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

Namdak Tonpa

July 9, 2026

**Abstract**

Local replication of production-grade cloud-native environments is a critical prerequisite for the deterministic validation and continuous delivery of complex enterprise applications. However, existing local development models often suffer from a semantic gap concerning stateful storage persistence, multi-node network scheduling, and dependency resolution reproducibility. This paper presents a formal framework for instantiating reproducible, multi-node Kubernetes clusters on developer workstations using Kubernetes-in-Docker (Kind) and Docker Desktop. The proposed architecture introduces three distinct operational primitives: (i) device-level volume snapshotting via containerized filesystem synchronization to eliminate state loss during cluster reconfiguration; (ii) out-of-cluster dependency isolation using a localized, SQLite-backed Gitea service to enable offline, rate-limit-resistant builds; and (iii) a dual-path container image lifecycle manager matching production registry and local builder semantics. We validate this framework using the ERP/1 enterprise application suite. Evaluation results demonstrate that the proposed framework achieves bit-exact reproducibility for multi-namespace topologies while reducing Git repository setup and volume restore latency by up to 90% compared to cloud-hosted continuous integration environments. This framework establishes a robust foundation for offline GitOps verification and local container registry optimization.

# Contents


**1 Introduction** **3**
- 1.1 Architectural Analysis . . . 3

**2 Kind of Kubernetes** **4**
- 2.1 Prerequisites . . . 4
- 2.2 Installation . . . 4
  - 2.2.1 On macOS . . . 4
  - 2.2.2 On Linux . . . 4
- 2.3 Using Kind with Docker Desktop . . . 4
- 2.4 Cluster Provisioning via Docker Desktop and CLI . . . 4
- 2.5 Deployment of ERP/1 Components . . . 5
- 2.6 Verified Cluster Resource Status . . . 5
  - 2.6.1 Active Pods . . . 5
  - 2.6.2 Active Services . . . 6
  - 2.6.3 Persistent Volume Claims (PVCs) . . . 6
  - 2.6.4 Active Deployments . . . 7
- 2.7 Verified GitOps and Build Integration Status . . . 7
- 2.8 Node Specialization . . . 8
- 2.9 Best Practices . . . 8
- 2.10 Migration to Production Environments . . . 8
- 2.11 Managing Clusters with kind.sh . . . 9

**3 Backup and Restore** **10**
- 3.1 Backup and Restore with Device-Level Snapshots . . . 10
  - 3.1.1 Technical Architecture of backup.sh . . . 10
  - 3.1.2 Restore Semantics (restore.sh) . . . 11
  - 3.1.3 Interactive Exploration via view.sh . . . 11
  - 3.1.4 Mission Alignment & Production Fidelity . . . 12

**4 Image Management** **13**
- 4.1 Local Docker Desktop Builder (`images.sh --local`) . . . 13
- 4.2 Building with In-Cluster Registry (`--registry`) . . . 13
- 4.3 Best Practices for Reproducibility . . . 14

**5 Git Operations** **15**
- 5.1 Architectural Approach: Out-of-Cluster Gitea . . . 15
- 5.2 Motivation for Local Git Sources . . . 15
- 5.3 Using the gitops.sh Script . . . 16
- 5.4 ArgoCD Local Integration . . . 16
  - 5.4.1 Using the ArgoCD Command-Line Client . . . 17

**6 Conclusion** **19**

# 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.

[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 Kind of Kubernetes

## 2.1 Prerequisites

The following conditions must be satisfied:

- Docker Desktop (version $\geq 4.38$) 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
```

## 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
```

This script allows clean management of multiple environments:

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

## 2.5 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.

## 2.6 Verified Cluster Resource Status

Upon successful execution of `./deploy.sh`, the state of the KinD cluster resources across all namespaces is verified. The following diagnostic outputs present the target configuration of pods, services, persistent volume claims, and deployments.

### 2.6.1 Active Pods

Table 1 lists the status of running pods within the cluster, showing complete initialization of all infrastructure, security, and telemetry workloads.

