Code-To-Container Traceability

Table Of Contents

Code-to-Container Traceability

1. Introduction

Modern cloud-native application delivery requires more than just continuous integration and deployment. Organizations need **end-to-end traceability**, visibility, and observability across the entire software delivery lifecycle — from source code commit to running container in production.

This project, **Code-to-Container Traceability Platform**, was designed and implemented to achieve full traceability across CI/CD pipelines, Kubernetes deployments, and runtime infrastructure, while integrating centralized metadata storage and visualization dashboards.

The system combines **CI automation (Jenkins), containerization (Docker), GitOps-based CD (ArgoCD), Kubernetes orchestration, OpenSearch-based metadata storage, and Grafana-based observability dashboards** to provide complete release transparency.

1.1 Project Overview

The Code-to-Container Traceability Platform is a cloud-native DevOps architecture that enables:

- End-to-end traceability from Git commit → Docker image → Kubernetes deployment → Running Pod
- Immutable image tracking using SHA256 digests
- GitOps-driven Continuous Deployment using ArgoCD
- Automated deployment metadata collection
- Centralized metadata storage in OpenSearch
- Visualization of release and runtime insights using Grafana dashboards
- Periodic cluster-wide state scanning using Kubernetes CronJobs

The system operates on a Kubernetes cluster (K3s-based environment) and implements industry best practices such as:

- Infrastructure as Code (IaC)
- GitOps workflows
- Immutable artifact versioning
- Observability-first design
- Declarative deployment model
- Automated metadata enrichment
- Secure in-cluster service communication

1.2 Problemt Statement

In traditional CI/CD setups, the build pipeline and deployment pipeline often operate in isolation. While builds generate container images and deployments roll them out, there is usually no centralized mechanism to answer critical operational questions such as:

- Which Git commit is currently running in production?
- Which developer authored the deployed code?
- What Docker image digest is deployed?
- When was a specific version released?
- What Kubernetes node is hosting the application?
- How many replicas are active?
- What was the deployment strategy used?
- What other resources were impacted by the deployment?

Without unified traceability:

- Root cause analysis becomes difficult
- Incident response time increases
- Compliance and audit tracking is weak
- Deployment impact assessment is manual
- Rollback decisions lack data context
- Platform observability remains fragmented

Additionally, most monitoring solutions focus only on infrastructure metrics, not on code-level release traceability.

There was a need to build a system that bridges the gap between:

- Application source control
- CI pipelines
- Container registries
- GitOps-based CD
- Kubernetes runtime state
- Observability platforms

1.3 Solution Approach

To address the above challenges, a fully integrated cloud-native traceability framework was designed using the following approach:

1. CI Layer (Jenkins + Docker + Kaniko)

- Automatically triggered on Git commit
- Builds Docker image with unique tag
- Extracts image digest from Kaniko build logs
- Captures commit metadata (author, message, commit ID)
- Stores CI metadata into OpenSearch

2. CD Layer (ArgoCD – GitOps Model)

- Watches Git repository for manifest changes
- Automatically syncs desired state to Kubernetes
- Deploys updated container image
- Ensures self-healing and drift correction

3. Deployment Metadata Collector

- Captures runtime deployment details:
    - Pod information
    - Deployment spec
    - Image digest
    - Replica count
    - Node assignment
    - Deployment strategy
- Runs in:
    - Post-deployment mode
    - Periodic cluster-scan mode
- Pushes deployment metadata into OpenSearch

4. Data Layer (OpenSearch)
- Stores:
    - CI build metadata

    - Deployment metadata
    - Combined traceability documents
- Enables structured querying and time-series analysis

5. Observability Layer (Grafana + Prometheus)

- Visualizes:
    - Release history
    - Deployment frequency
    - Build success rate
    - Pod health
    - Runtime performance
    - Deployment impact
- Enables operational decision-making through dashboards

This architecture ensures a closed-loop DevOps lifecycle where code, container, deployment, and runtime states are continuously linked and observable.

1.4 Key Objectives

The primary objectives of this project were:

1. Establish full commit-to-container traceability
2. Implement GitOps-based Continuous Deployment
3. Enable immutable image version tracking using SHA256 digests
4. Centralize CI and CD metadata into a searchable datastore
5. Automate runtime deployment metadata capture
6. Provide cluster-wide periodic state scanning
7. Enable observability-driven release governance
8. Improve auditability and compliance readiness
9. Reduce Mean Time To Detect (MTTD) and Mean Time To Recover (MTTR)
10. Provide a production-grade DevOps reference architecture

1.5 Key Acheivements

The project successfully delivered the following outcomes:

- Designed and implemented a production-ready CI/CD + Observability architecture
- Achieved full traceability from Git commit to running Kubernetes pod
- Integrated immutable Docker digest tracking into CI metadata
- Implemented GitOps model using ArgoCD with automated sync
- Developed custom Kubernetes metadata collector using Python client
- Stored structured deployment metadata in OpenSearch
- Created multiple Grafana dashboards covering:
    - Release tracking
    - Deployment history
    - Pod runtime health
    - Resource utilization
    - Deployment impact analysis
- Enabled cluster-wide periodic scanning via Kubernetes CronJobs
- Established centralized observability model for platform monitoring
- Reduced manual deployment tracking effort to zero
- Built a reusable cloud-native DevOps reference implementation

2. System Architecture

The Code-to-Container Traceability Platform is built using a layered, cloud-native architecture that integrates Continuous Integration (CI), Continuous Deployment (CD), Kubernetes orchestration, metadata persistence, and observability tooling into a unified traceability framework.

The architecture follows DevOps and GitOps best practices, ensuring:

- Declarative deployments
- Immutable artifacts
- Automated synchronization
- Metadata enrichment
- Centralized observability
- Scalable and loosely coupled components

The system is modular and extensible, allowing additional services, clusters, or environments to be integrated without architectural redesign.

