The Kind of Kubernetes: Using Docker Desktop for Reproducible Kubernetes Clusters

Namdak Tonpa

July 5, 2026

Contents

1 Introduction

Kind (Kubernetes IN Docker) constitutes a precise instrument for instantiating standards-compliant Kubernetes clusters wherein each node is realized as a Docker container.

For the development of ERP/1, the combination of Kind and Docker Desktop yields a controlled environment that closely approximates production conditions on both Linux and macOS systems. The present document provides a rigorous exposition of the configuration and utilization of this system.

Kind was introduced in 2018 [1] and employs the official `kubeadm` bootstrap mechanism, ensuring semantic equivalence with production Kubernetes deployments at the level of the control plane and worker node behavior.

1.1 Architectural Analysis

The system operates by constructing a graph of Docker containers, each executing a full `kubelet` and associated Kubernetes components. Cluster initialization follows the canonical `kubeadm init` and `kubeadm join` protocol.

This design confers two principal advantages: (i) exact reproducibility of Kubernetes semantics, and (ii) minimal deviation from the production execution model. Consequently, scheduling, networking, storage, and security primitives behave in a manner that is formally transferable to bare-metal or cloud-orchestrated clusters.

2 Kind of Kubernetes

2.1 Prerequisites

The following conditions must be satisfied:

- Docker Desktop (version ≥ 4.38) with containerd image store enabled.
- Minimum 8 GB RAM (16 GB or higher recommended for multi-node configurations hosting ERP/1 workloads).
- Hardware-assisted virtualization support.
- Working installation of `kubectl` and Helm.

2.2 Installation

2.2.1 On macOS

```
brew install kind
```

2.2.2 On Linux

Download the binary or use package managers:

```
curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.32.0/kind-linux-amd64
chmod +x ./kind
sudo mv ./kind /usr/local/bin/kind
```

[1] The project Kind was started by Ben Elder (a Kubernetes maintainer) and quickly became very popular in the community because it is lightweight, fast, and great for local development and testing. By 2019–2020, it had gained massive adoption and is now one of the most widely used tools for local Kubernetes clusters.

2.3 Using Kind with Docker Desktop

Docker Desktop implements Kind as a native provisioner. For reproducible deployments, the command-line interface with an explicit configuration file is preferred. A recommended configuration for ERP/1 workloads is as follows:

1. Open Docker Desktop Settings → Kubernetes
2. Enable Kubernetes and select **Kind** as the provisioner
3. Create a multi-node cluster

Alternatively, use the CLI for more control:

2.4 Cluster Provisioning via Docker Desktop and CLI

Docker Desktop implements Kind as a native provisioner. For reproducible deployments, the command-line interface with an explicit configuration file is preferred. A recommended configuration for ERP/1 workloads is as follows:

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
name: erp1-local
nodes:
  - role: control-plane
  - role: worker
  - role: worker
kubeadmConfigPatches:
  - |
    kind: ClusterConfiguration
    networking:
      podSubnet: "10.244.0.0/16"
```

Cluster creation command:

```
kind create cluster --config kind-config.yaml --name erp1-local
```

2.5 Recommended kind-config.yaml for ERP/1

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
name: erp1-prod-like
nodes:
  - role: control-plane
  - role: worker
  - role: worker
kubeadmConfigPatches:
  - |
```

```
    kind: ClusterConfiguration
    networking:
      podSubnet: "10.244.0.0/16"
containerdConfigPatches:
  - |-
    [plugins."io.containerd.grpc.v1.cri".registry.mirrors]
      [plugins."io.containerd.grpc.v1.cri".registry.mirrors."docker.io"]
        endpoint = ["https://registry-1.docker.io"]
```

2.6 Deployment of ERP/1 Components

Integration with the repository tooling proceeds as:

```
./prereq.sh
kind create cluster --config kind-config.yaml
./deploy.sh
```

The multi-node topology permits accurate modeling of pod scheduling, inter-service communication, and resource isolation.

2.7 Node Specialization

Critical services such as `ns-dns` and `ca-pki` benefit from explicit node dedication. Labeling and affinity rules are applied as follows:

```
kubectl label node <worker-name> dedicated=dns
kubectl label node <worker-name> dedicated=pki
```

Subsequent Deployments should specify `nodeSelector` or `nodeAffinity` fields to enforce placement constraints. This practice mirrors production hardening strategies and facilitates early detection of resource contention.

Key advantages for ERP/1:

- Multi-node setup mimics production topology
- Fast iteration cycle
- Full support for Helm, Operators, and complex dependencies (ns-dns, ca-pki, etc.)
- Easy node dedication using labels and affinity

2.8 Best Practices

- Maintain strict version alignment between local Kind clusters and target production environments.
- Prefer containerd over Docker shim for improved isolation and performance.

- Systematically export manifests via `kubectl get --all-namespaces -o yaml` for GitOps synchronization.
- Employ Kustomize overlays to manage environment-specific variations.
- Continuously monitor resource utilization through `kubectl top` and Docker Desktop metrics.

2.9 Migration to Production Environments

Owing to Kind's use of unmodified Kubernetes components, the transition of manifests to production clusters requires only localized modifications, primarily concerning persistent storage (replacement of `hostPath` with CSI volumes) and ingress/load-balancing configuration.

2.10 Managing Clusters with kind.sh

For convenient management of multiple local clusters alongside Docker Desktop, we provide a custom wrapper script `kind.sh`.

```
#!/bin/bash

set -euo pipefail

NAME="${2:-desktop}"
CLUSTER_NAME="$NAME"

create() {
  echo "Creating KinD cluster: $CLUSTER_NAME"
  kind create cluster --name "$CLUSTER_NAME" --wait 2m
  kind get kubeconfig --name "$CLUSTER_NAME" > /tmp/kc.yaml
  KUBECONFIG=~/.kube/config:/tmp/kc.yaml \
  kubectl config view --flatten > ~/.kube/config.new
  mv ~/.kube/config.new ~/.kube/config
  rm -f /tmp/kc.yaml
  echo "Cluster $CLUSTER_NAME created"
}

delete() {
  echo "Deleting cluster: $CLUSTER_NAME"
  kind delete cluster --name "$CLUSTER_NAME" 2>/dev/null || true
  kubectl config delete-context "kind-$CLUSTER_NAME" 2>/dev/null || true
  kubectl config delete-cluster "$CLUSTER_NAME" 2>/dev/null || true
  kubectl config delete-user "$CLUSTER_NAME" 2>/dev/null || true
  echo "Deleted $CLUSTER_NAME"
}
```

```
list() {
  echo "=== KinD Clusters ==="
  kind get clusters || echo "None"
  echo ""
  echo "=== Kubectl Contexts ==="
  kubectl config get-contexts
}

case "${1:-list}" in
  create) create ;;
  delete) delete ;;
  list) list ;;
  *) echo "Usage: ./kind.sh <create|delete|list> [name]" ;;
esac
```

This script allows clean management of multiple environments:

```
./kind.sh create neo
./kind.sh list
./kind.sh delete neo
```

3 Conclusion

Kind, when orchestrated through Docker Desktop, provides a mathematically clean and operationally efficient platform for local validation of ERP/1 systems. Its fidelity to the Kubernetes specification renders it a rational choice for developers seeking to minimize the semantic gap between development and production.