

i created one master instance and 2 worker nodes in our cluster and nothing else.

we run :`kubectl get pod`

what will this command give me

If you created one master instance and two worker nodes in your Kubernetes cluster and haven't deployed any applications or created any pods, running the command:

text

```
kubectl get pod
```

will likely return **no pods** in the default namespace, or an empty list, because no pods have been created yet. The output would look something like this:

text

```
No resources found in default namespace.
```

Explanation:

- The `kubectl get pod` command lists all pods in the **default namespace** unless a specific namespace is provided (e.g., `kubectl get pod -n <namespace>`).
- Since you mentioned that you only set up the cluster (one master and two worker nodes) and did nothing else, there are no user-created pods or workloads (e.g., Deployments, StatefulSets, or ReplicaSets) running in the cluster.

- By default, Kubernetes does not create any pods in the `default` namespace unless you explicitly deploy something or if certain system components (e.g., DNS or Kubernetes Dashboard) are configured to run there. However, system pods (like `kube-scheduler`, `kube-controller-manager`, or `coredns`) typically run in the `kube-system` namespace, not the default namespace.

If you want to see system pods:

To check for pods in the `kube-system` namespace (where Kubernetes system components run), you can use:

```
text
```

```
kubectl get pod -n kube-system
```

This might show pods like `coredns`, `kube-proxy`, or others, depending on your cluster setup (e.g., whether you're using a managed service like GKE, EKS, or a self-managed cluster). The output might look like:

```
text
```

| NAME | READY | STATUS | RESTARTS | AGE |
|-----------------|-------|---------|----------|-----|
| coredns-xxx-xxx | 1/1 | Running | 0 | 1h |
| kube-proxy-xxx | 1/1 | Running | 0 | 1h |
| ... | | | | |

Notes:

- If you want to see pods across **all namespaces**, use:

```
text
```

```
kubectl get pod --all-namespaces
```

- If you expected pods to be running but see none, ensure that:
 - You have deployed workloads (e.g., via `kubectl apply -f <yaml-file>` or `kubectl create deployment`).
 - You are checking the correct namespace.

- Your cluster is healthy (check node status with `kubectl get nodes` to confirm the master and worker nodes are in a `Ready` state).

If you have more details about your cluster setup (e.g., is it a managed cluster like GKE/AKS/EKS, or a custom setup like k8s with kubeadm?), I can provide a more tailored response. Let me know if you need further clarification!

kube system namespace is there by default?

Yes, the `kube-system` namespace is created **by default** in every Kubernetes cluster. It is a predefined namespace used to run critical system components and services required for the cluster to function properly.

Details:

- The `kube-system` namespace is automatically created when a Kubernetes cluster is initialized (e.g., using `kubeadm init` or by a managed Kubernetes service like GKE, EKS, or AKS).
- It contains pods and resources for core Kubernetes components, such as:
 - `kube-scheduler` : Schedules pods to nodes.
 - `kube-controller-manager` : Manages controllers like replication controllers.
 - `kube-proxy` : Handles networking and load balancing on each node.
 - `coredns` or `kube-dns` : Provides DNS resolution for services within the cluster.
 - Other add-ons or system services (e.g., metrics-server, dashboard, or cloud-specific components).
- You can verify its existence by running:

```
text
```

```
kubectl get namespaces
```

This will list all namespaces, including `kube-system` , with an output like:

```
text
```

| NAME | STATUS | AGE |
|---------|--------|-----|
| default | Active | 1h |

| | | |
|-----------------|--------|----|
| kube-system | Active | 1h |
| kube-public | Active | 1h |
| kube-node-lease | Active | 1h |

- To see pods in the **kube-system** namespace, use:

```
text
```

```
kubectl get pod -n kube-system
```

Notes:

- You should avoid deploying user applications to the **kube-system** namespace, as it is reserved for Kubernetes system components.
- Other default namespaces that may exist include **default** (for user workloads if no namespace is specified), **kube-public** (for resources accessible without authentication), and **kube-node-lease** (for node lease objects).