2.1 High Level Architecture Diagram

Developer Commit (GitHub)

↓

CI Layer (Jenkins + Docker + Kaniko)

↓

Container Registry (Docker Hub)

↓

CD Layer (ArgoCD – GitOps)

↓

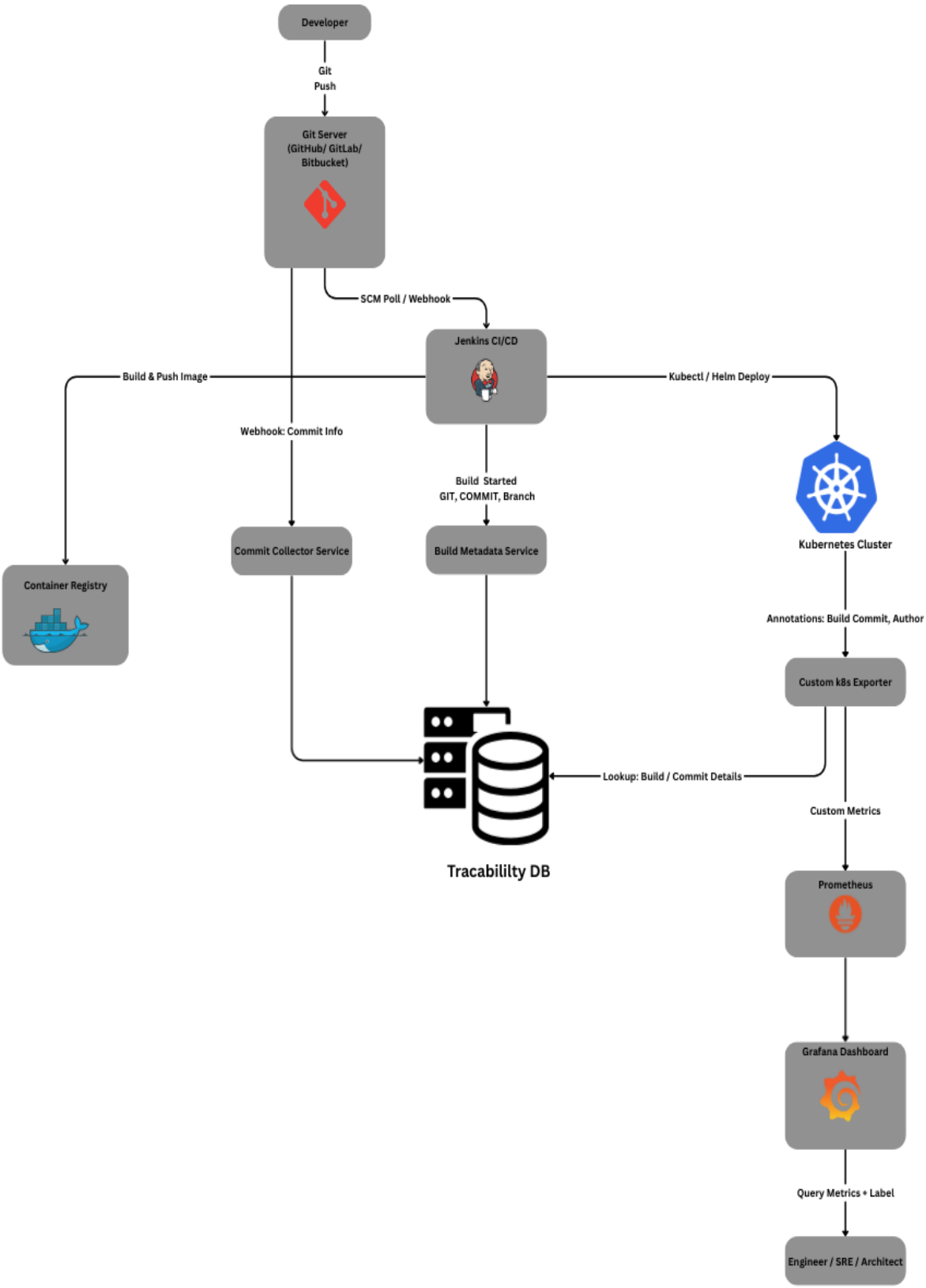
Kubernetes Cluster (Application + Collector Jobs)

↓

Metadata Storage (OpenSearch)

↓

Visualization (Grafana + Prometheus)



Architecture Diagram

Architectural Components

1. Source Control
    - GitHub repository
    - Stores application source code and Kubernetes manifests
    - Triggers CI pipeline on commit
2. CI Layer
    - Jenkins pipeline automation
    - Docker image build via Kaniko
    - Extracts image SHA256 digest
    - Generates CI metadata JSON
    - Pushes build metadata to OpenSearch
3. Container Registry
    - Docker Hub
    - Stores versioned container images
    - Image tags include build number and commit ID
    - Digest ensures immutability
4. CD Layer
    - ArgoCD (GitOps controller)
    - Monitors Kubernetes manifest repository
    - Automatically syncs cluster state with Git state
    - Performs declarative deployments
5. Kubernetes Runtime Layer
    - Deployments, Pods, Services, IngressRoutes
    - Collector CronJobs for cluster scanning
    - Deployment-specific trace jobs
    - RBAC-controlled service accounts

6. Metadata Layer
    - OpenSearch
    - Stores:
        - CI build metadata
        - Deployment metadata
        - Combined traceability documents
7. Observability Layer
    - Prometheus (metrics scraping)
    - Grafana dashboards
    - OpenSearch as data source
    - Release intelligence visualization

This layered architecture ensures complete separation of concerns while maintaining strong traceability linkage.

2.2 End-to-End Workflow

The platform implements a complete end-to-end release lifecycle.

Step 1 – Code Commit

- Developer pushes code to GitHub.
- Webhook triggers Jenkins CI pipeline.

Step 2 – Continuous Integration

Jenkins pipeline performs:

1. Source checkout
2. Commit metadata extraction
3. Docker image build using Kaniko
4. Image push to Docker Hub
5. Image digest extraction (SHA256)
6. Build metadata JSON generation
7. Metadata storage in OpenSearch (ci-build-metadata index)

Generated CI Metadata includes:

- Build ID
- Commit ID
- Author details
- Image tag
- Image digest
- Branch
- Timestamp

Step 3 – GitOps Deployment (ArgoCD)

- Kubernetes manifest repository updated with new image tag.
- ArgoCD detects change.
- Automatically syncs application.
- Deployment is created/updated in cluster.

