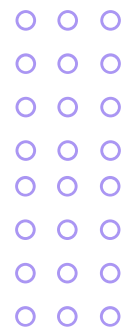


A 10x10 grid of 100 small purple circles. The circles are arranged in 10 rows and 10 columns, forming a square pattern. Each circle is a solid purple color and has a diameter of approximately 10 pixels. The grid is centered on a white background.

FUNDAMENTALS



About Me

My name is Kirk Patrick, and I have 17+ years of experience in the technology industry, including:

- 10 years as a Software Engineer
- 4 years as a DevOps Engineer
- 3 years as a Solution Architect

Academic Background

- Technical Degree in Accounting — Rui Barbosa School
- Bachelor's Degree in Systems Analysis and Development — FIAP
- Professional Baccalaureate in Data Science — Datatech Florida
- Postgraduate Degree in Machine Learning — FIAP
- Currently pursuing a Postgraduate Degree in Astronomy with an emphasis in Planetary Geology – FGE São Paulo / Space Today

Additional Information

- Speaker at Campus Party Brazil (Goiânia Edition), my hometown.
- Guest Mentor for NASA events in Brazil (NASA Space Apps Challenge)
- Certified Professional in AWS, Azure, and GCP (Google Cloud Platform).

Contact

- LinkedIn → <https://www.linkedin.com/in/kirkgo/>

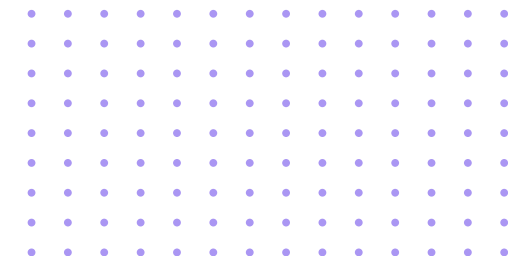
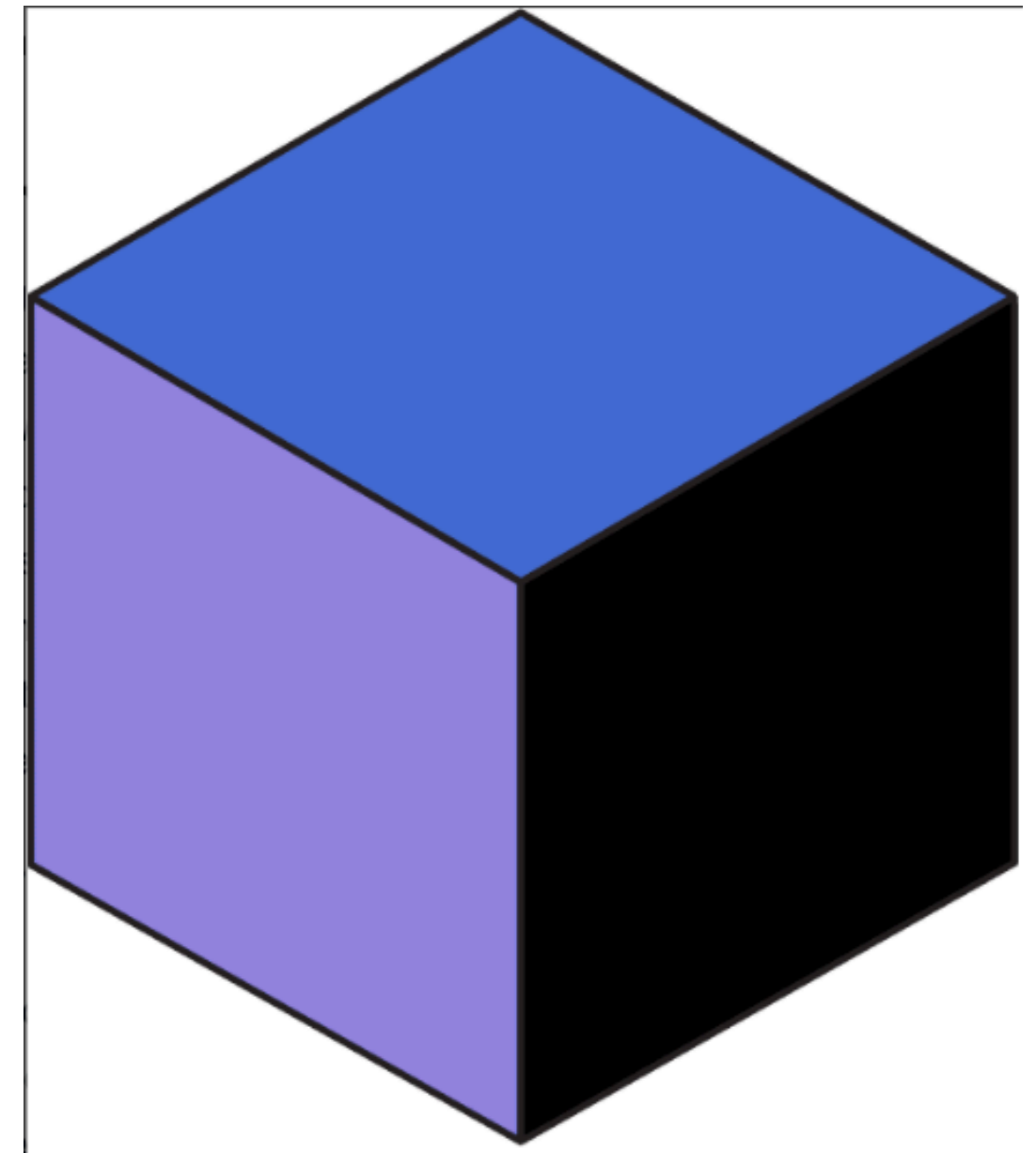


From Monolith to Microservices – The Beginning

How it All Started

The journey of deployment architectures began with monoliths—single, unified applications that were simple to deploy and manage.

- Monolith Architecture
 - Easy to monitor and deploy—one package, one team.
 - Deployment was infrequent and manually handled by operations.
 - Failures were resolved by manually shifting workloads to healthy servers.
 - Limitations
 - Difficult to scale components independently.
 - Even small changes required redeploying the entire application.
 - Poor modularity often led to tightly coupled, hard-to-maintain designs.

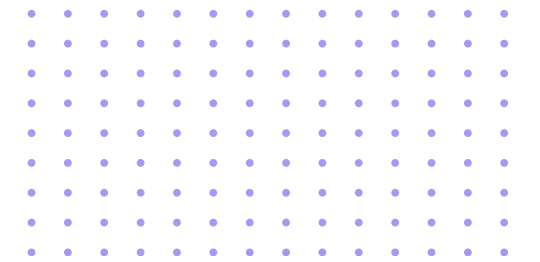
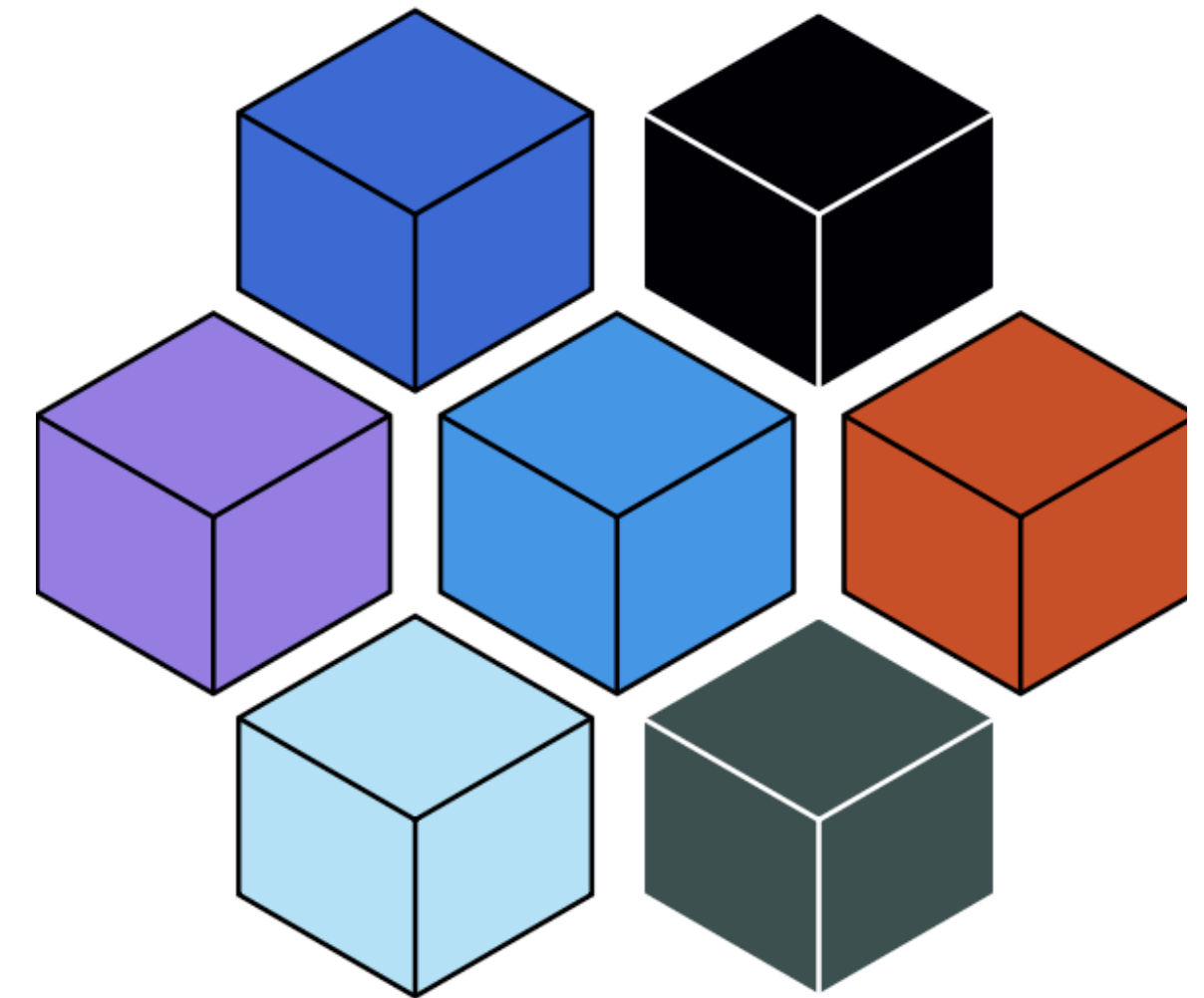


Enter Microservices – More Power, More Complexity

Microservice-based Architectures

As systems evolved, monoliths gave way to microservices— independent, small services that could be developed, deployed, and scaled separately.