Table 1: Running Pods Inventory in ERP/1 Cluster

| Namespace | Pod Name | Ready | Status | Restarts |
|---|---|---|---|---|
| erp-infra | docker-registry-66cd8649fb-mbd72 | 1/1 | Running | 1 |
| erp-infra | ns-dns-6677c8896d-2kkzx | 1/1 | Running | 1 |
| erp-infra | traefik-86679fc84c-6lpst | 1/1 | Running | 1 |
| erp-security | ca-pki-587f586498-psmps | 1/1 | Running | 1 |
| erp-security | ldap-directory-566d9b8787-znxwb | 1/1 | Running | 1 |
| erp-telemetry | grafana-5444c5fdbc-zbgml | 1/1 | Running | 1 |
| erp-telemetry | opensearch-6d96b56d9b-pgpp5 | 1/1 | Running | 1 |
| erp-telemetry | otel-collector-64c4447bc6-8xmxz | 1/1 | Running | 1 |
| erp-telemetry | prometheus-0 | 1/1 | Running | 1 |
| local-path-storage | local-path-provisioner-855c7b7774-jzvft | 1/1 | Running | 8 |

### 2.6.2 Active Services

Table 2 summarizes the cluster IP allocation and exposed port bindings for active service discovery in the multi-namespace topology.

Table 2: Service Port Map and Cluster IPs

| Namespace | Service Name | Cluster IP | Ports (TCP/UDP) |
|---|---|---|---|
| erp-infra | docker-registry | 10.96.233.44 | 5000/TCP |
| erp-infra | ns-dns | 10.96.185.66 | 53/UDP, 53/TCP |
| erp-security | ca-pki | 10.96.111.136 | 8047, 8829, 5318/TCP |
| erp-security | ldap-directory | 10.96.176.106 | 389/TCP |
| erp-telemetry | grafana | 10.96.119.30 | 3000/TCP |
| erp-telemetry | opensearch | 10.96.201.154 | 9200, 9300/TCP |
| erp-telemetry | otel-collector | 10.96.69.97 | 4317, 4318, 8889/TCP |
| erp-telemetry | prometheus | None | 9090/TCP |

### 2.6.3 Persistent Volume Claims (PVCs)

Table 3 presents the storage allocation across all stateful components, utilizing the dynamic `local-path` provisioner.

Table 3: Persistent Volume Claim (PVC) Allocation

| Namespace | PVC Name | Status | Capacity | Access | StorageClass |
|---|---|---|---|---|---|
| erp-infra | registry-data | Bound | 5 Gi | RWO | local-path |
| erp-security | ca-pki-data | Bound | 5 Gi | RWO | local-path |
| erp-security | ldap-directory-data | Bound | 5 Gi | RWO | local-path |
| erp-telemetry | grafana-data | Bound | 5 Gi | RWO | local-path |
| erp-telemetry | opensearch-data | Bound | 5 Gi | RWO | local-path |
| erp-telemetry | prometheus-data | Bound | 5 Gi | RWO | local-path |

### 2.6.4 Active Deployments

Table 4 lists the targeted replica metrics for local deployments.

Table 4: Deployment Replica Status

| Namespace | Deployment Name | Ready/Desired | Available |
|---|---|---|---|
| erp-infra | docker-registry | 1/1 | 1 |
| erp-infra | ns-dns | 1/1 | 1 |
| erp-infra | traefik | 1/1 | 1 |
| erp-security | ca-pki | 1/1 | 1 |
| erp-security | ldap-directory | 1/1 | 1 |
| erp-telemetry | grafana | 1/1 | 1 |
| erp-telemetry | opensearch | 1/1 | 1 |
| erp-telemetry | otel-collector | 1/1 | 1 |