ArgoCD ensures:

- Declarative desired state enforcement
- Drift detection
- Self-healing behavior

Step 4 – Deployment Metadata Collection

Deployment metadata collector performs:

- Fetch deployment details via Kubernetes API
- Capture:
    - Pod name
    - Node name
    - Replica count
    - Image tag
    - Image digest
    - Deployment strategy
    - Labels

    - Runtime status
- Store data in OpenSearch (deployment-metadata index)

Collector operates in two modes:

1. Post-deployment mode (triggered after sync)
2. Periodic cluster-scan mode (CronJob)

Step 5 – Metadata Aggregation

CI and CD metadata are merged into a combined document:

- Commit → Build → Image → Deployment → Pod
- Stored in a unified index
- Enables complete release traceability

Step 6 – Observability & Visualization

Grafana dashboards provide:

- Release history visualization
- Deployment frequency
- Pod health monitoring
- Resource utilization metrics
- Impact analysis
- Build success trends

This creates a closed-loop observability pipeline from code to runtime.

2.3 Architecture Design Pipelines

The platform was built using the following key cloud-native design principles:

1. GitOps-Driven Deployment

- Kubernetes manifests stored in Git
- ArgoCD reconciles cluster state
- Eliminates manual kubectl-based deployments

2. Immutable Infrastructure

- Docker images are versioned using build number + commit ID
- SHA256 digest ensures content immutability
- No mutable latest tags in production

3. Separation of Concerns

- CI handles build logic
- CD handles deployment logic
- Collector handles metadata extraction
- OpenSearch handles storage
- Grafana handles visualization

Each layer is independently scalable.

4. Observability-First Architecture

- Metadata treated as first-class data
- All release events are persisted
- Historical traceability enabled
- Supports compliance and audit readiness

5. Declarative & Automated Operations

- Kubernetes resources defined as YAML
- CronJobs automate cluster scanning

- No manual runtime data collection

6. API-Driven Integration

- Kubernetes Python client used for metadata collection
- OpenSearch REST APIs used for ingestion
- Prometheus metrics scraped via standard exporters

7. Cloud-Native Scalability

- Kubernetes-based execution
- Horizontal scalability
- Stateless collector pods
- Scalable OpenSearch backend

2.4 Technology Stack Overview

The platform integrates multiple cloud-native and DevOps tools.

CI & Build Layer

- Jenkins
- Declarative Pipeline (Groovy)
- Docker
- Kaniko (rootless image builder)

Version Control

- GitHub
- Webhooks

Container Registry

- Docker Hub

CD Layer

- ArgoCD
- GitOps workflow
- Kubernetes manifests

Orchestration

- Kubernetes (K3s-based cluster)
- Deployments
- Services
- IngressRoute (Traefik)
- CronJobs
- ServiceAccounts & RBAC

Metadata & Storage

- OpenSearch
- REST API ingestion
- Custom index mappings

Observability & Visualization

- Prometheus
- Grafana
- OpenSearch data source
- Custom dashboards

Programming & Automation

- Python (Kubernetes client library)
- Shell scripting
- Curl-based REST interactions
- YAML (Infrastructure as Code)

3. Continuous Integration (CI) Layer

The Continuous Integration (CI) layer is responsible for transforming source code into immutable, versioned container artifacts while generating structured build metadata for traceability.

This layer ensures:

- Automated build execution on code commit
- Reproducible container image creation
- Immutable artifact versioning
- Metadata enrichment for traceability
- Seamless integration with CD and observability layers

The CI layer is implemented using Jenkins, Kaniko, Docker Hub, and structured metadata publishing to OpenSearch.

3.1 Jenkins Architecture

The CI system is built on a cloud-native Jenkins deployment running inside Kubernetes.

Architectural Components

1. Jenkins Controller

- Orchestrates pipeline execution
- Stores pipeline definitions (Jenkinsfile)
- Manages credentials and job configurations

2. Kubernetes Agent Pods

- Dynamically provisioned using Jenkins Kubernetes plugin
- Ephemeral and build-specific
- Auto-destroyed after build completion

3. Kaniko Build Container

- Runs inside agent pod
- Builds Docker images without requiring Docker daemon
- Pushes image directly to Docker Hub

4. Credential Management

- Docker registry credentials stored as Kubernetes secret

- GitHub credentials managed via Jenkins credential store

Execution Flow

1. GitHub webhook triggers Jenkins.
2. Jenkins provisions dynamic Kubernetes build pod.
3. Kaniko container builds and pushes image.
4. Metadata is generated and stored.

3.2 CI Pipeline Workflow

The Jenkins pipeline is implemented using a Declarative Pipeline (Jenkinsfile).

Stage 1 – Source Checkout

- Pulls latest code from GitHub.
- Captures commit history.
- Determines commit range from last successful build.

Stage 2 – Metadata Preparation

- Generates unique Build ID:

  JOB_NAME-BUILD_NUMBER

- Extracts:
  - Commit ID
  - Author
  - Commit message
  - Email
- Generates initial build-metadata.json.

Stage 3 – Docker Image Build & Push

- Builds image using Kaniko.
- Pushes image to Docker Hub.
- Extracts SHA256 image digest.
- Injects digest into metadata JSON.

Stage 4 – Metadata Storage

- Sends final JSON to OpenSearch.
- Stored in ci-build-metadata index.

The pipeline is fully automated and runs on every commit.

3.3 Docker Image Build using Kaniko

Why Kaniko?

Kaniko is used instead of Docker because:

- Runs inside Kubernetes without privileged mode
- Does not require Docker daemon
- Secure for production CI pipelines
- Supports registry authentication

Build Execution

Kaniko runs with:

```
/kaniko/executor \
  --dockerfile=Dockerfile \
  --context=workspace \
  --destination=dockerhub/image:tag \
  --verbosity=info
```

Advantages

- Rootless build process
- Cloud-native compatible
- Immutable image push
- Registry-native digest generation

This aligns with modern container security best practices.

3.4 Image Tagging Strategy

A strong tagging strategy is critical for traceability.

```
<safe-job-name>-<build-number>-<short-commit-id>
```

Why This Strategy?

