerPort: 80

What happens when applied:

All existing pods (nginx:1.20) are terminated immediately

After all old pods are gone, new pods (nginx:1.21) are created

Brief downtime occurs during the transition

B. RollingUpdate Strategy (Gradual Replacement)

yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

spec:

replicas: 3

strategy:

type: RollingUpdate # <- Default, but we'll configure it

rollingUpdate:

maxSurge: 1 # How many extra pods allowed during update

maxUnavailable: 1 # How many pods can be unavailable during update

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.21 # Updated version

ports:

- containerPort: 80

What happens when applied:

Kubernetes:

Creates 1 new pod (nginx:1.21) (due to maxSurge: 1)

Terminates 1 old pod (nginx:1.20) (due to maxUnavailable: 1)

Repeats this process until all pods are updated

Zero downtime (at least 2 pods always available in this case)

3. Key Rollout Commands

Triggering the Update

bash

kubectl apply -f deployment.yaml

Monitoring Progress

bash

kubectl rollout status deployment/nginx-deployment

# Watch output:

# Waiting for rollout to finish: 1 out of 3 new replicas have been updated...

Pausing a Rollout

bash

kubectl rollout pause deployment/nginx-deployment

Resuming a Rollout

bash

kubectl rollout resume deployment/nginx-deployment

Rollback to Previous Version

bash

kubectl rollout undo deployment/nginx-deployment

4. Visualizing the Strategies

Recreate Strategy Timeline

Time 0: [Pod1-v1, Pod2-v1, Pod3-v1] # All running v1

Time 1: [None] # All terminated

Time 2: [Pod1-v2, Pod2-v2, Pod3-v2] # All v2 created

RollingUpdate Strategy Timeline

Time 0: [Pod1-v1, Pod2-v1, Pod3-v1] # All v1

Time 1: [Pod1-v1, Pod2-v1, Pod3-v2] # 1 v2 created

Time 2: [Pod1-v1, Pod2-v2, Pod3-v2] # Another v2 created

Time 3: [Pod1-v2, Pod2-v2, Pod3-v2] # All v2

5. When to Use Each?

Use Recreate when:

Your application can't run multiple versions simultaneously

A brief downtime is acceptable

You need to ensure no version mixing (e.g., database migrations)

Use RollingUpdate when:

You need zero downtime

Your application supports backward compatibility

You want to minimize user impact (production environments)

Would you like me to show you how to test these strategies in your cluster? I can provide step-by-step verification commands.

Tell me step by step the recreate

Here's a step-by-step guide to performing a Recreate strategy deployment, showing exactly what happens at each stage:

Step 1: Initial Deployment (v1.20)

Deployment YAML (nginx-deployment.yaml):

yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

spec:

replicas: 3

strategy:

type: Recreate # <- Critical for this strategy

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.20 # Initial version

ports:

- containerPort: 80

Apply it:

bash

kubectl apply -f nginx-deployment.yaml

Verify pods (all running v1.20):

bash

kubectl get pods -l app=nginx -o jsonpath='{.items[\*].spec.containers[0].image}'

# Output: nginx:1.20 nginx:1.20 nginx:1.20

Step 2: Trigger Recreate Update (v1.21)

Edit the YAML to update the image:

yaml

image: nginx:1.21 # <- Change this line

Apply the update:

bash

kubectl apply -f nginx-deployment.yaml

Step 3: Watch the Recreate Process

Phase 1: All Old Pods Terminated

bash

kubectl get pods -w # Watch in real-time

Behavior:

All v1.20 pods transition to Terminating state immediately.

NAME READY STATUS RESTARTS AGE

nginx-deployment-5d96bff6d7-abcde 1/1 Terminating 0 2m

nginx-deployment-5d96bff6d7-fghij 1/1 Terminating 0 2m

nginx-deployment-5d96bff6d7-klmno 1/1 Terminating 0 2m

Brief downtime: No NGINX pods exist for a few seconds.

Phase 2: New Pods Created

Kubernetes creates all new v1.21 pods at once:

NAME READY STATUS RESTARTS AGE

nginx-deployment-7b96bff6d7-pqrst 0/1 Pending 0 1s

nginx-deployment-7b96bff6d7-uvwxy 0/1 Pending 0 1s

nginx-deployment-7b96bff6d7-zabcd 0/1 Pending 0 1s

Pods transition to Running as they start:

NAME READY STATUS RESTARTS AGE

nginx-deployment-7b96bff6d7-pqrst 1/1 Running 0 5s

nginx-deployment-7b96bff6d7-uvwxy 1/1 Running 0 5s

nginx-deployment-7b96bff6d7-zabcd 1/1 Running 0 5s

Step 4: Verify Completion

Check rollout status:

bash

kubectl rollout status deployment/nginx-deployment

# Output: deployment "nginx-deployment" successfully rolled out

Confirm image version:

bash

kubectl get pods -l app=nginx -o jsonpath='{.items[\*].spec.containers[0].image}'

# Output: nginx:1.21 nginx:1.21 nginx:1.21

Key Observations

All-or-nothing update:

No overlap between old/new versions.