- Advantages
 - Services are independently deployable and scalable.
 - Teams gained autonomy, often merging development and operations (DevOps).
- Challenges
 - Managing hundreds or thousands of services is complex.
 - Hosting each service on its own machine is cost-prohibitive.
 - Services must coexist on shared infrastructure, often with conflicting dependencies and resource needs.



Introduction to Kubernetes

Why Do We Need Kubernetes?

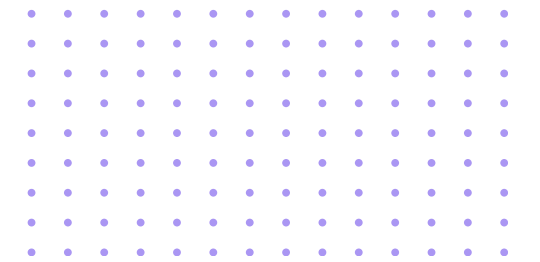
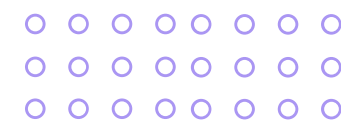
After packaging applications as container images, we need a system to run and manage them reliably—this is where Kubernetes comes in.

- **What Kubernetes Does**

- Groups multiple machines and exposes them as a single compute resource.
- Acts like an OS for the datacenter: abstracts away machine-level details.
- Users simply declare their needs; Kubernetes figures out how to meet them.

- **How It Works**

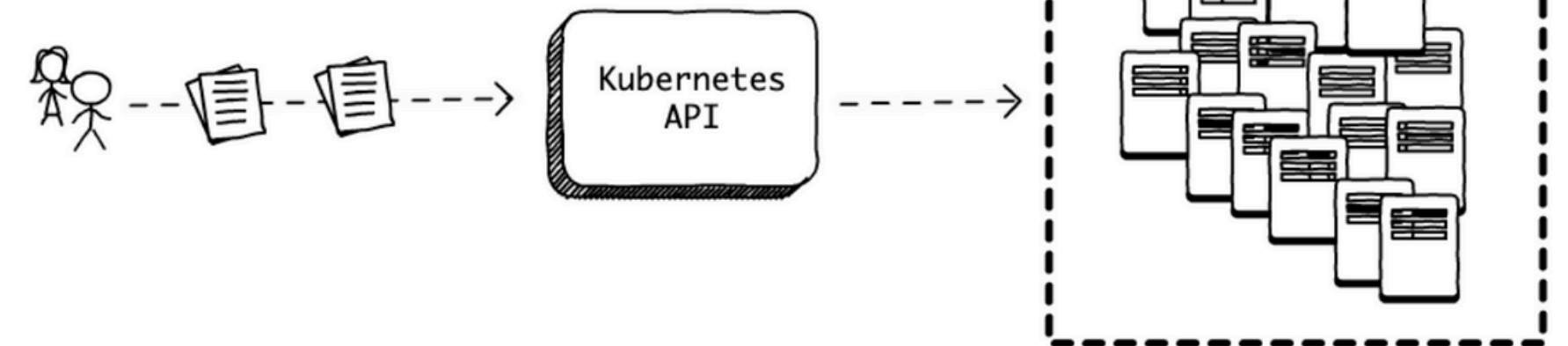
- You don't specify where to run your app—just what you want (e.g., "2 replicas, 2 CPU cores, 5 GB RAM").
- Kubernetes handles deployment, resource allocation, restarts on failure, and isolation between applications.



Kubernetes – Key Features

What Makes Kubernetes Powerful?

- Declarative API
 - Define desired state; Kubernetes maintains it automatically.
- Resource Management
 - Ensures applications stay within their defined limits.
 - Prevents one app from impacting others on shared infrastructure.
- Built-in Capabilities
 - Self-healing: Automatically restarts crashed apps.
 - Load balancing: Distributes traffic across replicas.
 - Networking: Exposes services internally or externally.
 - Configuration and Secrets Management: Secure and flexible handling.
 - Scheduled and One-off Tasks: Run jobs on demand or via cron-style schedules.

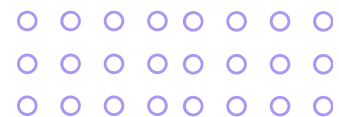


Deploying Our First Application

Let's deploy a simple app (nginx) to see how Kubernetes works in practice.

- Kubernetes is Declarative
 - We describe what we want, not how to do it.
 - This is done using a YAML manifest file

```
nginx.yaml x
k8s-fundamentals > 001-first-deploy > nginx.yaml > {} spec > [ ] containers > {} 0
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: nginx
5  spec:
6    containers:
7      - name: nginx-container
8        image: nginx
9
```



Deploying Our First Application

Deployment Command

Using the CLI tool kubectl, we apply our manifest:

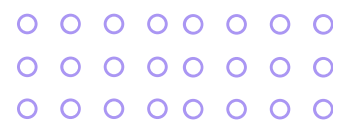
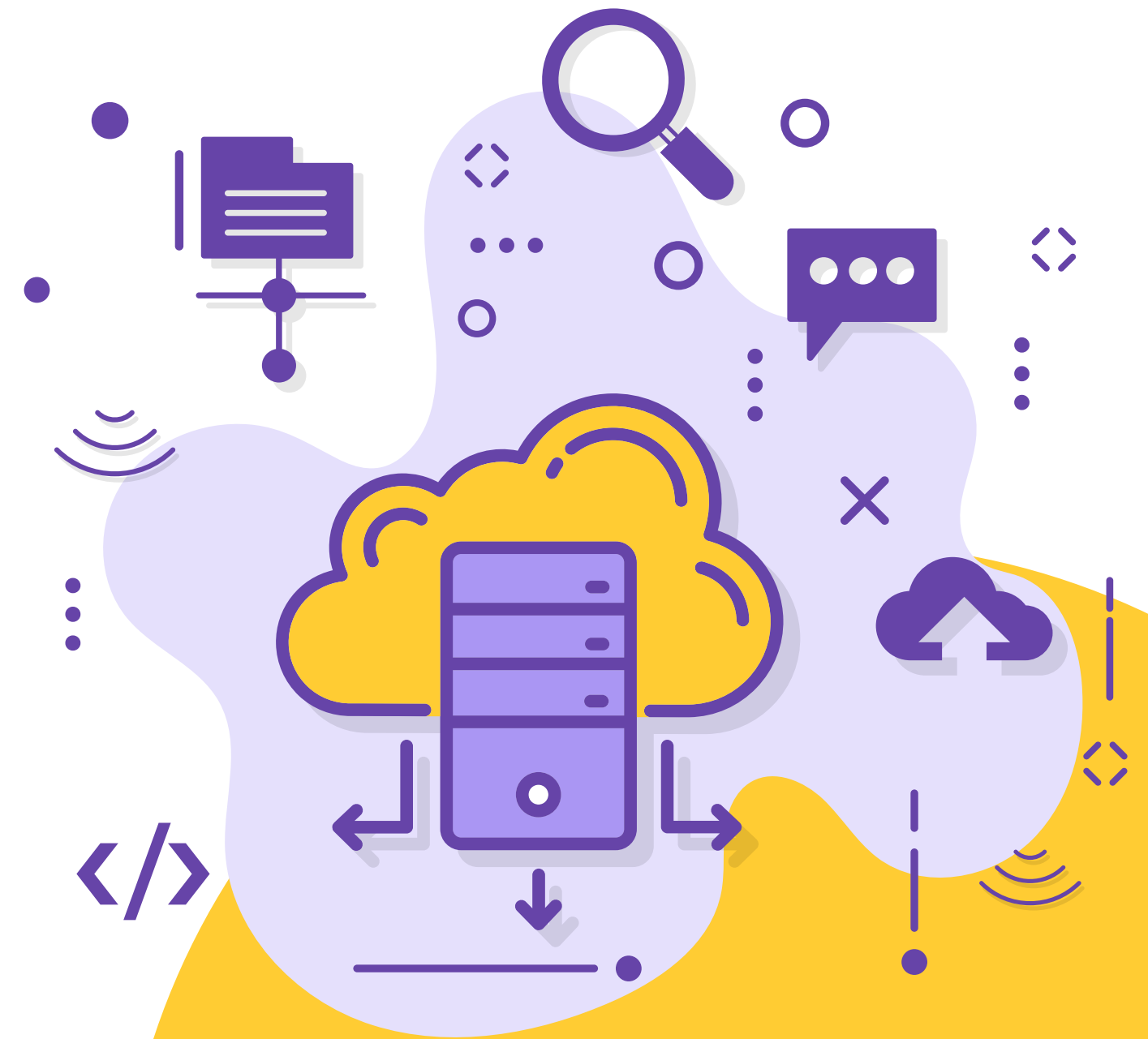
- `kubectl apply -f nginx.yaml`

Result

- This tells Kubernetes to create and run an nginx pod (containerized app).
 - Response: *pod/nginx created*

Checking if Nginx is Running

- Monitor Pod Status
 - Run this command to check the pod:
 - `kubectl get pods`
 - Status will first show as ContainerCreating while the image downloads.
 - Once ready, the status should be Running.