- Avoids using mutable latest
- Links image directly to:
  - CI job
  - Build number
  - Specific commit
- Enables rollback using build ID

Best Practices Applied

- Immutable image references
- Commit-based versioning
- No overwriting of tags
- Clear audit mapping

3.5 Image Digest Extraction

After pushing image to Docker Hub, Kaniko outputs:

```
Pushed index.docker.io/...@sha256:<digest>
```

The pipeline extracts digest using:

```
grep -o 'sha256:[a-f0-9]\{64\}'
```

Why Digest is Important

Tags can be mutable.
Digest is immutable.

Using digest ensures:

- Binary-level integrity
- No tampering
- Strong release verification
- Supply chain security compliance

Both tag and digest are stored in metadata.

3.6 CI Metadata Collection

The CI pipeline generates structured metadata in JSON format.

CI Metadata Fields


```
{
  "build_id": "Node-JS/master-67",
  "job_name": "Node-JS/master",
  "build_number": 67,
  "branch": "master",
  "image_name": "narendra115c/node-js",
  "image_tag": "node-js-master-67-3606e8f",
  "image_digest":
"sha256:a0243d98dd20494f723081c48fa5417bde51458f579718cdccdde03a740725f7",
  "commits": [
    {
      "commit_id": "3606e8f26e051e93d8784ae87879c77845029b33",
      "author": "Narendra Sivangula",
      "email": "narendrakumarsivangula@gmail.com",
      "message": "Update index.js"
    }
  ],
  "authors": [
    "Narendra Sivangula <narendrakumarsivangula@gmail.com>"
  ],
  "build_status": "SUCCESS",
  "timestamp": "2026-01-31T05:45:18Z"
}
```


Key Traceability Links

- Commit → Build ID
- Build ID → Image Tag
- Image Tag → Image Digest

This metadata enables complete release lineage tracking.

3.7 CI Metadata Storage

Metadata is stored in OpenSearch using REST API:

POST /ci-build-metadata/_doc

Index Design

Index: **ci-build-metadata**

Stores:

- Build metadata
- Commit history
- Image information
- Timestamps

Benefits of Centralized Metadata Storage

- Historical build tracking
- Audit readiness
- Queryable release data
- Integration with Grafana dashboards
- Aggregation with deployment metadata

CI Layer Outcome

After CI execution, the system has:

- Immutable Docker image
- Extracted SHA256 digest
- Structured build metadata
- Stored searchable document in OpenSearch
- Full commit-to-image traceability

This forms the foundation for the CD and deployment traceability layers.

4. Continuous Delivery (CD) Layer – GitOps

The Continuous Delivery (CD) layer implements a GitOps-based deployment strategy using ArgoCD.

This layer is responsible for:

- Declarative Kubernetes deployments
- Automated synchronization of Git changes
- Environment consistency and drift correction
- Deployment traceability
- Immutable artifact promotion

The CD system ensures that every deployment is reproducible, version-controlled, and auditable.

4.1 GitOps Concept

What is GitOps?

GitOps is an operational framework that uses Git as the single source of truth for infrastructure and application configuration.

Instead of manually deploying workloads:

- All Kubernetes manifests are stored in Git.
- Any change to Git automatically triggers deployment.
- The cluster state is continuously reconciled with Git state.

Core GitOps Principles Applied

1. Declarative Infrastructure
    - Kubernetes manifests define desired state.
    - No manual kubectl changes in production.
2. Version-Controlled Configuration
    - Every deployment is tied to a Git commit.
    - Full audit trail is maintained.
3. Automated Reconciliation
    - ArgoCD continuously monitors Git.
    - Automatically applies changes.

4. Drift Detection & Self-Healing
    - If cluster state deviates, ArgoCD corrects it.

Benefits in This Project

- Commit → Image → Deployment mapping
- Zero manual deployment steps
- Traceable environment state
- Controlled rollback capability

4.2 ArgoCD Architecture

ArgoCD is deployed inside the Kubernetes cluster as a set of controller components.

Core Components

1. ArgoCD API Server

- Exposes UI and REST API
- Manages application configurations
- Provides authentication and RBAC

2. Application Controller

- Continuously compares Git state with cluster state
- Applies changes if drift detected

3. Repository Server

- Clones Git repository
- Generates Kubernetes manifests
- Supports Helm, Kustomize, plain YAML

4. Redis

- Used for caching and state management

Deployment Model Used

- ArgoCD runs inside argocd namespace.
- Accessed via Traefik IngressRoute.
- Configured for HTTP access (non-TLS in lab environment).
- Connected to Git repository containing Kubernetes manifests.

High-Level Flow

1. Developer pushes code.
2. Jenkins builds image and pushes to registry.
3. Deployment manifest in Git is updated with new image tag.
4. ArgoCD detects Git change.
5. ArgoCD syncs new state to cluster.
6. Deployment rollout begins.

This ensures fully automated CI → CD integration.

4.3 Application Configuration

4.4 Auto-Sync & Self-Heal

4.5 Deployment Trigger Flow

The deployment flow integrates CI and CD seamlessly.

End-to-End Flow

1. Developer pushes commit to GitHub.
2. Jenkins CI pipeline:
    - Builds Docker image
    - Tags image with build number + commit ID
    - Extracts image digest
    - Stores CI metadata in OpenSearch
3. Deployment Manifest Update:
    - Kubernetes manifest references updated image tag.
4. ArgoCD Detection:
    - ArgoCD monitors Git repository.
    - Detects change in image tag.
5. Synchronization:
    - ArgoCD applies updated manifest.
    - Kubernetes Deployment performs rolling update.

6. Deployment Metadata Collector:
    - Captures runtime metadata:
        - Pod name
        - Node name
        - Image digest
        - Deployment strategy
        - Replica count
        - Status
    - Sends data to OpenSearch.

Traceability Chain

Commit → Build → Image → Digest → Deployment → Pod → Runtime Status

This ensures full lifecycle visibility from code to production runtime.

CD Layer Outcome

After implementing the CD layer:

- Git is the single source of truth.
- Deployments are fully automated.
- Cluster drift is eliminated.
- Rollbacks are version-controlled.
- Deployment events are traceable.
- Runtime metadata is collected and stored.

This forms the backbone of the traceability framework.

5. Kubernetes Deployment Layer

The Kubernetes Deployment layer is responsible for executing application workloads in the cluster using declarative configuration and automated orchestration.