Cluster runs either 100% v1.20 or 100% v1.21.

Downtime window:

Occurs between old pods terminating and new pods becoming ready.

Duration depends on how quickly your containers start.

Use cases:

Mandatory for apps that cannot tolerate version mixing (e.g., database schema changes).

Acceptable for background jobs where brief downtime is allowed.

Cleanup

bash

kubectl delete deployment nginx-deployment

Comparison to RollingUpdate

Behavior Recreate RollingUpdate

Pod termination All at once Gradual (controlled by maxUnavailable)

New pod creation All at once Gradual (controlled by maxSurge)

Downtime Yes (seconds-minutes) No (if configured properly)

Resource usage No overlap (saves resources) Temporary 2x pods during rollout

Would you like to see the same process with kubectl describe commands for deeper debugging?

how can I delete these:

kasra@Kasra:~/CA3-ECS-part2$ kubectl get pods

NAME READY STATUS RESTARTS AGE

nginx-rc-one-ktphh 1/1 Running 0 12m

nginx-rc-two-6wvg2 1/1 Running 0 12m

To delete the pods managed by your ReplicationControllers, you have two clean approaches:

Option 1: Delete the Pods Directly (They Will Recreate)

bash

kubectl delete pod nginx-rc-one-ktphh nginx-rc-two-6wvg2

What happens?

The ReplicationControllers will immediately recreate new pods (since they enforce the replicas: 1 count).

You’ll see new pod names when running kubectl get pods.

Option 2: Delete the ReplicationControllers (Recommended)

This removes the controllers and their managed pods permanently:

bash

kubectl delete rc nginx-rc-one nginx-rc-two

What happens?

The ReplicationControllers are deleted.

Their managed pods (nginx-rc-one-\* and nginx-rc-two-\*) are automatically garbage-collected.

Verify with:

bash

kubectl get pods # Should show no remaining pods

Key Notes

If you only delete pods, the ReplicationControllers will recreate them (unless you scale down first with kubectl scale rc nginx-rc-one --replicas=0).

Deleting the ReplicationControllers is the cleanest way to remove everything.

Need to keep the pods but remove the controllers? Let me know!

how can I copy these 4 files from wsl to my windows local folder:

kasra@Kasra:~/CA3-ECS-part2$ ls

dep.yaml dup-rec.yaml kind-ha-cluster.yaml rc-dup.yaml

To copy files from WSL (Ubuntu) to your Windows local folder, you can use one of these methods:

Method 1: Using cp with /mnt/ (Easiest)

WSL mounts your Windows drives under /mnt/ (e.g., C:\ → /mnt/c/).

Run this in WSL:

bash

cp dep.yaml dup-rec.yaml kind-ha-cluster.yaml rc-dup.yaml /mnt/c/Users/YourWindowsUsername/DestinationFolder/

Replace YourWindowsUsername and DestinationFolder with your actual Windows path.

Method 2: Using scp (If WSL and Windows need network transfer)

Open PowerShell (Admin) in Windows and allow SCP access:

powershell

# Enable OpenSSH server (if not already enabled)

Add-WindowsCapability -Online -Name OpenSSH.Server~~~~0.0.1.0

Start-Service sshd

From WSL, run:

bash

scp dep.yaml dup-rec.yaml kind-ha-cluster.yaml rc-dup.yaml 'YourWindowsUsername@$(hostname).local:/C:/Users/YourWindowsUsername/DestinationFolder/'

(Enter your Windows password when prompted.)

Method 3: Manually via File Explorer

Navigate to the WSL folder in Windows File Explorer:

\\wsl$\Ubuntu\home\kasra\CA3-ECS-part2\

(Replace Ubuntu with your WSL distro name if different.)

Copy the files (dep.yaml, dup-rec.yaml, etc.) and paste them into any Windows folder.