If you have more questions about namespaces or your cluster setup, let me know!

now i wanna create a node with a container apache on
(i) default namespace
(ii) arka namespace - which i create

To create a pod with an Apache container in both the **default** namespace and a custom **arka** namespace, I'll guide you through the steps. Since you mentioned a "node" but likely meant a **pod** (as nodes are physical/virtual machines in the cluster, while pods contain containers), I'll assume you want to create a pod running an Apache container.

Below are the steps to:

1. Create a pod with an Apache container in the **default** namespace.
2. Create a custom namespace called **arka** and deploy a pod with an Apache container in it.

I'll use the **httpd** image from Docker Hub for Apache, as it's a common choice for running an Apache web server.

Prerequisites

- Ensure your Kubernetes cluster (with 1 master and 2 worker nodes) is running and `kubectl` is configured to communicate with it.
- Verify the cluster status with:

```
text
```

```
kubectl get nodes
```

Ensure all nodes are in the `Ready` state.

- The `httpd` image will be used for the Apache container (official Apache HTTP server image).

(i) Create a Pod with Apache in the Default Namespace

1. Create a Pod Definition:

Create a YAML file named `apache-pod-default.yaml` to define the pod:

```
yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: apache-pod
  namespace: default
spec:
  containers:
  - name: apache-container
    image: httpd:latest
    ports:
    - containerPort: 80
```

This YAML defines a pod named `apache-pod` in the `default` namespace, running an Apache container using the `httpd:latest` image, exposing port 80.

2. Apply the Pod Definition:

Run the following command to create the pod:

```
text
```

```
kubectl apply -f apache-pod-default.yaml
```

3. Verify the Pod:

Check the status of the pod in the **default** namespace:

text

```
kubectl get pod -n default
```

Expected output:

text

| NAME | READY | STATUS | RESTARTS | AGE |
|------------|-------|---------|----------|-----|
| apache-pod | 1/1 | Running | 0 | 10s |

4. Check Pod Details (optional):

To confirm the pod is running correctly:

text

```
kubectl describe pod apache-pod -n default
```

(ii) Create a Pod with Apache in the Arka Namespace

1. Create the Arka Namespace:

Run the following command to create a namespace called **arka** :

text

```
kubectl create namespace arka
```

Verify the namespace was created:

text

```
kubectl get namespaces
```

Expected output (partial):

text

| NAME | STATUS | AGE |
|-------------|--------|-----|
| arka | Active | 10s |
| default | Active | 1h |
| kube-system | Active | 1h |
| ... | | |

2. Create a Pod Definition for the Arka Namespace:

Create a YAML file named `apache-pod-arka.yaml` :

yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: apache-pod
  namespace: arka
spec:
  containers:
  - name: apache-container
    image: httpd:latest
    ports:
    - containerPort: 80
```

This YAML defines a pod named `apache-pod` in the `arka` namespace, running an Apache container.

3. Apply the Pod Definition:

Run the following command to create the pod in the `arka` namespace:

text

```
kubectl apply -f apache-pod-arka.yaml
```

4. Verify the Pod:

Check the status of the pod in the `arka` namespace:

text

```
kubectl get pod -n arka
```

Expected output:

```
text
```

| NAME | READY | STATUS | RESTARTS | AGE |
|------------|-------|---------|----------|-----|
| apache-pod | 1/1 | Running | 0 | 10s |

5. Check Pod Details (optional):

To confirm the pod is running correctly:

```
text
```

```
kubectl describe pod apache-pod -n arka
```

Notes

- **Pod Name:** I used the same pod name (`apache-pod`) in both namespaces because namespaces allow duplicate names (pods are scoped to their namespace). If you want different names, update the `metadata.name` field in the YAML files.
- **Accessing Apache:** The Apache container is running, but to access it, you may need to:
 - Expose the pod via a Kubernetes **Service** (e.g., `ClusterIP` , `NodePort` , or `LoadBalancer`).
 - Use `kubectl port-forward` to test locally, e.g.:

```
text
```

```
kubectl port-forward apache-pod 8080:80 -n default
```

Then access `http://localhost:8080` in your browser.

- **Cluster Setup:** If your cluster is a managed service (e.g., GKE, EKS, AKS) or a custom setup (e.g., kubeadm), ensure the nodes have enough resources to run the pods and that the container runtime (e.g., Docker, containerd) can pull the `httpd` image.

- **Image Pull:** The `httpd:latest` image will be pulled from Docker Hub. Ensure your cluster has internet access, or specify a different Apache image if needed.
- **Error Handling:** If the pod is not in the `Running` state, check logs with:

