🚀 Migration Guide: From Docker (VM) to Kubernetes (K8s)

This document explains how the sandbox platform was migrated from a **VM-based Docker architecture** to a **Kubernetes-native architecture**.

If you were previously using Docker directly inside a VM, this guide explains:

- What changed
- Why it changed
- How Kubernetes replaces Docker logic
- What you must configure
- How to test locally using Minikube

Previous Architecture (VM + Docker)

How It Worked

```
User Request
     ↓
Express API
     ↓
DockerManager (dockerode)
     ↓
Docker Daemon (VM)
     ↓
Container Created
```

Key Characteristics

- API directly controlled Docker
- Used `docker.createContainer()`
- Used `docker.exec()` to run code
- Used `docker.stop()` to clean up
- Resource limits set via Docker HostConfig

Docker Resource Limits Example

```
HostConfig: {
  Memory: 512 * 1024 * 1024,
  CpuQuota: 50000,
  NetworkMode: "none",
  PidsLimit: 64
}
```

Limitations

- Only works on single VM
- No cluster-level scaling
- Manual container lifecycle management
- Not cloud-native

New Architecture (Kubernetes Native)

How It Works Now

```
User Request
     ↓
Express API
     ↓
KubernetesManager
     ↓
Kubernetes API Server
     ↓
Pod Created
     ↓
Container Runs Inside Pod
```

Instead of controlling Docker directly, the API now talks to the **Kubernetes API Server**.

🚡 What Changed in Code

Removed

- dockerode
- docker.buildImage()
- docker.createContainer()
- docker.exec()
- docker.stop()

Added

- `@kubernetes/client-node`
- `createNamespacedPod()`
- `exec()` via Kubernetes client
- `deleteNamespacedPod()`

🏩 Feature Comparison

Feature	VM + Docker	Kubernetes
Container Creation	docker.createContainer	createNamespacedPod
Code Execution	docker.exec	k8s exec
Cleanup	docker.remove	deleteNamespacedPod
Resource Limits	HostConfig	Pod resources.limits
Scaling	Manual	Automatic via cluster
Scheduling	Single machine	Multi-node cluster

Kubernetes Pod Configuration

Example Pod Spec:

```
const podManifest = {
  metadata: { name: podName },
  spec: {
    restartPolicy: "Never",
    containers: [
      {
        name: "sandbox",
        image: "custom-node-sandbox:latest",
        resources: {
          limits: {
            memory: "512Mi",
            cpu: "500m"
          }
        }
      }
    ]
  }
};
```

Resource Limits Now Set As:

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

🚚 Security Improvements

Kubernetes provides:

- Pod-level isolation
- Resource quotas
- Namespace isolation
- RBAC permissions
- Network policies

This is more secure than mounting Docker socket.

🛕 Testing with Minikube (Local Kubernetes)

Step 1: Start Minikube

```
minikube start
```

Step 2: Build Image Inside Minikube

```
eval $(minikube docker-env)
docker build -t custom-node-sandbox:latest .
```

Step 3: Run API

```
npm run dev
```

Step 4: Verify Pods

```
kubectl get pods
```

🚨 Important Production Notes

Wait for Pod to Be Ready Before Exec

Pods take time to start.

Add RBAC Role

Your API must have permission to: - create pods - delete pods - exec into pods

Do Not Use Docker-in-Docker

Avoid mounting `/var/run/docker.sock`.

🏗️ New System Architecture

```
Client
   ↓
API Deployment
   ↓
Kubernetes API
   ↓
Sandbox Pods
```

This is cloud-native, scalable, and production-ready.

🚀 🛻 Why This Is Better

- Works in cloud environments
- Scales across multiple nodes
- Self-healing
- Resource management at cluster level
- Production SaaS ready

Final Summary

You migrated from:

❌ VM-controlled Docker containers

To:

✅ Kubernetes-managed Pods

Now your platform is:

- Cloud-native
- Scalable
- Multi-node ready
- Infrastructure-grade

🛳️ Next Possible Improvements

- Add Pod readiness checks
- Add execution timeouts
- Use Kubernetes Jobs for short executions
- Add namespace-per-user isolation
- Add auto-cleanup controller

Your sandbox platform is now Kubernetes-native 🚯

You’ve officially moved from "container user" to "container orchestrator client."