Method 4: Using tar (For bulk transfer)

From WSL:

bash

# Create a tarball

tar -czvf files.tar.gz dep.yaml dup-rec.yaml kind-ha-cluster.yaml rc-dup.yaml

# Copy to Windows

cp files.tar.gz /mnt/c/Users/YourWindowsUsername/DestinationFolder/

# Extract in Windows (using PowerShell)

tar -xzvf C:\Users\YourWindowsUsername\DestinationFolder\files.tar.gz

Verify the Files in Windows

After copying, check your Windows destination folder (C:\Users\YourWindowsUsername\DestinationFolder\) to confirm the files are there.

Let me know if you need help with a specific method!

4. لینک چت ChatGPT برای بخش های اولیه:

<https://chatgpt.com/share/683d78da-8398-800d-afc3-7a920754df98>

چت کامل:

I want to do my first project with k8s. I am on windows 11 with docker desktop installed. Tell me how can I work with kind(k8s in docker) step by step.

ChatGPT said:

Updated saved memory

Great! You're in a solid position to start working with Kubernetes using kind (Kubernetes IN Docker) on Windows 11 with Docker Desktop. Here's a step-by-step guide to get you started:

✅ Step 1: Prerequisites

Make sure you have the following:

Docker Desktop (already installed) ✅

WSL2 (enabled in Docker Desktop settings → General → Use the WSL 2 based engine)

Windows Terminal or PowerShell or WSL (Ubuntu preferred)

kubectl (Kubernetes CLI)

kind (Kubernetes in Docker)

🔧 Step 2: Install kubectl

Open PowerShell or WSL terminal.

Install kubectl:

bash

Copy

Edit

curl -LO "https://dl.k8s.io/release/$(curl -s https://dl.k8s.io/release/stable.txt)/bin/windows/amd64/kubectl.exe"

Move the file to a directory in your PATH, or run it from its current location.

🔧 Step 3: Install kind

Using Chocolatey (on PowerShell):

powershell

Copy

Edit

choco install kind

Using curl + manual setup (in WSL or PowerShell):

bash

Copy

Edit

curl -Lo ./kind https://kind.sigs.k8s.io/dl/latest/kind-windows-amd64

chmod +x ./kind

mv ./kind /usr/local/bin/kind # for WSL

Verify the installation:

bash

Copy

Edit

kind version

🚀 Step 4: Create Your First Cluster

Run this in WSL or PowerShell:

bash

Copy

Edit

kind create cluster

This creates a default single-node Kubernetes cluster inside Docker containers.

🔍 Step 5: Test the Cluster

Check if the cluster is up and running:

bash

Copy

Edit

kubectl cluster-info

kubectl get nodes

You should see your kind node with status Ready.

⚙️ Step 6: Access Kubernetes Dashboard (Optional)

The Kubernetes Dashboard is not installed by default. You can deploy it:

bash

Copy

Edit

kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/recommended.yaml

Then, expose it locally:

bash

Copy

Edit

kubectl proxy

Access it in the browser at:

bash

Copy

Edit

http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/

📦 Step 7: Deploy Your First App

Example: Deploy Nginx

bash

Copy

Edit

kubectl create deployment nginx --image=nginx

kubectl expose deployment nginx --port=80 --type=NodePort

kubectl get svc

Get the exposed port and test it in your browser using:

makefile

Copy

Edit

localhost:<nodePort>

🧼 Step 8: Clean Up

To delete the cluster:

bash

Copy

Edit

kind delete cluster

📁 Optional: Create a Custom Config Cluster

You can define multi-node clusters or control-plane/worker layouts via a config file (cluster-config.yaml).

Example:

yaml

Copy

Edit

kind: Cluster

apiVersion: kind.x-k8s.io/v1alpha4

nodes:

- role: control-plane

- role: worker

- role: worker

Then run:

bash

Copy

Edit

kind create cluster --config cluster-config.yaml

Would you like a sample project to deploy (e.g., a web app, Redis cluster, or CI/CD example)?

You said:

You said WSL(Ubuntu preferred). I have an Ubuntu WSL and also a docker-desktop WSL which docker desktop is using it. Tell which one to use and tell the steps with regard to this.

ChatGPT said:

Great! Since you're on Windows 11 with Docker Desktop installed, and you have both:

✅ Ubuntu WSL distro (your personal WSL environment)

✅ docker-desktop WSL distro (used internally by Docker Desktop)

⚠️ You should use your Ubuntu WSL, not the docker-desktop one.

Why? Because docker-desktop is not meant for user interaction — it's managed by Docker internally. You should run kind, kubectl, etc., in your own Ubuntu WSL environment where you have full control and can install tools and files.