This layer transforms the desired state defined in Git (via ArgoCD) into running application instances while ensuring:

- High availability
- Rolling updates
- Fault tolerance
- Auto-healing
- Runtime consistency
- Traceability of deployed workloads

The deployment process is fully automated and tightly integrated with the GitOps delivery pipeline.

5.1 Deployment Lifecycle

The Kubernetes Deployment lifecycle represents the progression of an application from manifest definition to runtime execution.

Lifecycle Stages

1. Manifest Application

- ArgoCD applies Kubernetes Deployment YAML.
- Deployment object is created in the cluster.

2. ReplicaSet Creation

- Kubernetes automatically generates a ReplicaSet to manage pod replicas.

3. Pod Creation

- ReplicaSet creates pods based on desired replica count.

4. Image Pull

- Pods pull container image from registry using tag/digest.

5. Scheduling

- Scheduler assigns pods to worker nodes.

6. Runtime Execution

- Containers start and application becomes available.

7. Health Monitoring

- Readiness and liveness probes ensure application health.

8. Scaling / Update

- Rolling update strategy ensures zero-downtime deployment.

Traceability Integration

Each lifecycle stage contributes metadata used for observability:

- Deployment name
- Replica count
- Pod status
- Node placement
- Image digest
- Runtime health state

5.2 ReplicaSet & Pod Creation

ReplicaSet Role

ReplicaSet ensures that the desired number of pods is always maintained.

Responsibilities include:

- Creating pods during deployment
- Recreating pods if deleted
- Scaling pods up/down
- Managing rolling updates

Pod Creation Process

When ReplicaSet is created:

1. Pod template is extracted from Deployment spec.
2. Pods are created with:
    - Container image
    - Environment variables
    - Labels and selectors

- Resource limits
- Volume mounts

3. Pod status transitions:
    - Pending → ContainerCreating → Running

Observability Data Collected

The traceability collector captures:

- Pod name
- Deployment ownership
- Namespace
- Node assignment
- Container image
- Runtime status

This enables mapping between deployment and actual runtime pods.

5.3 Image Pull Using Digest

Why Digest-Based Deployment

The system uses image digest instead of mutable tags for runtime verification.

Benefits include:

- Immutable artifact identification
- Supply chain security
- Exact version traceability
- Reproducible deployments
- Prevention of tag drift

Deployment Behavior

1. Jenkins extracts image digest during CI build.
2. Digest is stored in OpenSearch metadata.
3. Kubernetes pulls image from registry.
4. Collector verifies actual running image digest.

Traceability Mapping

This enables a deterministic mapping:

Commit → Image Tag → Image Digest → Running Container

Ensuring complete software supply chain transparency.

5.4 Scheduling & Runtime Behavior

Scheduling Process

The Kubernetes scheduler determines node placement based on:

- Resource availability (CPU, memory)
- Node affinity / anti-affinity
- Taints and tolerations
- Pod priority
- Topology constraints

Runtime Execution

Once scheduled:

- Container runtime pulls image
- Pod networking is configured
- Application process starts
- Service discovery registers endpoints

Runtime Metadata Captured

The traceability collector records:

- Node name
- Pod IP
- Container runtime status
- Resource allocation
- Deployment strategy
- Replica count

This data enables operational visibility and performance analysis.

5.5 Readiness & Liveness Strategy

Health checks are critical to ensuring application reliability.

Readiness Probe

Determines if the pod is ready to receive traffic.

Use cases:

- Application startup delay
- Dependency initialization
- Service warm-up

If readiness fails:

- Pod is removed from Service endpoints
- Traffic is not routed

Liveness Probe

Detects application failures and triggers container restart.

Use cases:

- Deadlock detection
- Runtime crash recovery
- Memory leak handling

If liveness fails:

- Container is restarted automatically

Benefits

- Zero-downtime deployments
- Auto-healing runtime
- Improved SLA reliability
- Faster fault recovery

Observability Integration

Health status is captured in deployment metadata:

- Pod phase
- Restart count
- Container state

- Availability status

6. Deployment Metadata Collector

The Deployment Metadata Collector is a custom-built Kubernetes telemetry component responsible for capturing runtime deployment data and correlating it with CI pipeline metadata.

It enables **end-to-end traceability** from:

Git Commit → CI Build → Container Image → Kubernetes Deployment → Running Pod

The collector operates as an event-driven and scheduled data ingestion service that continuously captures cluster state and publishes structured metadata to OpenSearch for observability and analytics.

6.1 Collector Architecture

The metadata collector is implemented as a lightweight Python-based service leveraging the Kubernetes client SDK and OpenSearch REST APIs.

Architecture Components

1. Kubernetes API Client

- Fetches Deployment, Pod, and runtime metadata
- Reads image details and cluster state

2. Execution Modes

- Post-sync event collector (ArgoCD hook)
- Periodic cluster scanner (CronJob)

3. Correlation Engine

- Matches deployment metadata with CI build metadata
- Maps image digest to build artifacts

4. Data Publisher

- Sends structured JSON documents to OpenSearch

Architectural Benefits

- Event-driven observability
- Low resource overhead
- Extensible metadata schema

- Cloud-native integration
- GitOps-compatible design

6.2 Post-Sync Execution Mode

Post-sync execution mode captures deployment metadata immediately after ArgoCD completes synchronization.

Execution Flow

1. Git commit triggers Jenkins CI pipeline
2. CI builds and pushes container image
3. ArgoCD syncs manifests
4. ArgoCD PostSync hook triggers collector Job
5. Collector captures deployment metadata
6. Data is published to OpenSearch

Advantages

- Near real-time deployment traceability
- Accurate deployment version tracking
- Eliminates polling latency
- Event-driven observability

Data Captured

- Deployment name
- Namespace
- Replica count
- Pod list
- Node placement
- Image tag and digest
- Deployment strategy
- Runtime status

6.3 Cluster Scan (CronJob) Mode

Cluster scan mode periodically collects metadata across all workloads to detect configuration drift and runtime changes.

Execution Behavior

- Runs as Kubernetes CronJob
- Scans entire cluster resources
- Detects new deployments and pods
- Updates OpenSearch metadata

Schedule Strategy

Example interval:

```
*/10 * * * *
```

Benefits

- Continuous cluster visibility
- Drift detection
- Runtime change monitoring
- Historical workload tracking
- Coverage for manual deployments

6.4 Metadata Fields Captured

The collector captures comprehensive metadata across multiple layers.