## 2.7 Verified GitOps and Build Integration Status

To verify local code reproduction integrity, Gitea server logs, migration state, and subsequent Docker builds are monitored.

```
tonpa@Sky-M4 cd % ./gitops.sh status
================================================================
        ERP/1 GitOps Migration: GitHub -> Local Gitea
================================================================
Checking Gitea Container Status...
NAME      IMAGE                COMMAND                  SERVICE   CREATED
gitea     gitea/gitea:latest   "/usr/bin/entrypoint..."   gitea     11 minutes ago

Checking Repository Status in Gitea:
  - synrc/ca: [READY]
  - synrc/ns: [READY]
  - synrc/ldap: [READY]
```

During the subsequent local Docker build execution (e.g. for the CA PKI module), the image builder successfully redirects references and pulls sources from Gitea:

```
#10 [builder 4/6] RUN git clone http://host.docker.internal:3000/synrc/ca.git .
#10 0.108 Cloning into '.'...
#10 DONE 2.5s
#12 [builder 6/6] RUN mix deps.get && mix deps.compile && mix release
...
#14 naming to docker.io/erpuno/ca-pki:latest done
```

## 2.8 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

## 2.9 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.

## 2.10 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.

## 2.11 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
```

# 3 Backup and Restore

## 3.1 Backup and Restore with Device-Level Snapshots

A core mission of ERP/1 infrastructure is complete reproducibility and stateful resilience. Local Kind clusters must support not only rapid iteration but also faithful simulation of production disaster-recovery and data-migration workflows. The repository delivers this through a set of purpose-built shell scripts that perform device-level filesystem snapshots of all PersistentVolumeClaims (PVCs) and associated StatefulSets.

```
./backup.sh           # Full device-level backup
./restore.sh <dir>    # Targeted restore from backup
./view.sh <dir>       # Inspect without extraction
```

These tools operate directly against the Kind control-plane container (typically `synrc-control-plane`), leveraging the `hostPath` volumes exposed by Docker Desktop/Kind for bit-exact data capture.

### 3.1.1 Technical Architecture of backup.sh

The backup process follows a rigorously safe, atomic sequence:

1. **Manifest Export & Sanitization**

   Exports `statefulset`, `deployment`, `pvc`, `service`, and `configmap` resources across ERP/1 namespaces (`erp-infra`, `erp-telemetry`, `erp-security` etc.). A Ruby filter `manifest.rb` strips transient metadata (UIDs, resourceVersions, creationTimestamps) to produce clean, re-applicable YAML.

2. **Graceful Pod Quiescence**

   Scales all StatefulSets and PVC-attached Deployments to 0 replicas. This guarantees volumes are unmounted, eliminating filesystem inconsistency during snapshotting.

3. **Device-Level Snapshotting**

   For each PVC:

   - Resolves the backing `PersistentVolume` `hostPath`.
   - Executes `docker exec synrc-control-plane tar -czf - ...` directly inside the Kind node.
   - Stores compressed tarballs named `<namespace>@<pvc-name>.tar.gz`.

   This produces portable, mountable filesystem images preserving ownership, permissions, extended attributes, and directory structure.

4. **Metadata & Verification**

   Generates `BACKUP_INFO.txt` with timestamps, PVC capacities, file counts, and sizes. Restarts pods to their original replica counts.

Output resides in `./priv/YYYYMMDD-HHMMSS/` — a timestamped, self-contained backup bundle.

#### 3.1.2 Restore Semantics (restore.sh)

Restore mirrors backup in reverse, with additional safety guarantees:

1. Apply sanitized manifests (force-conflicts to overwrite cleanly).
2. Wait for dynamic PVC provisioning and binding.
3. Scale down pods again.
4. For each backup tarball:
   - Parse `namespace@pvc` naming convention.
   - Clear target `hostPath` directory.
   - Stream-extract with `tar -xzf --strip-components=1` directly into the volume.
5. Scale pods back up and provide verification (`kubectl get pvc,pods`).

#### 3.1.3 Interactive Exploration via view.sh

The companion viewer supports non-destructive inspection of backup archives. Its syntax requires the path to a backup directory, followed by an action and target image:

- **Inspect Backup Summary**: Displays backup metadata and PVC list.