Accessing the Application

Viewing the Nginx Welcome Page

- Port Forwarding
 - To access the app from your browser, use:
 - `kubectl port-forward --address 0.0.0.0 nginx 3000:80`
 - Forwards local port 3000 to container port 80.
 - This makes nginx available at `http://localhost:3000`.
 - Another way of port-forward:
 - `kubectl port-forward nginx 3000:80`

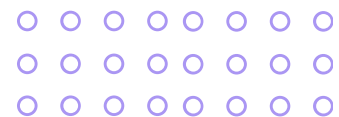


Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

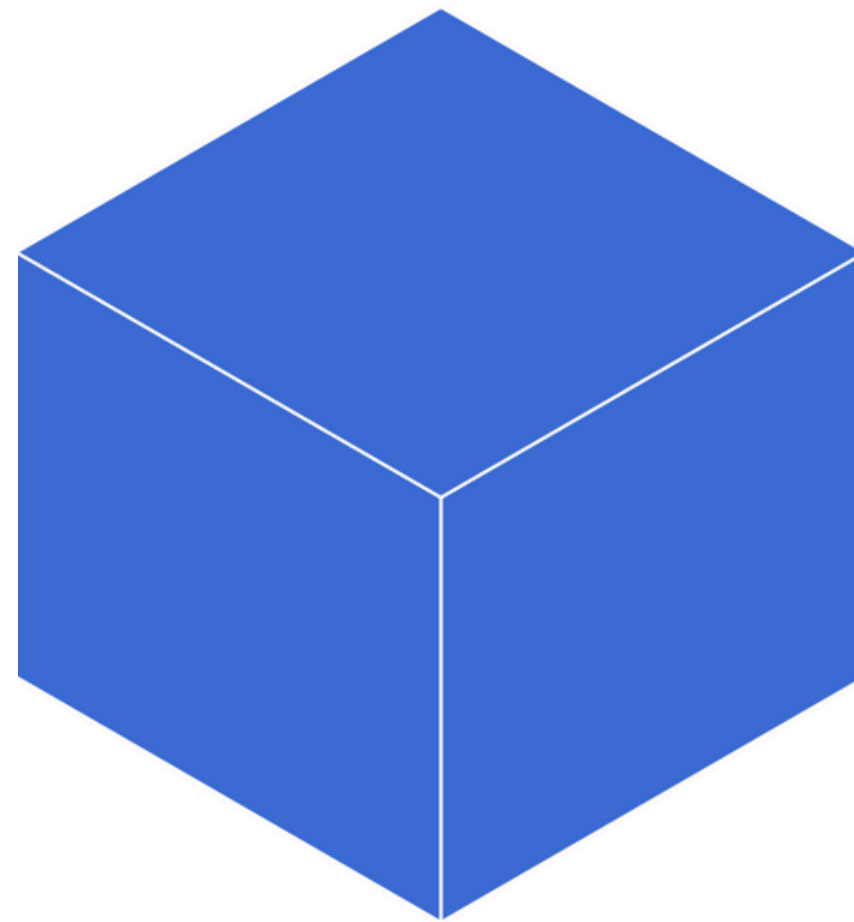
Thank you for using nginx.



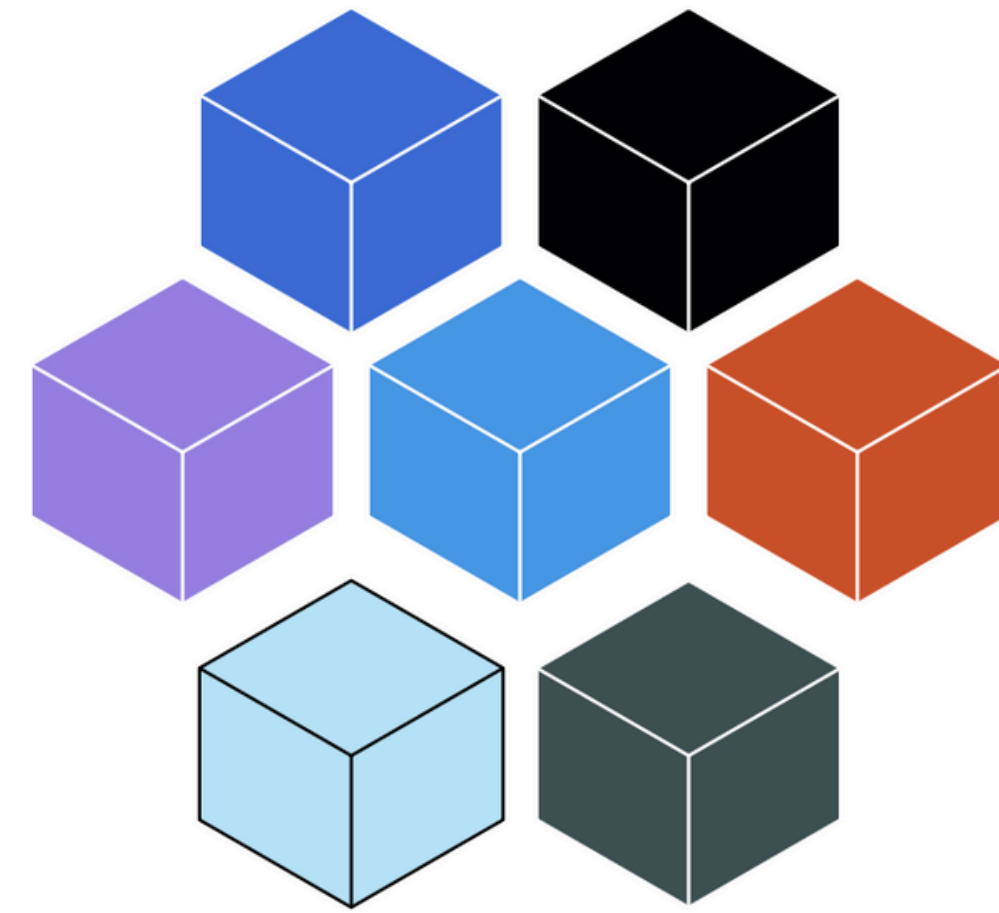
Kubernetes Cluster

When we say “cluster,” we are talking about the two main components of Kubernetes:

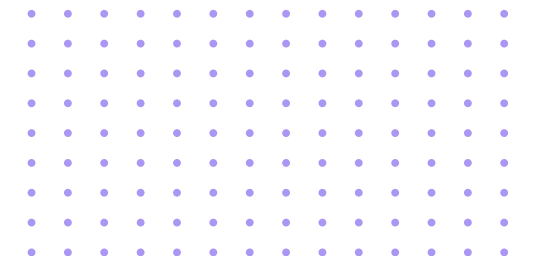
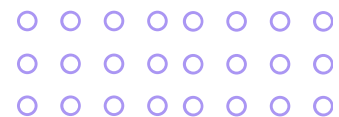
- The master node
- Worker nodes



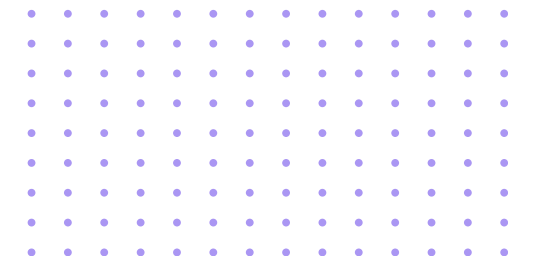
Master Node



Worker Node



Understanding Worker Nodes

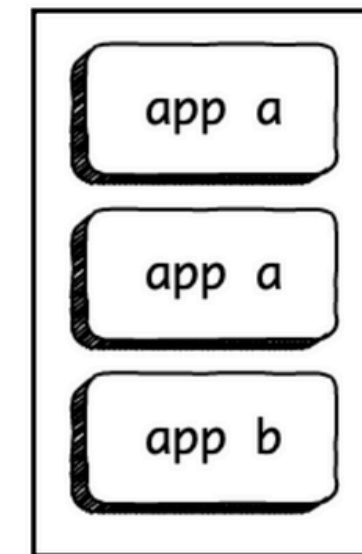


What Are Worker Nodes?

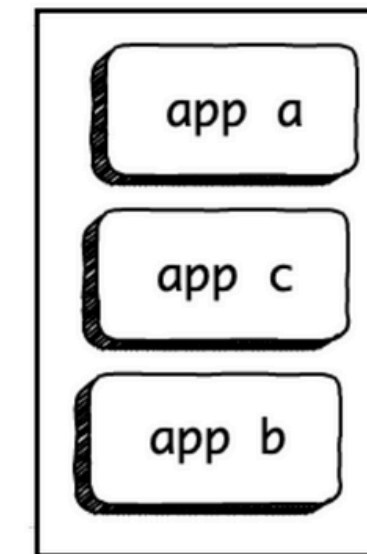
Worker nodes are the machines (virtual or physical) in a Kubernetes cluster that actually run your application workloads—such as containers and pods.

- Each node includes the necessary components to run containers, including a container runtime (like containerd or Docker), a kubelet agent, and a network proxy.
- The Kubernetes control plane (not the worker) tells these nodes what to do.

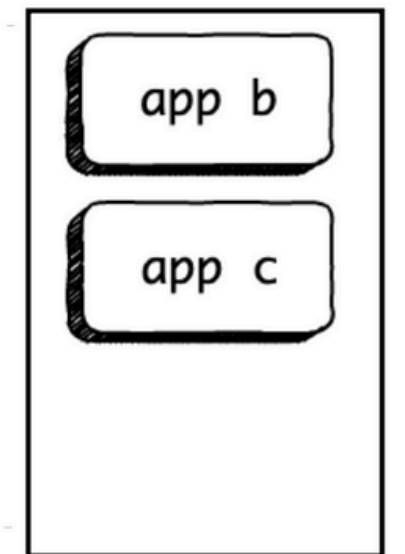
Worker Node 1



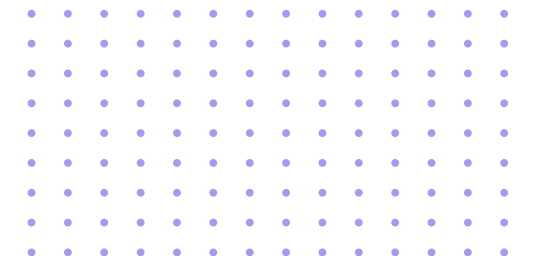
Worker Node 2



Worker Node 3

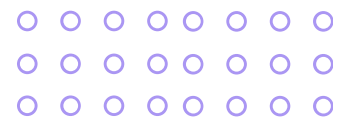


Understanding Worker Nodes

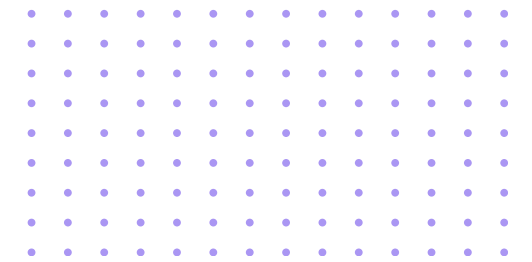


Key Concepts:

- You Rarely Interact With Them Directly
 - Developers and operators don't need to SSH into nodes or manually manage them.
 - Kubernetes manages deployment, scaling, and health checks automatically.
 - This abstraction makes nodes easily replaceable if they fail.
- Multiple Applications per Node
 - A single worker node can host multiple applications (pods) at the same time, as long as resource limits (CPU, memory) are respected.
 - Kubernetes schedules pods efficiently across available nodes.
- One Application, Many Replicas
 - An application can be replicated for scalability or high availability.
 - These replicas are distributed across multiple nodes to prevent a single point of failure.