Deployment-Level Metadata

- Deployment name
- Namespace
- Replica count
- Update strategy
- Labels
- Annotation metadata

Pod-Level Metadata

- Pod name

- Node assignment
- Container image
- Image digest
- Pod phase and status
- Restart count

CI Correlation Metadata

- Build ID
- Commit ID
- Author details
- Image tag
- Build status

Runtime Metadata

- Scheduling node
- Timestamp
- Health state
- Container runtime status

This data enables deep operational analytics and traceability.

6.5 Data Correlation Logic

The collector performs intelligent correlation between CI and deployment data using immutable identifiers.

Correlation Keys

Primary Keys

- Image digest
- Image tag
- Build ID

Secondary Keys

- Commit ID
- Deployment labels
- Pod owner references

Correlation Workflow

1. Deployment metadata is captured
2. Image digest extracted from pod spec
3. CI metadata queried from OpenSearch
4. Records merged into unified traceability document
5. Combined document stored in traceability index

Outcome

This correlation enables answering critical questions:

- Which commit caused this deployment?
- Who authored the deployed change?
- Which image version is running?
- What pods are affected?
- Which node hosts the deployment?

6.6 Storage in OpenSearch

The collector publishes metadata as structured JSON documents to OpenSearch indices.

Indices Used

- ci-build-metadata
- deployment-metadata
- traceability-metadata (correlated index)

Storage Benefits

- Distributed indexing
- Fast search and aggregation
- Time-series analysis
- Dashboard integration with Grafana
- Historical traceability

Data Ingestion Method

The collector uses REST-based ingestion:

POST /deployment-metadata/_doc

Documents are timestamped to enable time-based querying and visualization.

7. Data Layer – OpenSearch

The OpenSearch data layer acts as the centralized telemetry storage platform for CI/CD metadata, Kubernetes runtime information, and traceability analytics.

It provides scalable indexing, fast querying, and time-series data storage capabilities required for building observability dashboards and performing deployment impact analysis.

OpenSearch serves as the system's traceability backbone, enabling unified visibility across the entire software delivery lifecycle.

7.1 Role of OpenSearch

OpenSearch is used as a distributed search and analytics engine to store structured metadata generated across CI, CD, and runtime environments.

Key Responsibilities

- Storage of CI build metadata
- Storage of Kubernetes deployment metadata
- Correlation of CI and deployment data
- Time-series traceability analytics
- Enabling Grafana dashboard queries
- Supporting root cause analysis workflows

Architectural Benefits

- Horizontally scalable indexing
- Near real-time search capability
- Schema-flexible document storage
- Powerful aggregation support
- Integration with Grafana
- Efficient log and telemetry analysis

This makes OpenSearch an ideal choice for cloud-native observability pipelines.

7.2 Index Design

The index design follows a domain-driven data modeling approach, where each lifecycle stage has a dedicated index.

Index Strategy

1. CI Build Metadata Index
Stores pipeline execution and image build information.

2. Deployment Metadata Index
Stores Kubernetes runtime deployment details.

3. Traceability Index
Stores correlated data linking CI and runtime layers.

Design Principles

- Separation of concerns
- Immutable document storage
- Time-based indexing
- Correlation through unique identifiers
- Optimized query performance
- Schema evolution flexibility

7.3 CI Metadata Index

The CI metadata index captures build pipeline telemetry generated by Jenkins.

Index Name

ci-build-metadata

Stored Data

- Build ID
- Job name
- Build number
- Git branch
- Commit metadata
- Author information
- Image name

- Image tag
- Image digest
- Build status
- Timestamp

Purpose

- Track software build lineage
- Identify image version origins
- Support release history analytics
- Enable commit-to-image traceability

7.4 Deployment Metadata Index

The deployment metadata index stores Kubernetes runtime deployment telemetry collected by the metadata collector.

Index Name

deployment-metadata

Stored Data

- Deployment name
- Namespace
- Pod name
- Image tag
- Image digest
- Node placement
- Replica count
- Deployment strategy
- Labels and annotations
- Runtime status
- Scheduling metadata
- Timestamp

Purpose

- Monitor runtime deployment state
- Enable workload health analytics
- Detect configuration drift
- Provide deployment audit trail

7.5 Traceability Index

The traceability index stores correlated CI and deployment data to provide end-to-end visibility across the delivery lifecycle.

Index Name

traceability-metadata

Correlated Fields

- Build ID
- Commit ID
- Author metadata
- Image tag
- Image digest
- Deployment metadata
- Pod runtime state
- Node scheduling data
- Deployment strategy
- Timestamp

Benefits

- End-to-end traceability
- Release impact analysis
- Root cause investigation
- Change intelligence
- DevOps observability
- Release governance

This index is the core foundation for traceability dashboards and analytics.

7.6 Querying & Data Correlation

OpenSearch enables advanced querying and aggregation for traceability analysis.

Correlation Strategy

Data is correlated using immutable identifiers:

Primary Correlation Keys

- Image digest
- Image tag
- Build ID

Secondary Correlation Keys

- Commit ID
- Deployment labels
- Pod owner references

Query Capabilities

- Release history tracking
- Deployment timeline analysis
- Author-based change tracking
- Pod health monitoring
- Impacted deployment identification
- Drift detection analytics

Query Integration

Queries are consumed by:

- Grafana dashboards
- Debugging workflows
- Incident response analysis
- Deployment validation pipelines

OpenSearch Data Layer Outcome

After implementing the OpenSearch data layer:

- Centralized metadata storage is achieved
- CI and runtime telemetry is unified
- End-to-end traceability becomes searchable
- Historical deployment intelligence is preserved
- Dashboard-driven observability is enabled
- Root cause analysis becomes faster
- Release governance improves

The OpenSearch data layer forms the observability intelligence engine of the architecture.

8. Observability & Monitoring

The observability layer provides deep visibility into CI/CD pipelines, Kubernetes deployments, runtime health, and release traceability through metrics collection, log analytics, and dashboard visualization.