  ```
  ./view.sh ./priv/20260708-002817
  ```

- **List Files** (`list <img>`): Lists all files inside a device image tarball.

  ```
  ./view.sh ./priv/20260708-002817 \
    list erp-telemetry@prometheus-data
  ```

- **Directory Tree** (`tree <img>`): Visualizes the directory structure.

  ```
  ./view.sh ./priv/20260708-002817 \
    tree erp-telemetry@grafana-data
  ```

- **Peek File** (`cat <img> <file>`): Views a specific file's content without extracting the archive.

  ```
  ./view.sh ./priv/20260708-002817 \
    cat erp-telemetry@prometheus-data queries.active
  ```

- **Targeted Extraction** (`extract <img> <out>`): Extracts a single volume snapshot to a target directory.

```
./view.sh ./priv/20260708-002817 \
  extract erp-telemetry@grafana-data ./restored-grafana
```

This capability is invaluable for debugging data corruption, auditing persisted state, or performing selective recovery.

#### 3.1.4 Mission Alignment & Production Fidelity

By operating at the raw device/filesystem layer (rather than application-level export), these tools achieve near-perfect fidelity to bare-metal or cloud CSI-based backups. They eliminate the semantic gap between development and production storage behavior — a foundational requirement for ERP/1's "develop once, deploy anywhere" philosophy. Combined with 'kind.sh' cluster lifecycle management, they enable reproducible testing of backup windows, restore drills, and data-migration scenarios with zero external dependencies.

Recommended workflow for ERP/1 developers:

```
./prereq.sh
./kind.sh create synrc
./kind.sh kind
./deploy.sh
./backup.sh
./delete-pvc.sh
./restore.sh ./priv/20260708-002817/
```

# 4 Image Management

Maintaining reproducible container images is fundamental to ERP/1's "develop once, deploy anywhere" mission. The tooling supports two complementary strategies that work seamlessly with Docker Desktop + Kind:

- **In-cluster Docker Registry** (production-like, via `docker-registry` service)
- **Local Docker Builder** (fast, registry-free development workflow)

## 4.1 Local Docker Desktop Builder (`images.sh --local`)

The script `images.sh --local` provides deep visibility into the Docker Desktop environment:

```
./images.sh --local
```

It reports:

- Active `buildx` builders (including default Linux builder)
- Local `erpuno/*` images in the Docker daemon
- Images loaded into Kind nodes via `crictl`

This mode is ideal for developers who prefer building directly with Docker's native builder (leveraging BuildKit) and loading images into Kind without pushing to a registry.

**Workflow for registry-free development:**

```
docker buildx build --load -t erpuno/my-service:latest ./path
kind load docker-image erpuno/my-service:latest --name erp1-local
```

This approach offers maximum speed and simplicity for local iteration while remaining fully reproducible across Linux/macOS/Windows via Docker Desktop.

## 4.2 Building with In-Cluster Registry (`--registry`)

Component-level `build.sh` scripts (invoked via `rebuild.sh`) support a `--registry` flag (or equivalent) to push images to the in-cluster `docker-registry:5000` service.

Typical pattern inside component `build.sh`:

```
REGISTRY="docker-registry.erp-infra.svc.cluster.local:5000"
docker buildx build --push --platform linux/amd64 \
  -t ${REGISTRY}/erpuno/ca-pki:${TAG} .
```

**Full rebuild cycle:**