Understanding Master Node (Control Plane)

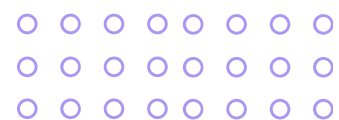


What Is the Master Node?

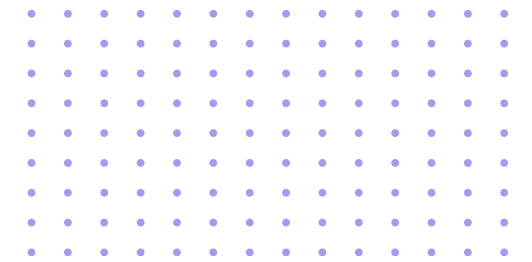
The master node, also known as the control plane, is the central brain of a Kubernetes cluster. It's responsible for orchestrating and managing everything that happens in the cluster.

How It Works

- You Declare the Desired State
 - You describe what you want (e.g., "I need 3 nginx pods") in a YAML manifest file.
 - You then send this file to Kubernetes using tools like *kubectl*.
- The Control Plane Takes Over
 - The master node receives your request and decides how to make it happen.
 - It doesn't just execute commands—it thinks:
 - "What's the current situation, and what should it look like?"
- The Reconciliation Loop
 - Kubernetes continuously compares the current state of the cluster with the desired state you defined.
 - If something goes out of sync (e.g., a pod crashes or is deleted), Kubernetes automatically takes action to restore the desired state.

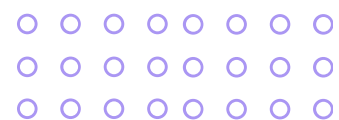
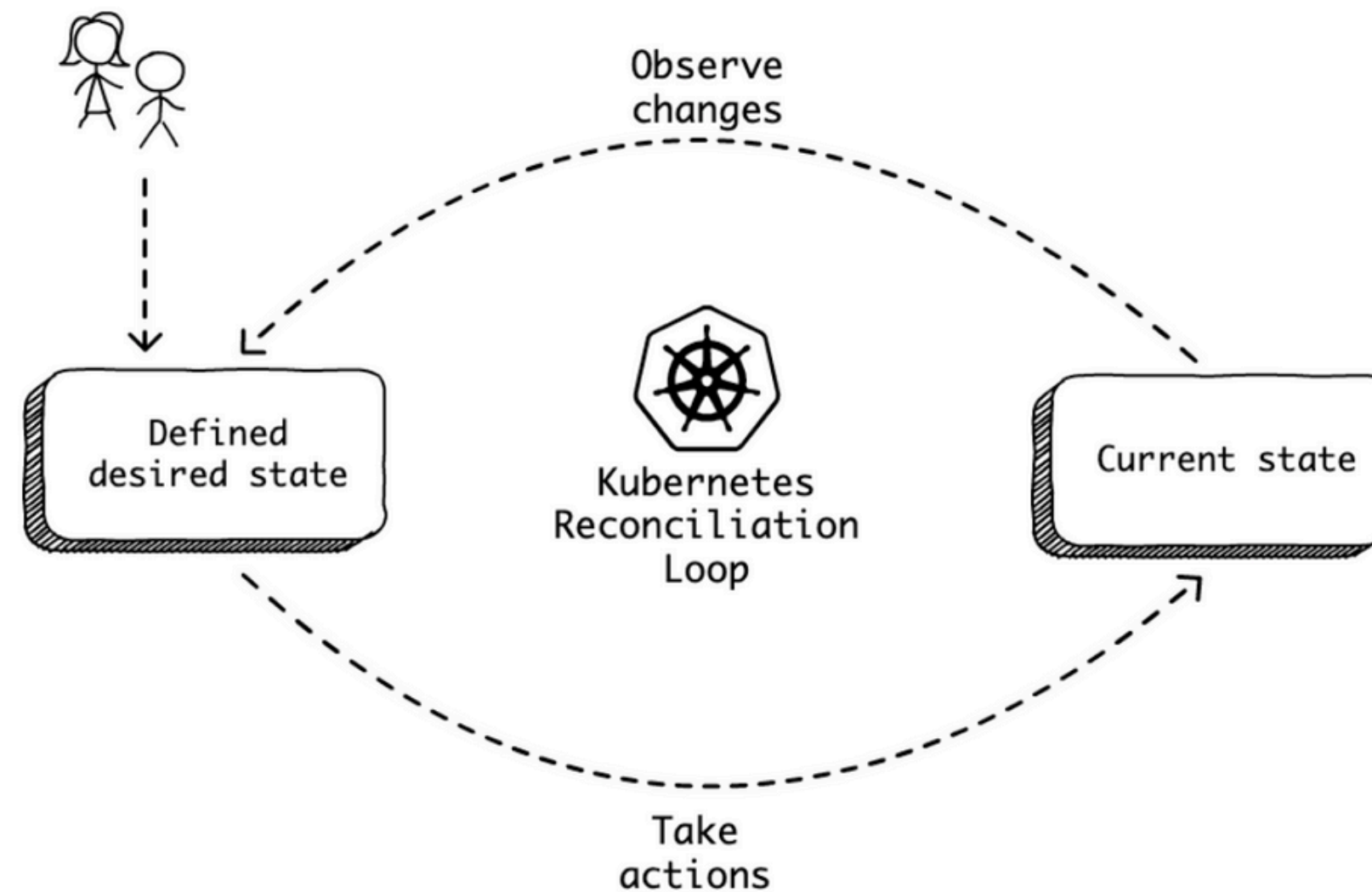


Understanding Master Node (Control Plane)



How It Works

- The Reconciliation Loop
 - Kubernetes continuously compares the current state of the cluster with the desired state you defined.
 - If something goes out of sync (e.g., a pod crashes or is deleted), Kubernetes automatically takes action to restore the desired state.



Introduction to *kubectl*

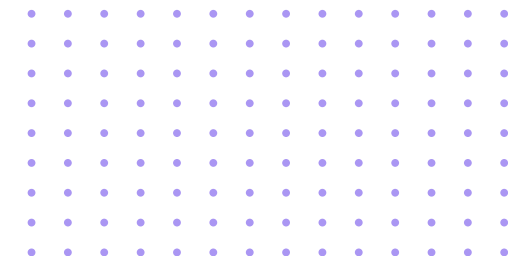
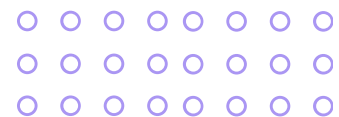
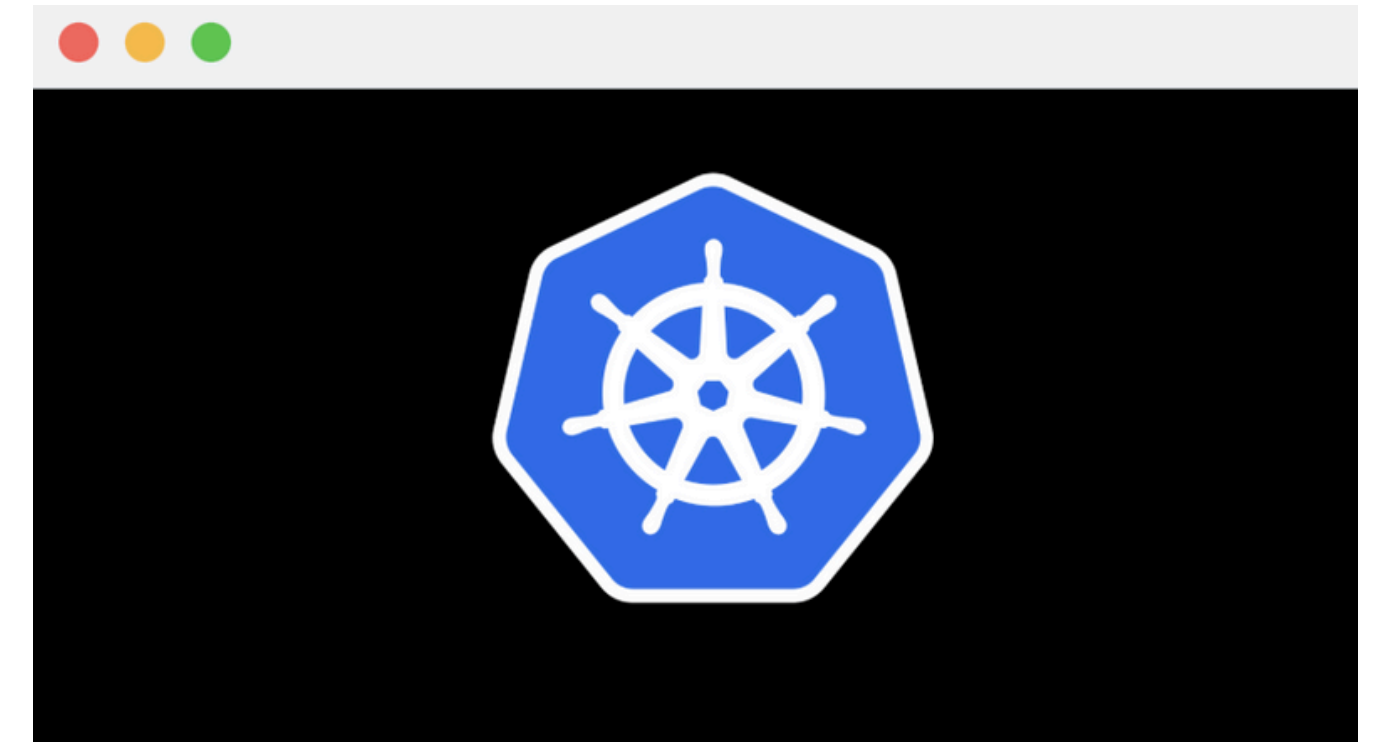
kubectl is the Kubernetes CLI, and it will be our main interface with Kubernetes for most of the time. It is the tool we use to send information to our cluster and to get information from it.