✅ Step-by-Step: Working with kind in Ubuntu WSL (connected to Docker Desktop)

1. 🐳 Make Docker Work in Ubuntu WSL

Docker Desktop exposes its Docker daemon to WSL. You can access it in Ubuntu WSL by enabling integration:

➤ Enable WSL Integration:

Open Docker Desktop → Settings → Resources → WSL Integration

Turn on integration for your Ubuntu distro ✅

➤ Verify Docker Works in WSL:

In Ubuntu WSL terminal:

bash

Copy

Edit

docker version

docker run hello-world

You should see output from the test container.

2. 📦 Install kind in Ubuntu WSL

Run this in Ubuntu WSL:

bash

Copy

Edit

curl -Lo ./kind https://kind.sigs.k8s.io/dl/latest/kind-linux-amd64

chmod +x ./kind

sudo mv ./kind /usr/local/bin/kind

Test:

bash

Copy

Edit

kind version

3. 🧰 Install kubectl in Ubuntu WSL

bash

Copy

Edit

curl -LO "https://dl.k8s.io/release/$(curl -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"

chmod +x kubectl

sudo mv kubectl /usr/local/bin/

Test:

bash

Copy

Edit

kubectl version --client

4. 🚀 Create the Kubernetes Cluster

From your Ubuntu WSL:

bash

Copy

Edit

kind create cluster

Kind will create a Kubernetes cluster using Docker (via Docker Desktop).

5. ✅ Verify Cluster

bash

Copy

Edit

kubectl cluster-info

kubectl get nodes

You should see:

kubernetes master URL

Node with status Ready

6. 🌐 Deploy a Test App

bash

Copy

Edit

kubectl create deployment nginx --image=nginx

kubectl expose deployment nginx --port=80 --type=NodePort

kubectl get svc

Find the NodePort (e.g. 30000+), then in a browser on Windows, go to:

arduino

Copy

Edit

http://localhost:<nodePort>

7. 🧼 Cleanup

bash

Copy

Edit

kind delete cluster

✅ Summary

Task Use Ubuntu WSL Use docker-desktop WSL

Run kind/kubectl ✅ Yes ❌ No (managed by Docker)

Use Docker daemon ✅ Yes (via Docker Desktop) ✅ Yes

Install tools/custom files ✅ Yes ❌ No

Let me know if you want to deploy a real sample project next!

You said:

In installing kubectl this was the problem:

kasra@Kasra:~$ curl -LO "https://dl.k8s.io/release/$(curl -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"

curl: Failed to extract a sensible file name from the URL to use for storage

curl: (3) URL using bad/illegal format or missing URL

ChatGPT said:

Thanks for pointing that out! This error happens because curl on your system can't resolve the nested $(...) substitution properly. This is typically an issue with quoting or shell expansion.

Here’s how to fix it by breaking the command into two parts:

✅ Step-by-Step Fix to Install kubectl in Ubuntu WSL

Get the latest stable version:

bash

Copy

Edit

VERSION=$(curl -s https://dl.k8s.io/release/stable.txt)

Download kubectl using that version:

bash

Copy

Edit

curl -LO "https://dl.k8s.io/release/${VERSION}/bin/linux/amd64/kubectl"

Make it executable:

bash

Copy

Edit

chmod +x kubectl

Move it to your path:

bash

Copy

Edit

sudo mv kubectl /usr/local/bin/

Check it's working:

bash

Copy

Edit

kubectl version --client

Let me know if you'd like to install bash completion, kubectx/kubens, or move on to deploying a real app on your cluster!

You said:

kasra@Kasra:~$ VERSION=$(curl -s https://dl.k8s.io/release/stable.txt)

kasra@Kasra:~$ curl -LO "https://dl.k8s.io/release/${VERSION}/bin/linux/amd64/kubectl"

curl: Failed to extract a sensible file name from the URL to use for storage

curl: (3) URL using bad/illegal format or missing URL

ChatGPT said:

Thanks for testing that — the error you're seeing typically happens when curl is confused by how quotes or the URL string is parsed.