This layer combines Prometheus for metrics collection, OpenSearch for metadata analytics, and Grafana for visualization, enabling full-stack DevOps observability across the software delivery lifecycle.

The monitoring architecture supports:

- Release traceability analytics
- Deployment health monitoring
- Runtime infrastructure visibility
- Resource utilization analysis
- Deployment impact detection
- Regression identification

8.1 Prometheus Metrics Collection

Prometheus acts as the primary metrics collection and time-series storage engine for Kubernetes and application-level telemetry.

Metrics Sources

- Kubernetes control plane metrics
- Node-level metrics via exporters
- Pod and container metrics
- Resource utilization metrics
- ArgoCD deployment metrics
- Application performance metrics

Key Prometheus Features

- Pull-based scraping model
- Label-based dimensional metrics
- Alerting and recording rules
- Time-series data retention
- Kubernetes service discovery

- Integration with Grafana

Observability Benefits

- Infrastructure health monitoring
- Performance trend analysis
- Capacity planning insights
- Real-time anomaly detection

8.2 Grafana Integration

Grafana is integrated as the unified visualization platform for metrics, metadata, and traceability analytics.

Data Sources Configured

- Prometheus (metrics)
- OpenSearch (traceability metadata)
- Kubernetes exporters

Integration Benefits

- Unified observability interface
- Cross-layer correlation (CI → CD → Runtime)
- Custom dashboard creation
- Real-time visualization
- Advanced query capabilities
- Alerting integration

Grafana serves as the primary user interface for operational insights and release intelligence.

8.3 Dashboard Architecture

The dashboard architecture follows a layered observability model:

Dashboard Layers

1. Release Observability Layer
Tracks CI/CD pipeline execution and release history.

2. Deployment Observability Layer
Monitors ArgoCD sync status and rollout behavior.

3. Runtime Observability Layer
Tracks Kubernetes resource health and pod behavior.

4. Traceability Layer
Provides commit-to-container correlation analytics.

Design Principles

- Correlation-driven visualization
- Time-based release analytics
- Drill-down capability
- Domain-based dashboards
- High signal-to-noise ratio
- Cloud-native observability patterns

8.4 Release History Dashboard

This dashboard provides visibility into historical releases and CI pipeline execution.

Dashboards Covered

- Application Release History Dashboard
- Jenkins Build Trace Dashboard
- Deployments Over Time Dashboard

Metrics & Insights

- Build frequency
- Release cadence
- Commit author analytics
- Image version history

- Deployment timelines
- Build success/failure patterns

Business Value

- Release governance
- Auditability
- Change intelligence
- Deployment trend analysis

8.5 Deployment Impact Dashboard

This dashboard identifies the operational impact of deployments and detects regressions.

Dashboards Covered

- Deployment Impact & Regression Dashboard
- ArgoCD Deployment Monitoring Dashboard

Insights Provided

- Rollout status
- Deployment health
- Sync success/failure analysis
- Regression detection
- Deployment duration analytics
- Drift detection

Business Value

- Faster root cause analysis
- Reduced MTTR
- Safer releases
- Deployment reliability tracking

8.6 Runtime Cluster Monitoring

Runtime dashboards provide real-time infrastructure and workload observability across the Kubernetes cluster.

Dashboards Covered

- Kubernetes Resource Utilization Dashboard
- Pod Health & Runtime Status Dashboard
- Code-To-Container Observability – Overview
- Code-To-Container Traceability Dashboard

Observability Coverage

- Node CPU and memory utilization
- Pod lifecycle monitoring
- Container restart patterns
- Resource saturation detection
- Scheduling distribution
- Runtime health analytics
- Commit-to-runtime correlation

Business Value

- Infrastructure reliability
- Performance monitoring
- Capacity planning
- Operational visibility
- Incident troubleshooting

Observability Layer Outcome

After implementing the observability layer:

- CI/CD pipelines become fully traceable
- Deployment health is continuously monitored
- Runtime infrastructure visibility is improved
- Release impact can be analyzed quickly
- Regression detection becomes proactive

- Root cause analysis is accelerated
- Cloud-native monitoring practices are established

This layer completes the end-to-end Code-to-Container Observability framework, transforming raw telemetry into actionable operational intelligence.

9. End-to-End Traceability Flow

The end-to-end traceability flow establishes a complete correlation between source code commits, CI builds, container images, Kubernetes deployments, runtime state, and visualization dashboards.

This traceability framework ensures that every running workload in the cluster can be mapped back to:

- Source commit
- Author
- Build pipeline execution
- Container image
- Deployment rollout
- Runtime health status

The architecture enables bidirectional traceability, allowing both forward tracking (commit → runtime) and reverse tracking (runtime → commit).

9.1 Commit to Build

The traceability lifecycle begins when a developer commits code changes to the Git repository.

Flow

1. Developer pushes commit to repository
2. Git webhook triggers Jenkins pipeline
3. Jenkins captures commit metadata
4. CI metadata JSON is generated
5. Metadata stored in OpenSearch

Metadata Captured

- Commit ID

- Author name and email
- Commit message
- Branch
- Timestamp
- Commit range
- Contributors list

Traceability Value

- Change auditability
- Contributor analytics
- Release attribution
- Deployment ownership tracking

9.2 Build to Image

During CI execution, the source code is transformed into a container image.

Flow

1. Jenkins pipeline triggers container build
2. Kaniko builds Docker image inside Kubernetes
3. Image pushed to container registry
4. Image digest extracted
5. Build metadata updated with digest

Metadata Captured

- Image name
- Image tag
- Image digest (immutable identifier)
- Build ID
- Build status
- Build timestamp

Traceability Value

- Immutable image tracking

- Supply chain integrity
- Reproducible builds
- Artifact lineage visibility

9.3 Image to Deployment

The GitOps layer deploys the image into Kubernetes using ArgoCD.

Flow

1. Image tag updated in GitOps repository
2. ArgoCD detects Git change
3. Application sync triggered
4. Kubernetes Deployment updated
5. Post-sync metadata collector captures deployment state

Metadata Captured

- Deployment name
- Namespace
- Replica configuration
- Image digest
- Deployment strategy
- Build ID correlation
- Commit ID correlation

Traceability Value

- GitOps-driven deployment auditability
- Deployment history tracking
- Drift detection capability
- Deployment lineage

9.4 Deployment to Runtime

Once deployed, Kubernetes schedules and runs workloads across cluster nodes.