```
text
```

```
kubectl logs apache-pod -n <namespace>
```

Or describe the pod to diagnose issues:

```
text
```

```
kubectl describe pod apache-pod -n <namespace>
```

Optional: Expose Apache for Testing

To make the Apache server accessible, you can create a **Service** for each pod. For example, for the `default` namespace:

1. Create a Service YAML (`apache-service-default.yaml`):

```
yaml
```

```
apiVersion: v1
kind: Service
metadata:
  name: apache-service
  namespace: default
spec:
  selector:
    app: apache
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: NodePort
```

Update the pod YAML to include the `app: apache` label:

```
yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: apache-pod
  namespace: default
  labels:
    app: apache
spec:
  containers:
  - name: apache-container
    image: httpd:latest
    ports:
    - containerPort: 80
```

2. Apply the Service:

text

```
kubectl apply -f apache-service-default.yaml
```

3. Get the Service details to find the NodePort:

text

```
kubectl get svc -n default
```

Access the Apache server using a worker node's IP and the assigned NodePort (e.g., <http://<node-ip>:<node-port>>).

Repeat similar steps for the `arka` namespace if needed.

Let me know if you need help with exposing the Apache server, debugging, or any other Kubernetes tasks! If you meant something else by "node" (e.g., adding a new worker node), please clarify, and I can adjust the response.

explain all commands written in the 3 pics above



Based on the handwritten notes in the three images, I'll explain each Kubernetes-related command step by step. The notes appear to document various Kubernetes operations, including cluster management, pod creation, namespace usage, and resource configuration. I'll break it down by section and provide explanations.

Image 1: Kubernetes Configuration and Namespace Commands

1. Change Context and Namespace:

- `kubectl config set-context --current --namespace=<namespace>`
 - Changes the current namespace for the `kubectl` context to the specified namespace (e.g., `default` or a custom one). This sets the default namespace for subsequent `kubectl` commands unless overridden with `-n`.
- Note: "We can create namespace through cmd or yaml file" indicates namespaces can be created either directly with a command or via a YAML file.

2. View Minify Config:

- `kubectl config view --minify --output='json'`
 - Displays a minimized version of the `kubectl` configuration (e.g., from `~/.kube/config`) in JSON format, showing only the current context and relevant details.

3. Set Context to Default:

- `kubectl config set-context --current`
 - Sets or updates the current context (e.g., to `default` if not specified). This ensures `kubectl` operates in the default context.

4. Create Namespace via YAML:

- `vim namespace.yaml`
 - Opens a file named `namespace.yaml` in the `vim` editor to define a namespace.
 - Example content (inferred from notes):

```
yaml
```

```
apiVersion: v1
kind: Namespace
metadata:
  name: devops
```

- `kubectl apply -f namespace.yaml`
 - Applies the `namespace.yaml` file to create the `devops` namespace.
- `kubectl get ns`
 - Lists all namespaces in the cluster to verify the `devops` namespace was created.
- `kubectl create -f https://<url>` (namespace used for identification)
 - Creates a resource (e.g., namespace) from a URL, where the namespace is used for identification.

5. Get Namespaces with Labels:

- `kubectl get namespaces --show-labels`
 - Lists all namespaces and includes any labels associated with them.

Image 2: Pod Creation, Resource Limits, and Namespace Management

1. Get Nodes and Pods:

- `kubectl get node` (to check healthy)
 - Lists all nodes in the cluster and their status (e.g., `Ready` indicates healthy nodes).
- `kubectl get pod --all-namespaces`
 - Lists all pods across all namespaces.
- `kubectl get pod -n kube-system` (all pod inside k8s)
 - Lists pods in the `kube-system` namespace, which contains Kubernetes system components (e.g., `coredns`, `kube-proxy`).
- `kubectl get pod -n kube-public`

- Lists pods in the `kube-public` namespace, typically for publicly accessible resources.

2. Create Pod via YAML:

- `vim my-pod.yaml`
- Opens `my-pod.yaml` in `vim` to define a pod.
- Example content (inferred):