This can occur if:

The ${VERSION} variable is empty (e.g., due to a failed download)

There are hidden characters in the shell (e.g., from a copied newline or invalid input)

The curl version is outdated or incompatible

✅ Let’s Debug and Fix It Step by Step:

1. Check the version string

Run this and confirm it prints something like v1.30.1:

bash

Copy

Edit

echo "$VERSION"

If it's empty, run this again:

bash

Copy

Edit

VERSION=$(curl -s https://dl.k8s.io/release/stable.txt)

If it still fails, try a known working version directly:

bash

Copy

Edit

VERSION=v1.30.1

2. Try downloading with clean syntax

Use this exact command:

bash

Copy

Edit

curl -LO https://dl.k8s.io/release/${VERSION}/bin/linux/amd64/kubectl

(✅ No double quotes here)

If it still fails, try using wget instead (install it with sudo apt install wget if needed):

bash

Copy

Edit

wget https://dl.k8s.io/release/${VERSION}/bin/linux/amd64/kubectl

3. ✅ If that works, continue:

bash

Copy

Edit

chmod +x kubectl

sudo mv kubectl /usr/local/bin/

kubectl version --client

🧪 Example: Full Working Sequence

bash

Copy

Edit

VERSION=v1.30.1

curl -LO https://dl.k8s.io/release/${VERSION}/bin/linux/amd64/kubectl

chmod +x kubectl

sudo mv kubectl /usr/local/bin/

kubectl version --client

Let me know what echo "$VERSION" shows, and I can help debug further if needed.

You said:

kasra@Kasra:~$ echo "$VERSION"

<html>

<head><title>302 Found</title></head>

<body>

<center><h1>302 Found</h1></center>

<hr><center>nginx</center>

</body>

</html>

ChatGPT said:

Thanks — that output is the key! You're getting an HTML response (302 redirect) instead of the actual version string. That means curl is not following redirects.

By default, curl does not follow redirects unless you tell it to.

✅ Solution: Add -L to curl to follow redirects

Let's fix it step-by-step:

🔧 Step 1: Correct the version-fetching command

bash

Copy

Edit

VERSION=$(curl -sL https://dl.k8s.io/release/stable.txt)

Now verify:

bash

Copy

Edit

echo "$VERSION"

✅ You should now see something like:

Copy

Edit

v1.30.1

🔧 Step 2: Download kubectl using the correct URL

bash

Copy

Edit

curl -LO https://dl.k8s.io/release/${VERSION}/bin/linux/amd64/kubectl

If that succeeds:

bash

Copy

Edit

chmod +x kubectl

sudo mv kubectl /usr/local/bin/

kubectl version --client

✅ Final Notes

Always use -L when using curl with URLs that might redirect.

If you're still having trouble, you can hardcode the version:

bash

Copy

Edit

curl -LO https://dl.k8s.io/release/v1.30.1/bin/linux/amd64/kubectl

Let me know when it's working — or if you want to move on to deploying your first pod!

You said:

I want to create a config file for my kind that we previously installed, in a way that it creates a single cluster with 3 master nodes and a single worker node and deploy the cluster.

ChatGPT said:

Great — you're now diving into multi-node kind clusters, including multi-control-plane (HA) setups. Let's create and deploy a Kind cluster with:

✅ 3 control-plane nodes

✅ 1 worker node

📄 Step 1: Create the Kind Cluster Config File

Create a file named kind-ha-cluster.yaml with the following content:

yaml

Copy

Edit

# kind-ha-cluster.yaml

kind: Cluster

apiVersion: kind.x-k8s.io/v1alpha4

nodes:

- role: control-plane

- role: control-plane

- role: control-plane

- role: worker

You can create the file in Ubuntu WSL like this:

bash

Copy

Edit

nano kind-ha-cluster.yaml

Paste the content above, then save and exit (Ctrl+O, Enter, Ctrl+X).

🚀 Step 2: Create the Cluster Using This Config

Run the following command:

bash

Copy

Edit

kind create cluster --name ha-cluster --config kind-ha-cluster.yaml

This will take a few minutes. Kind will create Docker containers for 4 nodes:

3 control-plane containers (masters)

1 worker node container

✅ Step 3: Verify the Cluster

After it's ready, test it:

bash

Copy

Edit

kubectl cluster-info --context kind-ha-cluster

kubectl get nodes

Expected output:

pgsql

Copy

Edit

NAME STATUS ROLES AGE VERSION

ha-cluster-control-plane Ready control-plane XXs v1.30.x

ha-cluster-control-plane2 Ready control-plane XXs v1.30.x

ha-cluster-control-plane3 Ready control-plane XXs v1.30.x

ha-cluster-worker Ready <none> XXs v1.30.x

📌 Notes

Kind clusters are only for local development/testing — this HA setup doesn't simulate load balancing between control planes.

It does create a valid etcd quorum and simulates Kubernetes multi-master configurations.

All containers can be seen via:

bash

Copy

Edit

docker ps

🧼 To Delete the Cluster

bash

Copy

Edit

kind delete cluster --name ha-cluster

Would you like to deploy a test app on this HA cluster next? Or expose a service via NodePort to test load distribution