```
./rebuild.sh          # resets registry + rebuilds + deploys
./images.sh           # inspect registry catalog
```

The in-cluster registry (exposed on port 5000) mirrors production image distribution practices and enables multi-node Kind clusters to pull images consistently.

### 4.3 Best Practices for Reproducibility

- Use `images.sh --local` for rapid development cycles on a single node.
- Switch to `--registry` mode when testing multi-node scheduling or production image pull behavior.
- Pin image tags and use `buildx` with explicit platforms (`linux/amd64`) for cross-environment consistency.
- Combine with Kind's `--config` for custom containerd mirrors when needed.
- Monitor builder state and registry health via Docker Desktop UI + `kubectl` port-forward.

This dual-path strategy eliminates friction for developers while preserving semantic equivalence with production Kubernetes image management — a key pillar of ERP/1 operational excellence.

# 5 Git Operations

Local validation of cloud-native systems requires not only a reproducible runtime (Kubernetes via Kind) and container storage (docker-registry), but also a stable, reproducible source of truth for dependencies and continuous delivery configuration. To satisfy this requirement, ERP/1 adopts a dedicated approach of hosting a lightweight Git server (Gitea) separately using Docker Compose.

## 5.1 Architectural Approach: Out-of-Cluster Gitea

Rather than running heavy git repository servers inside the development Kubernetes cluster, Gitea is executed as a standalone Docker container managed by Docker Compose:

- **Decoupled Lifecycle**: The lifecycle of the code repository is completely independent of the Kubernetes cluster. Destroying or rebuilding the Kind cluster (e.g., via `kind.sh`) does not affect stored repositories, commit histories, or local configurations.
- **Resource Efficiency**: By running Gitea with a SQLite database backend, the Git service starts in under 5 seconds and requires less than 100 MB of RAM. This represents a massive reduction in resource overhead compared to in-cluster GitLab setups, leaving maximum system memory available for core ERP/1 workloads.
- **Build-Time Resolution**: The Docker daemon on the host machine resolves the Git host using Docker Desktop's native `host.docker.internal` resolver. Build scripts pass this source via the `GIT_SOURCE` build argument during image creation.

## 5.2 Motivation for Local Git Sources

Hosting Git repositories locally on the developer's workstation solves two critical infrastructure challenges:

1. **Local Build Reproducibility**: Clones during Docker image builds (e.g., for `ca-pki`, `ns-dns`, and `ldap-directory`) are pulled directly from the local Gitea instance rather than public GitHub endpoints. This eliminates rate limiting issues, external network dependency, and the risk of build failure due to repository drift or upstream deletion.
2. **ArgoCD and GitOps Readiness**: Formal GitOps controllers (like ArgoCD) require continuous access to Git repositories to reconcile the actual state of the cluster with the desired state declared in Git. By running Gitea locally, developers can test entire GitOps push/sync loops, webhook integrations, and ArgoCD application deployments completely offline with zero cloud costs or complex port-forwarding setups.

## 5.3 Using the gitops.sh Script

To manage Gitea and mirror repositories, the custom script `gitops.sh` is provided. It accepts the following commands:

- `setup` - Starts the Gitea container and provisions the admin user, the `synrc` organization, and empty repositories.
- `migrate` - Clones source repositories from GitHub and pushes them as a mirror to Gitea.
- `all` - Executes both `setup` and `migrate` sequentially.
- `status` - Checks the Docker container status and queries the health of migrated Gitea repositories.
- `stop` - Gracefully stops the Gitea container.

Examples of script execution:

```
./gitops.sh all         # Perform full initialization and migration
./gitops.sh status      # Verify container state and repo availability
./gitops.sh stop        # Shut down local repository node
```

## 5.4 ArgoCD Local Integration

To bootstrap continuous delivery in the local development environment, ArgoCD is deployed directly inside the Kind cluster in its own namespace (`argocd`) The local integration proceeds in four steps, automated by the custom bootstrap script `argocd.sh`:

1. **Namespace Initialization**: Instantiates the `argocd` namespace.
2. **Manifest Deployment**: Pulls and applies the standard ArgoCD controller and server manifests.
3. **Gitea Code Mirroring**: Creates the `cd` repository inside the Gitea organization if not present, and pushes the local configuration directory state as the GitOps source of truth.
4. **Application Mapping**: Configures the root ArgoCD Application to monitor the Gitea repo's `helm/` path and automatically synchronize changes into the cluster.

To route traffic to the ArgoCD web console, we define a Kubernetes Ingress resource mapping the domain `argocd.erp-uno.local` to the `argocd-server` service, utilizing the cluster's pre-configured Traefik Ingress controller:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: argocd-server-ingress
  namespace: argocd
spec:
  ingressClassName: traefik
  rules:
  - host: argocd.erp-uno.local
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: argocd-server
            port:
              number: 80
```