```
yaml

apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
  - name: apache
    image: nginx:1.14
```

- `--save & exit`
- Saves and exits the `vim` editor.
- `kubectl apply -f my-pod.yaml`
- Creates the pod defined in `my-pod.yaml` in the default namespace (since no `-n` is specified).
- `kubectl get pod -n default`
- Lists pods in the `default` namespace to verify the `web-app` pod.

3. Pod Status and Description:

- `kubectl get pod`
- Lists pods in the current namespace (default if not changed).
- `kubectl describe pod web-app`
- Provides detailed information about the `web-app` pod, including events and status.
- `kubectl get ns (namespace)`

- Lists all namespaces (likely a typo, intended as `kubectl get ns`).

4. Create Namespace and Pod:

- `kubectl create ns prod` (Namespace created)
 - Creates a namespace named `prod`.
- `kubectl get ns`
 - Verifies the `prod` namespace.
- `vim dev-pod.yaml`
 - Opens `dev-pod.yaml` to define a pod.
 - Example content (inferred):

yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: dev-app
spec:
  containers:
  - name: nginx
    image: nginx
```

- `kubectl apply -f dev-pod.yaml`
 - Applies `dev-pod.yaml` (likely intended with `-n prod` to deploy in the `prod` namespace, though not specified).
- `kubectl get pod -n prod`
 - Lists pods in the `prod` namespace.

5. Cluster Management with Kops:

- `kops get clusters`
 - Lists all clusters managed by `kops`.
- `kops update cluster --name <cluster-name> --yes`
 - Updates the specified cluster configuration and applies changes with confirmation.

- `kops delete cluster <cluster-name> --yes`
- Deletes the specified cluster with confirmation.

Image 3: Resource Limits and Pod Deployment

1. Create Pod with Resource Limits:

- `vim test.yaml`
- Opens `test.yaml` in `vim` to define a pod with resource limits.
- Example content (inferred):

```
yaml

apiVersion: v1
kind: Pod
metadata:
  name: test-pod
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "1500Mi"
        cpu: "500m"
```

- `kubectl apply -f test.yaml --namespace=devops`
- Creates the `test-pod` in the `devops` namespace with the specified resource limits.
- `kubectl get namespace mem-cpu-demo --namespace=devops --output=yaml`
- Retrieves the `mem-cpu-demo` namespace configuration in YAML format (likely a typo; intended namespace might be `devops`).

2. Deploy Another Pod:

- `vim test2.yaml`
- Opens `test2.yaml` in `vim`.
- Example content (inferred):

yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: test2-pod
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      requests:
        memory: "500Mi"
      limits:
        memory: "800Mi"
        cpu: "400m"
```

- change Pod name -> metadata: name: test2-demo
- Updates the pod name to test2-demo in the YAML.
- kubectl apply -f test2.yaml --namespace=devops
- Applies test2.yaml to create test2-demo in the devops namespace.

3. Debugging and Validation:

- vim nginx-resource.yaml
- Opens nginx-resource.yaml (likely to increase resources).
- Example content (inferred, with increased resources):

yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      requests:
        cpu: "200m"
        memory: "256Mi"
```



```
memory: "256Mi"
limits:
  cpu: "500m"
  memory: "512Mi"
```

- `copy code`
 - Copies the code (likely for reuse).
- `vim devops.yaml`
 - Opens `devops.yaml`.
 - Example content (inferred):

yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: devops-pod
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      requests:
        cpu: "200m"
        memory: "256Mi"
      limits:
        cpu: "500m"
        memory: "512Mi"
```

- `kubectl apply -f devops.yaml --namespace=devops`
 - Applies `devops.yaml` in the `devops` namespace.
- `kubectl describe namespace devops`
 - Describes the `devops` namespace to check its details.

Key Concepts and Notes

- **Namespaces:** Used to isolate resources (e.g., `default`, `devops`, `prod`).
Commands like `kubectl get ns` list them, and `kubectl create ns` or YAML files

create them.

- **Pods:** Basic units running containers (e.g., `web-app` , `test-pod`). Defined via YAML and applied with `kubectl apply` .
- **Resource Limits:** `requests` and `limits` (e.g., CPU, memory) ensure pods use resources appropriately. If limits are exceeded, the pod may not deploy.
- **Context:** `kubectl config set-context` manages the current namespace and cluster context.
- **Kops:** A tool for managing Kubernetes clusters (e.g., `kops update` , `kops delete`).

Potential Issues

- Some commands lack namespaces (e.g., `kubectl apply -f my-pod.yaml` defaults to `default`).
- Typos (e.g., `get namespace` instead of `get ns`) may cause errors.
- Resource limits must match node capacity; otherwise, pods may remain in `Pending` state.

If you need clarification on any command or help executing them, let me know!

Change Context and Namespace

i didn't understand the context. what is it and how is it different from namespace

Let's break this down to clarify the concepts of **context** and **namespace** in Kubernetes, and how they differ.

What is a Context?

- A **context** in Kubernetes is a set of configuration details that `kubectl` uses to communicate with a specific cluster. It includes:
 - The **cluster** you want to interact with (e.g., its API server endpoint and credentials).
 - The **user** (e.g., your authentication details, like a certificate or token).

- The **namespace** (optional default namespace for operations, which can be overridden).
- The context is stored in the `kubeconfig` file (typically `~/.kube/config`) and defines the "environment" or "session" for your `kubectl` commands.
- You can have multiple contexts in your `kubeconfig` file (e.g., for different clusters or users) and switch between them using the `kubectl config use-context` command.

Example:

- You might have a context called `dev-cluster` for a development cluster and `prod-cluster` for a production cluster. Switching contexts changes which cluster and user `kubectl` targets.

Command to Change Context:

- `kubectl config set-context --current --namespace=<namespace>`
 - This modifies the current context to use a specific namespace by default. The `--current` flag refers to the context you're currently using.
- To switch to a different context entirely:
 - `kubectl config use-context <context-name>`
 - E.g., `kubectl config use-context dev-cluster` switches to the `dev-cluster` context.

What is a Namespace?

- A **namespace** is a logical partition within a single Kubernetes cluster. It provides a way to organize and isolate resources (e.g., pods, services, deployments) for different teams, projects, or environments (e.g., `dev`, `prod`, `test`).
- Namespaces are useful in multi-tenant environments to prevent naming conflicts (e.g., two teams can have a pod named `web-app` in different namespaces).
- By default, every cluster has namespaces like `default`, `kube-system`, and `kube-public`. You can create custom namespaces (e.g., `devops`) using `kubectl create namespace <name>` or a YAML file.

Example:

- In the `default` namespace, you might run a `web-app` pod. In the `devops` namespace, you could run a different `web-app` pod without conflict.

Command to Set Namespace:

- `kubectl config set-context --current --namespace=devops`
 - Sets the default namespace for the current context to `devops`. Future `kubectl` commands (e.g., `kubectl get pod`) will target the `devops` namespace unless overridden with `-n <namespace>`.
- To apply a namespace for a single command:
 - `kubectl get pod -n devops`
 - Lists pods specifically in the `devops` namespace.

Key Differences

| Aspect | Context | Namespace |
|---------------|--|---|
| Definition | A configuration set for cluster access (cluster, user, namespace). | A logical partition within a cluster for resource isolation. |
| Scope | Applies to the entire <code>kubectl</code> session or configuration. | Applies within a single cluster to group resources. |
| Purpose | Defines which cluster and user to interact with. | Organizes and isolates resources (e.g., for teams or projects). |
| Configuration | Managed in <code>kubeconfig</code> file with <code>kubectl config</code> . | Created and managed with <code>kubectl create ns</code> or YAML. |
| Switching | Use <code>kubectl config use-context <name></code> . | Use <code>-n <namespace></code> in commands or set via context. |
| Default | No default context until set; multiple contexts possible. | Default namespace is <code>default</code> unless changed. |

How They Work Together

- A **context** can include a default namespace, but you can override it per command using `-n`.
- For example:

- If your context is set to use the `devops` namespace, `kubectl get pod` will list pods in `devops`.
- If you run `kubectl get pod -n default`, it will override the context and list pods in the `default` namespace instead.
- Changing the context (e.g., to a different cluster) might reset the namespace to `default` unless explicitly set in the new context.

Practical Example

1. Check Current Context:

- `kubectl config current-context`
 - Outputs the current context name (e.g., `minikube`).

2. Set Namespace in Current Context:

- `kubectl config set-context --current --namespace=devops`
 - Updates the current context to use `devops` as the default namespace.

3. Verify:

- `kubectl get pod`
 - Lists pods in the `devops` namespace.
- `kubectl get pod -n default`