There are two main ways to use *kubectl*:

- Declarative way
- Imperative way

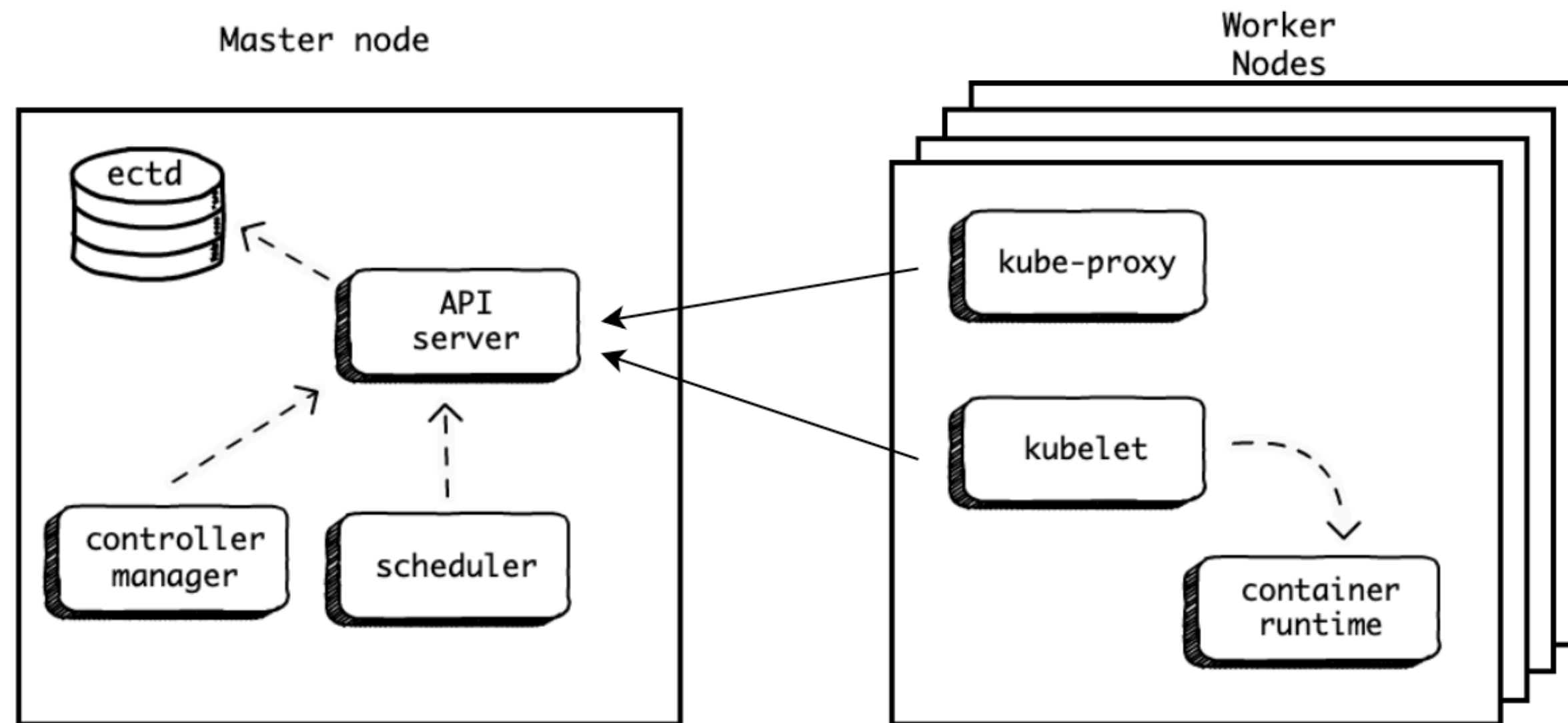
We use it declaratively when we create a manifest that defines what we want and just send that to our cluster that will then find a way to make that happen.

We use it imperatively when we give our cluster specific orders telling it how to do what we want. The imperative usage of *kubectl* is fine for development or to quickly test things out.

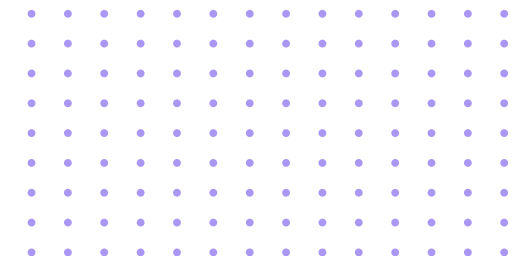


Kubernetes Architecture

There are multiple subcomponents running in the master and worker nodes.



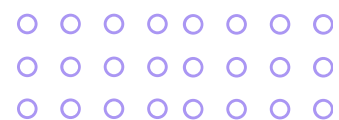
Kubernetes Architecture – Control Plane (Master Node)



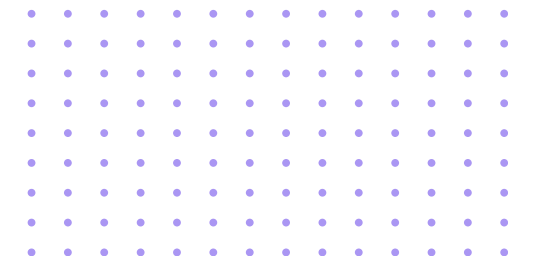
The control plane (often called the master node) is responsible for managing the entire Kubernetes cluster. It handles scheduling, decision-making, and maintaining the desired state.

- API Server
 - Acts as the entry point to the cluster. All commands (e.g., via kubectl) go through the API server.
- etcd
 - A key-value store that holds the cluster state—what's running, configurations, and desired specifications.
- Scheduler
 - Chooses which worker node should run a new pod, based on resources, constraints, and rules.
- Controller Manager
 - Watches the state of the cluster and takes corrective actions if the actual state doesn't match the desired state (e.g., restarts failed pods).

These components constantly communicate to keep the cluster stable, healthy, and in sync with your specifications.



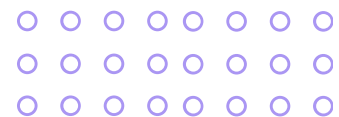
Kubernetes Architecture – Worker Nodes



Worker nodes are the machines that host and execute your application containers. Each worker has several key components:

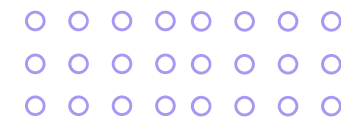
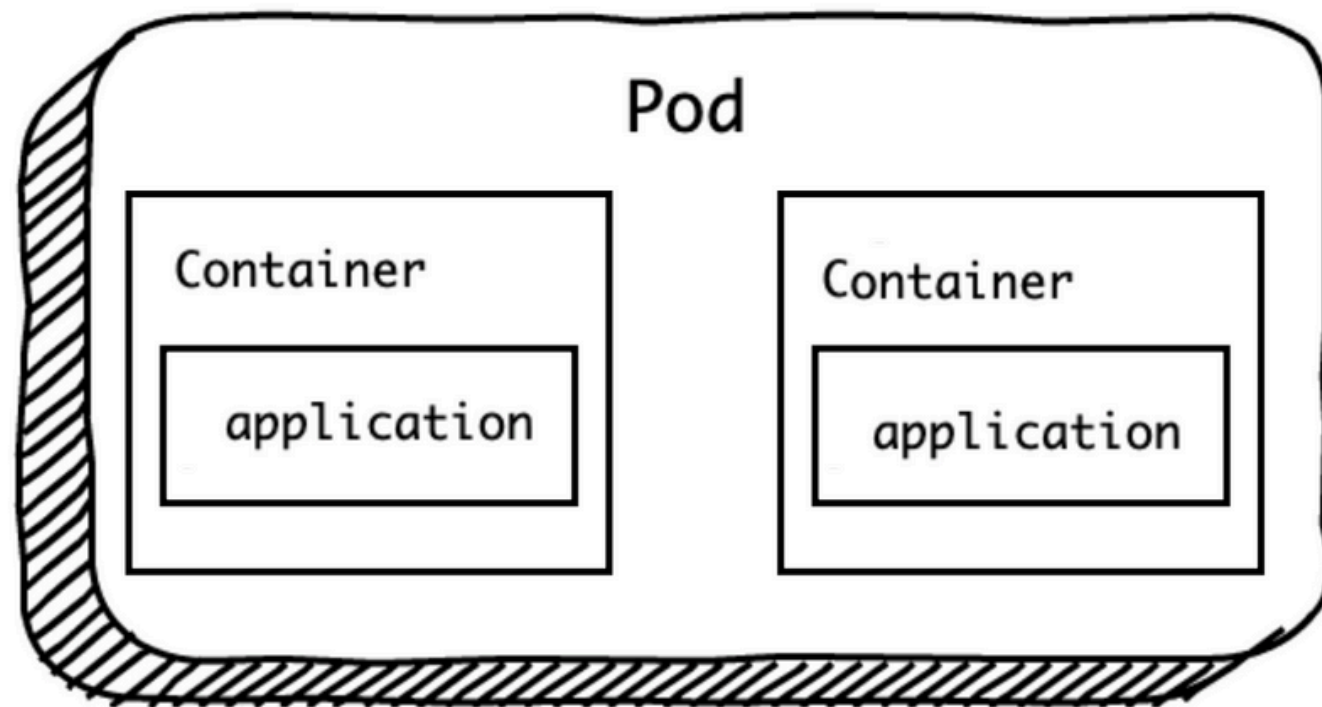
- Kubelet
 - Agent that talks to the API server and manages pods on the node. It ensures containers are running as instructed.
- Container Runtime
 - The engine (e.g., containerd, Docker) that actually runs the containers.
- Kube-Proxy
 - Handles networking for the node. It manages internal communication and load balancing for services.

Multiple applications can run on the same worker node, and each application can be replicated across several nodes for high availability and scalability.

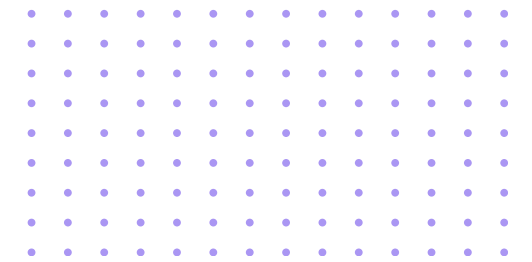


Introduction to pods

Pods are where our applications run. It's the basic building block and the atomic unit of scheduling in Kubernetes. Each pod will include one or more containers and every time we run a container in Kubernetes, it will be inside a pod.