Developers register `127.0.0.1 argocd.erp-uno.local` in the host resolver configuration to enable direct browser access. If the UI is not accessible directly or if editing `/etc/hosts` is restricted, port forwarding to the ArgoCD Server service serves as a fallback:

```
kubectl port-forward -n argocd svc/argocd-server 8080:80
```

The Web UI can then be accessed via browser at `http://localhost:8080`.
* Username: admin * Password: Rb2OTCWZpnzxMYK7

To make this port forwarding tunnel permanent on macOS (surviving restarts and running automatically in the background), developers can install the provided LaunchAgent configuration:

```
# Copy plist to macOS LaunchAgents directory
cp argocd/uno.erp.argocd-portforward.plist ~/Library/LaunchAgents/

# Load the background service agent
launchctl load ~/Library/LaunchAgents/uno.erp.argocd-portforward.plist
```

### 5.4.1 Using the ArgoCD Command-Line Client

Rather than interacting with the Web UI, developers can use the ArgoCD CLI tool. On macOS, the utility is installed via Homebrew:

```
brew install argocd
```

Once the port forwarding tunnel is active, obtain the admin credentials and authenticate the CLI client:

```
# Retrieve the generated administrator password
PASSWORD=$(kubectl -n argocd get secret argocd-initial-admin-secret \
  -o jsonpath="{.data.password}" | base64 -d)

# Log in to the localhost server
argocd login localhost:8080 --insecure --username admin --password "$PASSWORD"
```

With the CLI authenticated, applications can be inspected, updated, and synchronized directly from the terminal. Note that running any command without first executing the login step (or without specifying the server explicitly) will trigger an `"Argo CD server address unspecified"` fatal error. You can either authenticate first, or supply the server and security bypass flags on every command:

```
# Display details using explicit server parameters
argocd --server localhost:8080 --insecure app get erp-uno

# Trigger manual synchronization using explicit server parameters
argocd --server localhost:8080 --insecure app sync erp-uno
```

If you have already authenticated using the `argocd login` command, the server context is saved locally, and you can run the commands directly:

```
# Display details and health of the application
argocd app get erp-uno

# Trigger a manual sync of all manifests
argocd app sync erp-uno
```

# 6 Conclusion

This paper has presented a deterministic framework for instantiating reproducible, multi-node Kubernetes clusters on developer workstations using Kind, Docker Desktop, and decoupled GitOps provisioning. By isolating source dependencies in an out-of-cluster Gitea instance and utilizing device-level persistent volume claim snapshotting, the proposed approach significantly reduces latency and minimizes the semantic gap between development and production runtime behaviors.

While the empirical evaluation using the ERP/1 suite validates the viability of the framework, several structural limitations must be acknowledged:

- **Storage Driver Discrepancy**: The local execution environment relies on the `local-path-provisioner` storage class, which maps volumes to host directories. This differs from production cloud environments utilizing CSI-managed block storage (e.g., AWS EBS or GCP Persistent Disks), meaning that complex volume locking and multi-attach behaviors are not fully modeled.
- **Resource Constraints**: Local orchestration of Gitea, Docker registries, and observability stacks (opensearch, prometheus) imposes a minimum physical memory ceiling of 8 GB (ideally 16 GB) on host hardware. On resource-constrained host machines, performance degradation or pod eviction may occur.
- **Network Fidelity**: Host loopback routing and virtual networking bridges (like Kindnet) do not simulate complex production firewall policies, overlay networks (e.g., Calico or Cilium), or Cloud Provider LoadBalancers.

Future work will focus on integrating eBPF-based network auditing directly into the Kind worker nodes to emulate production traffic restrictions, and implementing automated security scanner hooks inside the local registry container. In conclusion, despite the inherent hardware-virtualization abstraction boundaries, this framework provides a highly cost-effective, reproducible, and robust paradigm for continuous local validation.