 - Lists pods in the `default` namespace, overriding the context setting.

Summary

- **Context** is about configuring your connection to a cluster and user, with an optional default namespace.
- **Namespace** is about organizing resources within that cluster.
- Use `kubectl config` commands to manage contexts and `-n` or context settings to manage namespaces.

If you're still unsure or want to try this with your cluster, let me know, and I can guide you through setting up a specific example!

i didn't understand again, hows context different from cluster

I apologize for the confusion! Let's clarify the difference between **context** and **cluster** in Kubernetes with a simpler explanation.

What is a Cluster?

- A **cluster** is the physical or virtual infrastructure where Kubernetes runs. It consists of:
 - **Nodes**: Machines (physical or virtual) that run your applications and Kubernetes components.
 - **Master Node(s)**: Manages the cluster (e.g., API server, scheduler, controller manager).
 - **Worker Nodes**: Run your application pods.
 - **Networking**: Connects all nodes and enables communication between pods.
- A cluster is the actual environment where your Kubernetes workloads (e.g., pods, services) are deployed and managed.
- You can have multiple clusters (e.g., a development cluster, a production cluster) managed by the same or different Kubernetes setups.

Example:

- You might have a cluster called `dev-cluster` on a cloud provider like AWS, and another called `prod-cluster` on Google Cloud.

What is a Context?

- A **context** is a configuration setting in your `kubectl` tool that tells it how to connect to a specific cluster. It includes:
 - The **cluster** name (e.g., `dev-cluster` or `prod-cluster`).
 - The **user** credentials (e.g., a certificate, token, or username/password) to authenticate with that cluster.
 - An optional **namespace** default for operations within that cluster.
- The context is stored in the `kubeconfig` file (usually `~/.kube/config`) and allows you to switch between different clusters or user settings without changing your configuration manually each time.

Example:

- You might have a context called `dev-context` that points to the `dev-cluster` with a specific user, and another context called `prod-context` for the `prod-cluster` with a different user.

Key Differences

| Aspect | Cluster | Context |
|------------|---|--|
| Definition | The actual Kubernetes environment (nodes, master, workloads). | A configuration in <code>kubectl</code> to access a cluster. |
| Scope | The physical/virtual setup running Kubernetes. | A logical setting in your local <code>kubectl</code> configuration. |
| Purpose | Runs and manages your applications and Kubernetes components. | Defines how <code>kubectl</code> connects to a cluster and user. |
| Location | Exists on the cloud, on-premises, or locally (e.g., Minikube). | Stored in the <code>kubeconfig</code> file on your local machine. |
| Switching | You don't switch clusters directly; you manage them via tools like <code>kops</code> or cloud consoles. | Switch with <code>kubectl config use-context <context-name></code> . |
| Default | No default cluster; you define multiple clusters. | No default context until set; multiple contexts possible. |

How They Relate

- A **cluster** is the target environment, while a **context** is the lens through which `kubectl` interacts with that cluster.
- You can have multiple clusters, and each cluster can be associated with one or more contexts (e.g., different users accessing the same cluster).
- When you run a `kubectl` command (e.g., `kubectl get pods`), the current context determines which cluster and user are targeted.

Practical Example

1. Check Clusters:

- `kubectl config get-clusters`

- Lists all clusters defined in your `kubeconfig` (e.g., `dev-cluster`, `prod-cluster`).

2. Check Contexts:

- `kubectl config get-contexts`
 - Lists all contexts, showing the cluster, user, and namespace for each (e.g., `dev-context` → `dev-cluster`, `user1`, `devops` namespace).

3. Switch Context:

- `kubectl config use-context dev-context`
 - Switches `kubectl` to use the `dev-context`, which points to the `dev-cluster`.

4. Run a Command:

- `kubectl get nodes`
 - Retrieves node information from the `dev-cluster` based on the current context.

Analogy

- Think of a **cluster** as a physical office building where work happens.
- A **context** is like your employee badge and access card, which specifies which building (cluster) you can enter and what permissions (user) you have inside.

Why It Matters

- If you have multiple clusters (e.g., dev, staging, prod), you use contexts to switch between them without manually updating connection details.
- The cluster remains the same unless you reconfigure your infrastructure, but you can change contexts to interact with different clusters or users.

Does this clear things up? If you'd like, I can walk you through setting up multiple contexts for your current cluster or provide more examples! Let me know!

default context mean?

The term "**default context**" in Kubernetes refers to the context that `kubectl` uses automatically when you run a command without explicitly specifying a different context. Let's break it down:

What is a Default Context?

- A **context** is a set of configuration details in your `kubeconfig` file (typically `~/.kube/config`) that defines:
 - The **cluster** to connect to (e.g., its API server address).
 - The **user** for authentication (e.g., a certificate or token).
 - An optional **default namespace** for operations.
- The **default context** is the context that is currently active or selected in your `kubectl` configuration unless you switch to another context manually.

How is the Default Context Set?

- When you first set up `kubectl` (e.g., with Minikube, a cloud provider, or `kubeadm`), a context is typically created and set as the default. For example:
 - If you use Minikube, it creates a context like `minikube` and sets it as the default.
 - The default context is the one marked with an asterisk (`*`) when you run:

```
text
```

```
kubectl config get-contexts
```

Example output:

```
text
```

| CURRENT | NAME | CLUSTER | AUTHINFO | NAMESPACE |
|---------|----------|--------------|-----------|-----------|
| * | minikube | minikube | minikube | default |
| | prod | prod-cluster | prod-user | prod |

Here, `minikube` is the default context because of the `*`.

- If no context is explicitly set, `kubectl` uses the first context listed in the `kubeconfig` file or the one last used.

What Does "Default" Mean in Practice?

- When you run a command like `kubectl get pods` without specifying a context or namespace, `kubectl` uses the default context to determine:
 - Which cluster to query (e.g., `minikube` or your cloud cluster).
 - Which user credentials to use.
 - Which namespace to target (if a default namespace is set in the context; otherwise, it defaults to the `default` namespace).
- For example, if your default context is `minikube` with the `default` namespace, `kubectl get pods` will show pods in the `default` namespace of the Minikube cluster.

How to Check or Set the Default Context

1. Check Current Context:

- `kubectl config current-context`
 - Outputs the name of the default context (e.g., `minikube`).

2. List All Contexts:

- `kubectl config get-contexts`
 - Shows all contexts with the current (default) one marked with `*`.

3. Set a Default Context:

- `kubectl config use-context <context-name>`
 - E.g., `kubectl config use-context minikube` sets `minikube` as the default context.

4. Modify Default Namespace in Context:

- `kubectl config set-context --current --namespace=devops`
 - Updates the default namespace for the current context to `devops`.

Key Points

- The **default context** is not a fixed setting; it's the context you're currently working with until you change it.
- It's different from the **default namespace** (e.g., `default`), which is a resource isolation level within a cluster. The default context can include a default namespace, but you can override it with `-n <namespace>` in commands.
- If you have multiple clusters (e.g., `dev`, `prod`), the default context determines which one `kubectl` interacts with by default.

Example Scenario

- You have two clusters: `dev-cluster` and `prod-cluster`.
- Your `kubeconfig` has two contexts:
 - `dev-context` (points to `dev-cluster`, user `dev-user`, namespace `dev`).
 - `prod-context` (points to `prod-cluster`, user `prod-user`, namespace `prod`).
- If `dev-context` is the default (marked with `*`), running `kubectl get pods` will target `dev-cluster` and show pods in the `dev` namespace unless overridden.

Current Context Check (as of now)

Since the current date and time is 02:11 PM IST on Thursday, August 21, 2025, you can run:

- `kubectl config current-context` to see your default context right now. If it's not set or you're unsure, it might default to the first context in your `kubeconfig` (e.g., from Minikube or a cloud setup).

Does this make it clearer? If you want help checking or setting your default context, let me know, and I can guide you with specific commands!

u yap too much nonsense, what's default context? prod-cluster? dev-cluster? what?

Fair point—let's cut to the chase. The **default context** is the context `kubectl` uses automatically when you run commands, based on your `kubeconfig` file. It's not inherently `prod-cluster` or `dev-cluster`—it's whatever context is currently set as active (marked with `*` in `kubectl config get-contexts`).

To find out your default context right now:

- Run `kubectl config current-context`.
- If it's not set, it might default to the first context in your `kubeconfig`, like `minikube` or something from your AWS setup.

If you've been working with `prod-cluster` or `dev-cluster`, check with the command above to see which one's active. To switch, use `kubectl config use-context <context-name>` (e.g., `kubectl config use-context prod-cluster`). That's it! Let me know if you need to check yours specifically.

resource limits pod i didn't understand the yaml codes

I'll explain the **resource limits** in a Kubernetes pod YAML file in a simple and direct way. Resource limits define how much CPU and memory a pod (and its containers) can use, preventing it from overwhelming the cluster. Let's break down the YAML code you might have seen in your notes (e.g., from the images) and clarify it step by step.

What Are Resource Limits?

- **Requests:** The minimum resources a container needs to run (e.g., "I need at least this much").
- **Limits:** The maximum resources a container can use (e.g., "Don't let me exceed this").
- If a pod tries to use more than its limits, Kubernetes may kill it or throttle it, depending on the situation.

Example YAML with Resource Limits

Here's a sample pod YAML from your notes (e.g., `test.yaml` or `nginx-resource.yaml`):

yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pod
spec:
  containers:
```

```
- name: nginx
  image: nginx
  resources:
    requests:
      memory: "500Mi"
      cpu: "200m"
    limits:
      memory: "1500Mi"
      cpu: "500m"
```