Pods: The Atomic Unit of Scheduling in Kubernetes

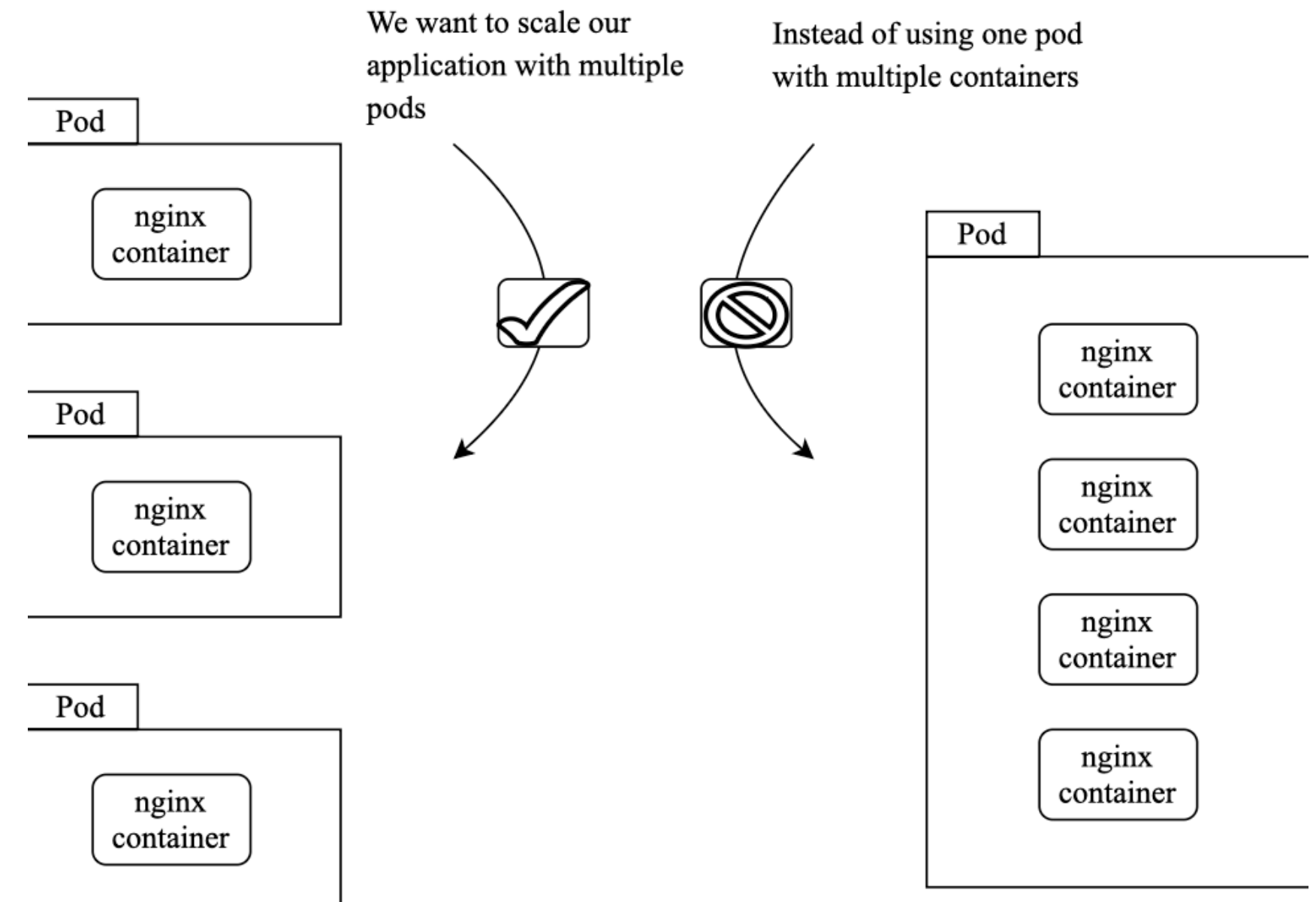


In Kubernetes, the pod is the smallest and most fundamental unit of scheduling. This means that when Kubernetes decides where and how to run an application, it schedules pods, not individual containers.

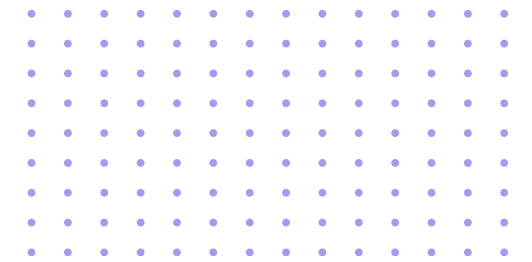
- *A pod typically contains one container, but it can host multiple tightly coupled containers that need to share resources like storage and networking.*

For example, if we want to run 10 replicas of an application, we don't create one pod with 10 containers. Instead, we create 10 pods, each running one instance of the container. This ensures:

- isolation
- scalability
- flexibility.

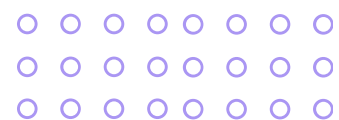


Understanding a Basic Pod Manifest



Kubernetes manifest files are written in YAML and describe what we want the cluster to create.

```
nginx.yaml X
k8s-fundamentals > 001-first-deploy > nginx.yaml > {} spec > [ ] containers > {} 0
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: nginx
5  spec:
6    containers:
7      - name: nginx-container
8        image: nginx
9
```



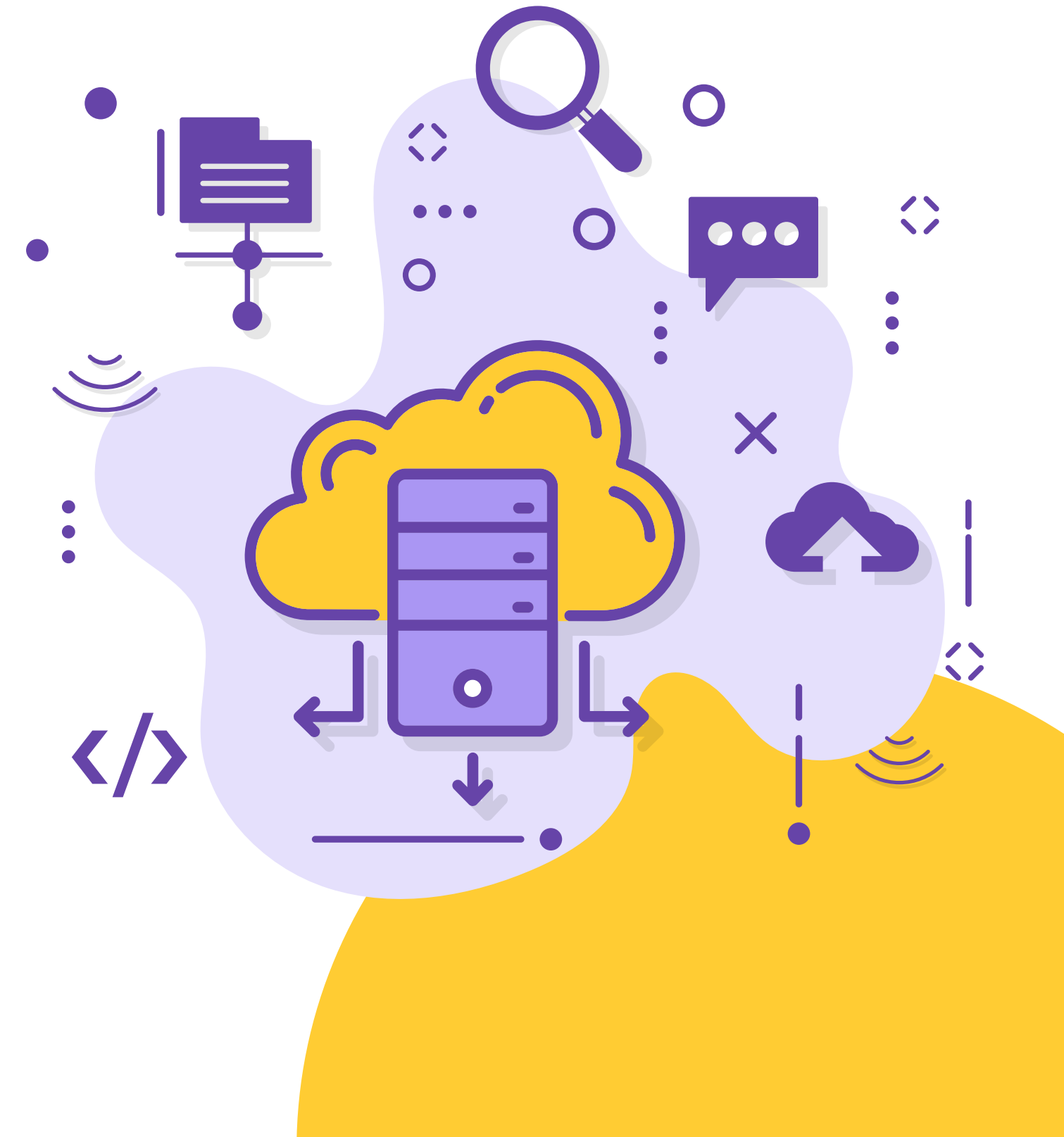
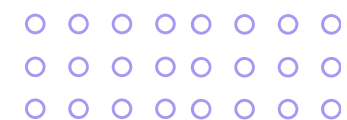
Understanding a Basic Pod Manifest

In the previous manifest we have:

- The *kind* key specifies the type of resource being defined — in this case, a Pod.
- Under *metadata*, we provide a unique name for the pod, e.g., nginx.
- The *spec* section defines what to run inside the pod.

```
5 spec:  
6   containers:  
7     - name: nginx-container  
8       image: nginx  
9
```

- This tells Kubernetes to run a container named nginx-container using the nginx Docker image.
- Most Kubernetes manifests follow this structure. While many other keys are available, this is the minimum needed to run a pod.



Where Does the Container Image Come From?