Breaking It Down

1. **apiVersion: v1** and **kind: Pod**

- Defines this as a pod resource in Kubernetes version 1.x.

2. **metadata:**

- `name: test-pod`
 - Names the pod `test-pod`.

3. **spec:**

- Defines the pod's specifications.

4. **containers:**

- Lists the containers in the pod.
- `- name: nginx`
 - Names the container `nginx`.
- `image: nginx`
 - Uses the `nginx` image from Docker Hub.

5. **resources:**

- Where the magic happens for resource limits.
- **requests:**
 - `memory: "500Mi"`
 - Requests 500 mebibytes (Mi) of memory. This is the minimum the container expects to function.

- `cpu: "200m"`
 - Requests 200 millicpus (m), where 1000m = 1 CPU core. So, this is 0.2 CPU cores.
 - Kubernetes schedules the pod on a node with at least this much available.
- **limits:**
 - `memory: "1500Mi"`
 - Caps memory usage at 1500 Mi. If the container exceeds this, it may be terminated.
 - `cpu: "500m"`
 - Caps CPU usage at 500m (0.5 CPU cores). Kubernetes throttles or kills the container if it goes over.

Why Use This?

- **Stability:** Prevents one pod from hogging all resources and crashing the node.
- **Fairness:** Ensures resources are shared among pods.
- **Scheduling:** Helps Kubernetes place pods on nodes with enough capacity.

From Your Notes

- In `test.yaml`, you had:
 - `limits: memory: "1500Mi", cpu: "500m"`
 - This sets a hard cap.
- In `test2.yaml`, you added `requests: memory: "500Mi"` with `limits: memory: "800Mi", cpu: "400m"`.
 - Requests are the baseline, limits are the ceiling.
- If resources exceed limits (e.g., node has only 600Mi memory available but limit is 1500Mi), the pod won't deploy and stays in `Pending` state.

Applying It

- Save the YAML (e.g., `test-pod.yaml`).

- Run: `kubectl apply -f test-pod.yaml -n <namespace>`
- Check: `kubectl get pod -n <namespace>`
- Describe for details: `kubectl describe pod test-pod -n <namespace>`

Quick Tips

- Use **Mi** for mebibytes (1024-based) or **M** for megabytes (1000-based).
- Use **m** for millicpus (e.g., 500m = 0.5 cores).

... If you want to get more details about a pod, use `kubectl describe pod <pod-name> -n <namespace>`.