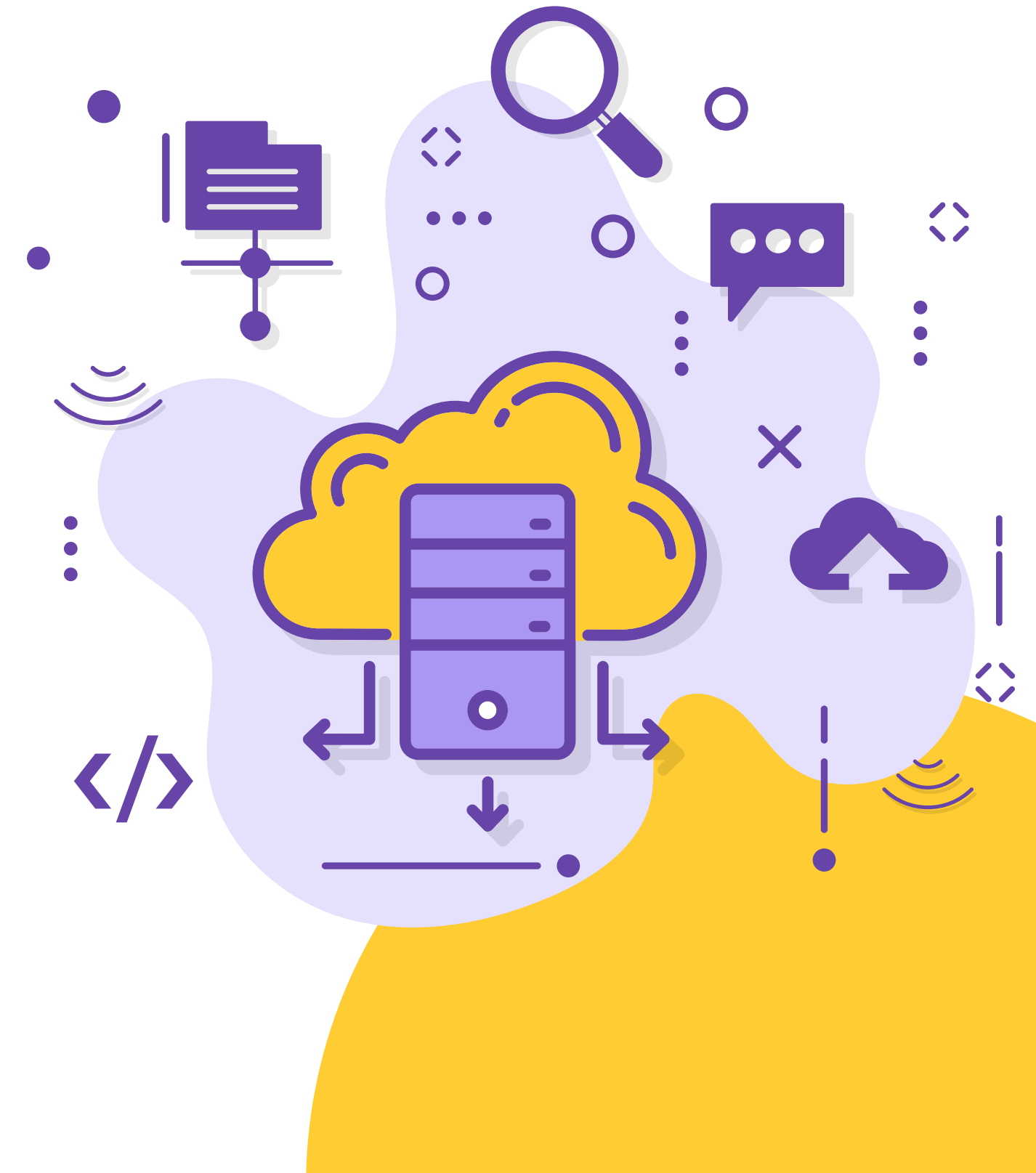
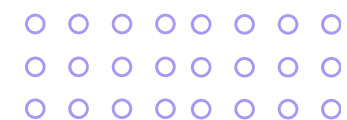
In our manifest, we declare the image name (nginx) but don't specify a location. If we don't specify, Kubernetes it will:

- Automatically pulls the image from a Docker registry.
- If no registry is defined, it defaults to Docker Hub.

Just like code is stored in GitHub or GitLab, container images are stored in image registries, such as:

- Docker Hub
- Google Container Registry
- Amazon ECR
- Private registries

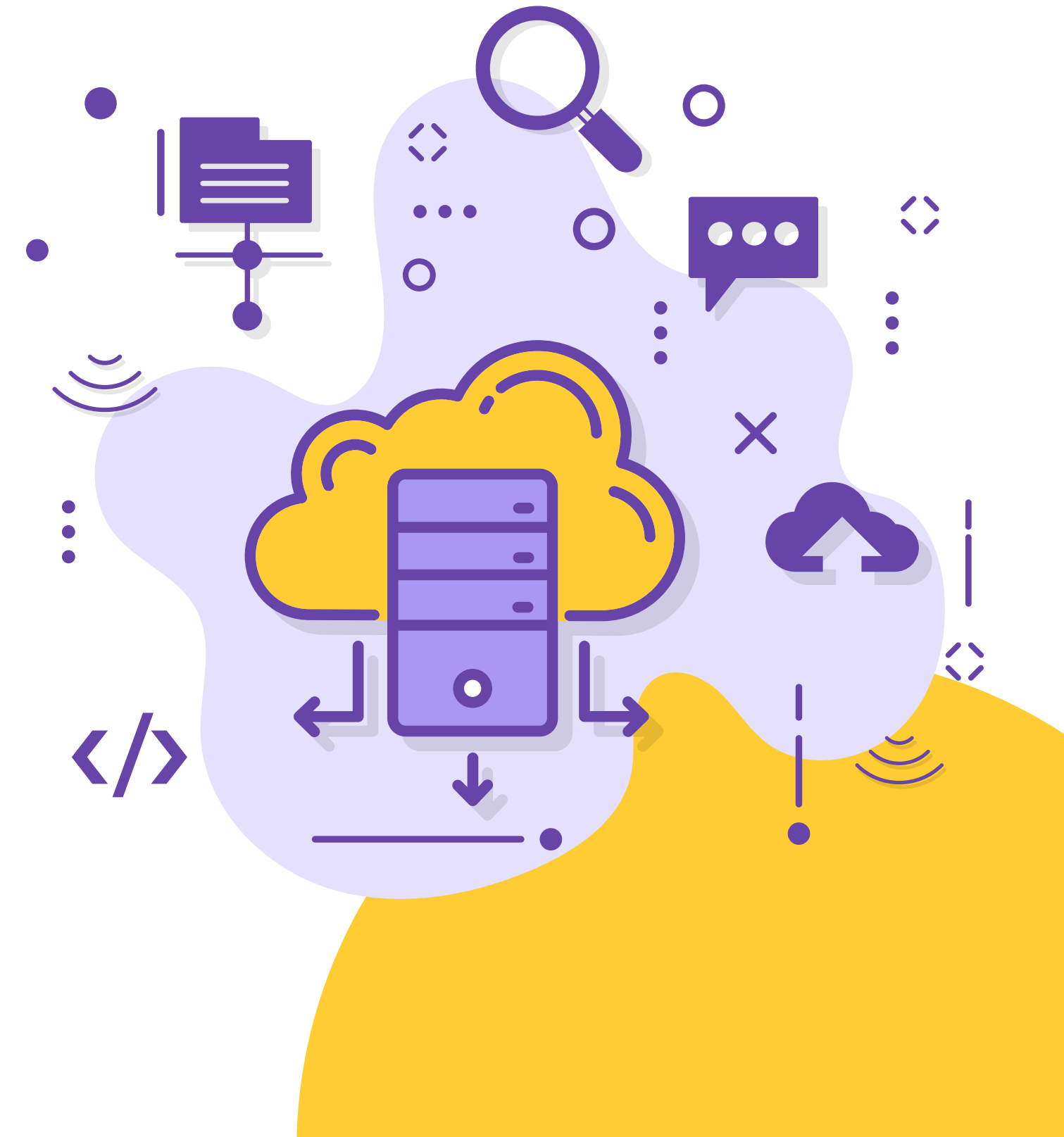
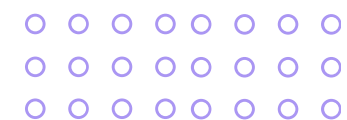
Important: you can always specify a full image path if you're using a custom or private registry.



Where Does the Container Image Come From?

If we want to be more explicit, we can change our manifest to use the full image path:

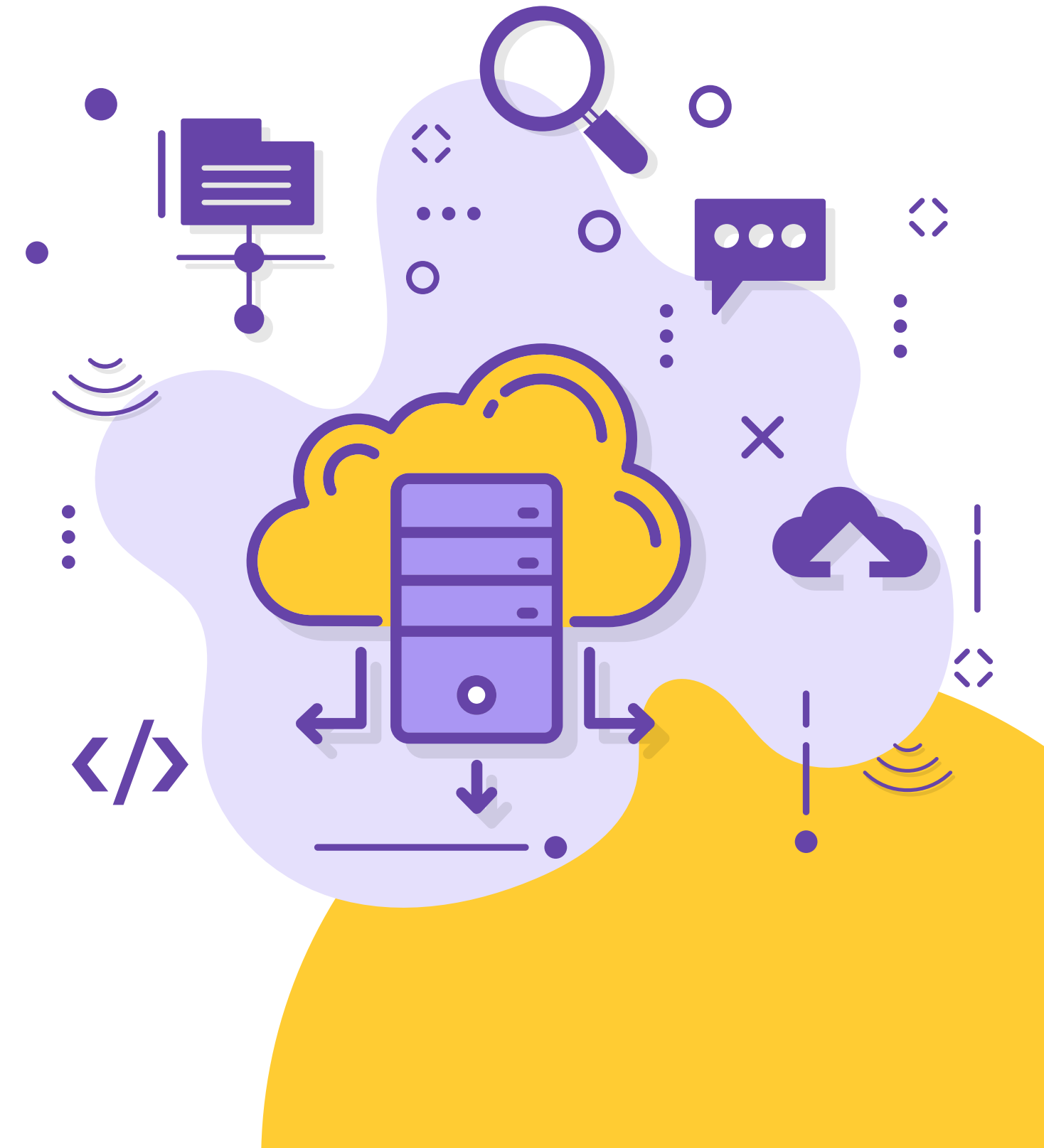
```
nginx.yaml X
k8s-fundamentals > 001-first-deploy > nginx.yaml > {} spec > [ ] containers > {} 0
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: nginx
5  spec:
6    containers:
7      - name: nginx-container
8        image: registry.hub.docker.com/library/nginx
9
```



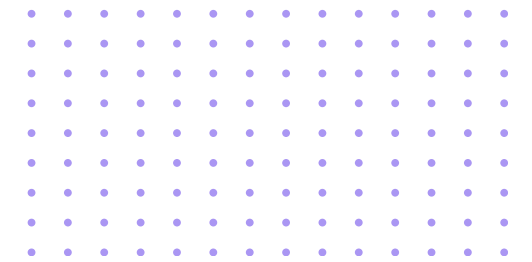
Where Does the Container Image Come From?

Or if you are using a different service to store images like ECR, we can define its full path as well:

```
nginx.yaml x
k8s-fundamentals > 001-first-deploy > nginx.yaml > {} spec > [ ] containers > {} 0
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: nginx
5  spec:
6    containers:
7      - name: nginx-container
8        image: public.ecr.aws/nginx/nginx
9
```



Pods vs. Containers: What's the Difference?



At first, pods may seem just like containers, but there's a key distinction:

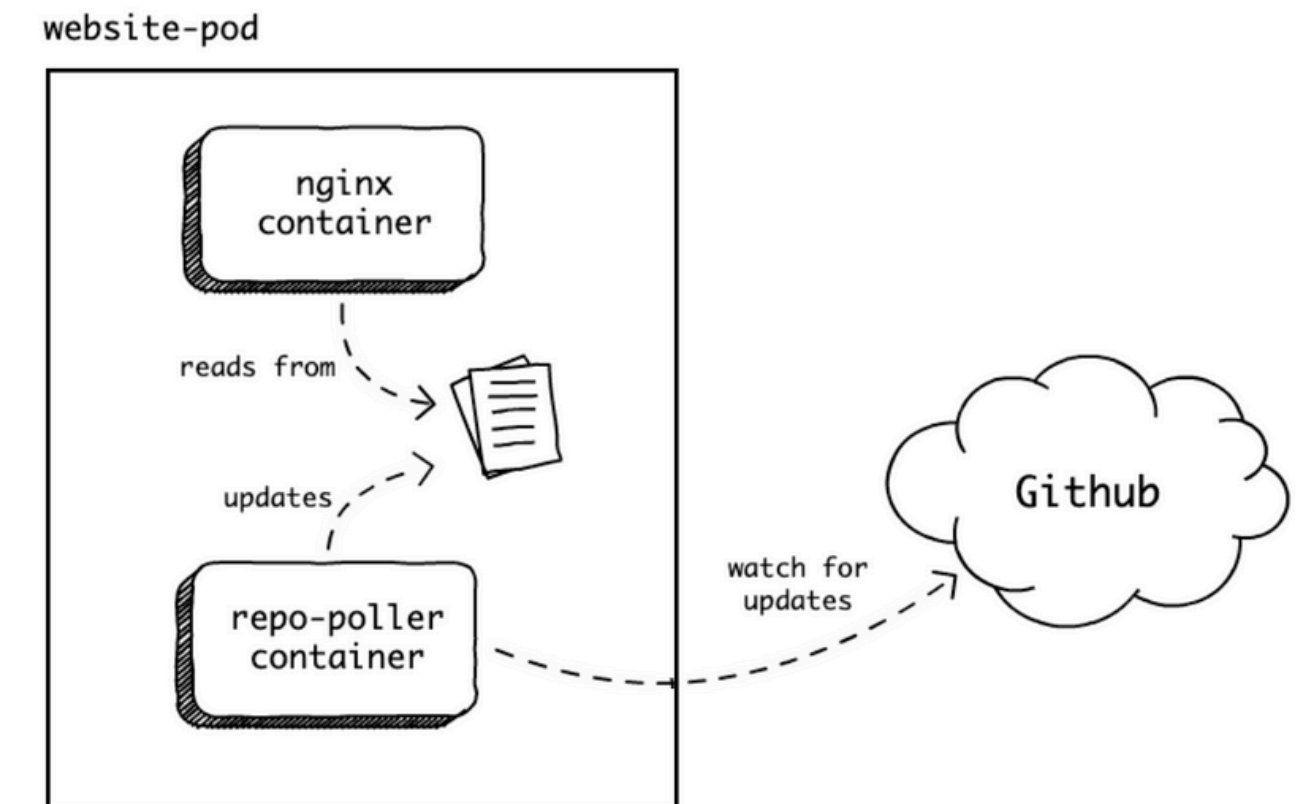
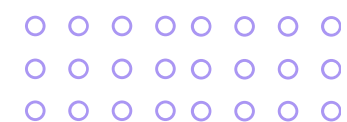
- A pod is a wrapper that can run one or more containers together as a unit.
- Containers in the same pod share networking and storage, and are scheduled together on the same node.

Think of a pod as:

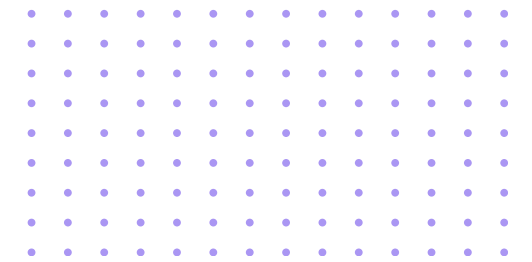
A logical group of tightly coupled containers that cooperate to perform a task.

- In most cases, especially for simple apps, we run one container per pod.
- In advanced scenarios, we can run helper containers in the same pod.
 - Example: An nginx container serving a website + a sidecar container that updates the site content from GitHub.

A container runs your app, while a pod organizes and manages one or more related containers.



Interacting with Running Pods – Accessing and Logs



Deploy and Access a Pod

- Run the pod using:
 - `kubectl apply -f nginx.yaml`
- Once the pod is in Running state, access it locally:
 - `kubectl port-forward --address 0.0.0.0 nginx 3001:80`
 - Running it in the background:
 - `nohup kubectl port-forward --address 0.0.0.0 nginx 3001:80 > /dev/null 2>&1 &`

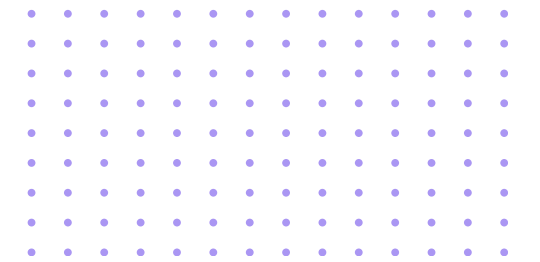
Monitor Logs in Real Time

- View real-time application logs:
 - `kubectl logs --follow nginx`
 - `nginx`: Name of the pod
 - `--follow`: Streams live logs

Logs show real-time HTTP requests/responses as users interact with your app.



Executing Commands Inside a Pod



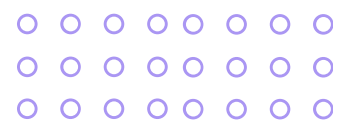
Inspect and Modify Containers

- Run a command inside the container:
 - `kubectl exec nginx -- ls`
- Start an interactive shell:
 - `kubectl exec -it nginx -- bash`
 - Example: Modify the HTML file being served:
 - `echo "Hello Kubernetes!" > /usr/share/nginx/html/index.html`

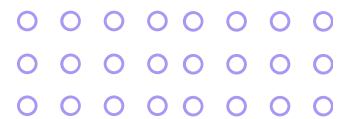
Killing a Pod

- Delete the pod by name:
 - `kubectl delete pod nginx`
- Or delete using the same manifest:
 - `kubectl delete -f nginx.yaml`

After deletion, running `kubectl get pods` should show that nginx is gone.



Questions?



Exercise - Running our own pod

Run Apache, which is another web server that works similarly. Here's a list of things you'll need to do:

- Write a pod manifest file to run the apache container (the docker image is called *httpd*).
- Use *kubectl* to apply this manifest and see it running.
- Use *kubectl port-forward* to send requests from the *port 3000* to the container's *port 80*.

*Note: you need to add the **--address 0.0.0.0** to the *kubectl port-forward* to run it on the platform.*

- Confirm that you can see Apache's default page saying "It works!" when you access the APP URL in the widget below.
- Enter the container, and change the contents of the file */usr/local/apache2/htdocs/index.html*, so when you refresh the page you can see your changes.