Flow

1. ReplicaSet creates pods
2. Scheduler assigns pods to nodes
3. Containers start and pull image via digest
4. Readiness and liveness probes validate health
5. Collector CronJob periodically captures runtime state

Metadata Captured

- Pod name
- Node assignment
- Container image and digest
- Runtime status
- Replica health
- Scheduling distribution
- Restart counts

Traceability Value

- Runtime workload visibility
- Deployment health monitoring
- Infrastructure impact analysis
- Failure root cause mapping

9.5 Runtime to Visualization

All collected metadata and metrics are visualized through Grafana dashboards.

Flow

1. CI metadata stored in OpenSearch
2. Deployment metadata stored in OpenSearch
3. Correlation index merges CI and deployment data
4. Prometheus collects runtime metrics

5. Grafana visualizes traceability and observability data

Visualization Coverage

- Release history dashboards
- Deployment monitoring dashboards
- Runtime health dashboards
- Resource utilization dashboards
- Regression detection dashboards
- Code-to-container trace dashboards

Traceability Value

- Real-time release intelligence
- Operational insights
- Deployment impact visualization
- Faster troubleshooting
- Data-driven release governance

End-to-End Traceability Outcome

The implemented traceability framework enables:

- Complete commit-to-runtime visibility
- Immutable artifact tracking
- Deployment lineage analytics
- Automated metadata correlation
- Proactive regression detection
- Faster incident diagnosis
- Enterprise-grade DevOps observability

This flow transforms the delivery pipeline into a traceable, auditable, and observable software supply chain, aligning with modern cloud-native DevOps and GitOps best practices.

10. Security & Best Practices

Security is a foundational aspect of the Code-to-Container Observability architecture. The platform incorporates multiple DevSecOps best practices across the CI/CD pipeline, container supply chain, Kubernetes runtime, and GitOps deployment model.

The implementation focuses on:

- Immutable artifact traceability
- Secure container image build process
- GitOps-driven controlled deployments
- Namespace-based multi-tenancy isolation
- Secure secret and credential management

These practices ensure the platform adheres to modern cloud-native security principles while maintaining operational efficiency.

10.1 Immutable Image Strategy

The architecture follows an immutable infrastructure model, ensuring that every deployment references a uniquely identifiable container image.

Implementation

- Image tagging includes build number and commit ID
- Image digest extracted and stored during CI
- Kubernetes deployments reference image digest
- No mutable tags (e.g., latest) used in production

Benefits

- Prevents configuration drift
- Guarantees reproducible deployments
- Enables forensic traceability
- Improves rollback reliability
- Strengthens supply chain integrity

10.2 Secure Image Build (Kaniko)

Container images are built using Kaniko inside Kubernetes without requiring privileged Docker daemon access.

Security Advantages

- Rootless build execution
- No Docker socket exposure
- Reduced attack surface
- Secure registry authentication via secrets
- Ephemeral build pods

Best Practices Implemented

- Build isolation within dedicated pods
- Secret-based registry credential injection
- Minimal base image usage
- Layer caching without daemon privileges

This approach aligns with Kubernetes-native secure build practices.

10.3 GitOps Security Model

ArgoCD enforces a GitOps-based deployment model where Git acts as the single source of truth.

Security Characteristics

- Declarative infrastructure
- Version-controlled deployments
- Automated drift detection
- Controlled deployment approvals via Git workflows
- Self-healing ensures state consistency

Benefits

- Eliminates manual cluster mutations
- Provides full deployment audit trail

- Enables secure rollback through Git history
- Reduces operational risk
- Strengthens compliance posture

10.4 Namespace Isolation

Workloads and platform components are segregated using Kubernetes namespaces to achieve logical isolation and multi-tenancy boundaries.

Namespace Strategy

- CI components in dedicated namespace
- CD layer isolated within ArgoCD namespace
- Application workloads separated by environment
- Observability stack isolated in monitoring namespace
- Data layer isolated within observability namespace

Benefits

- Blast radius reduction
- Resource quota enforcement capability
- Improved RBAC scoping
- Operational boundary clarity
- Environment-level separation

10.5 Secret Management

Sensitive credentials required for CI/CD and registry access are securely stored using Kubernetes Secrets.

Secrets Used

- Docker registry credentials
- Git repository authentication
- ArgoCD credentials
- Service account tokens
- API endpoints and tokens

Best Practices Applied

- Secrets mounted as volumes rather than environment variables where possible

- Namespace-scoped secret isolation
- Principle of least privilege for service accounts
- Avoidance of plaintext secrets in Git repositories
- Secure secret injection into build pods

Future Enhancements (Recommended)

- External secret manager integration (e.g., HashiCorp Vault)
- Sealed Secrets for GitOps secret encryption
- Secret rotation automation
- Workload identity-based access

Security Posture Summary

The platform integrates security across the entire software delivery lifecycle by combining:

- Immutable artifact strategy
- Secure container build pipeline
- GitOps-controlled deployments
- Kubernetes isolation mechanisms
- Centralized secret management

This layered security model aligns with cloud-native DevSecOps principles and ensures the traceability platform remains secure, auditable, and production-ready.

11. Troubleshooting & Debugging

The Code-to-Container Observability platform includes multiple layers across CI, CD, container runtime, and metadata correlation. Effective troubleshooting strategies are essential to maintain reliability and ensure accurate traceability across the pipeline.

This section outlines common failure scenarios and systematic debugging approaches for each layer.

11.1 CI Failures

CI failures typically occur during source checkout, dependency installation, container build, or metadata generation.

Common Causes

- Git authentication failures
- Jenkins agent connectivity issues
- Kaniko build errors
- Dependency resolution failures
- Metadata file creation errors
- Registry push failures
- Image digest extraction failures

Debugging Approach

Step 1 — Validate Jenkins logs

Jenkins → Job → Console Output

Step 2 — Verify Kaniko build logs

- Confirm image successfully pushed
- Ensure digest appears in logs

Step 3 — Validate metadata JSON

cat build-metadata.json

Step 4 — Check registry authentication

kubectl get secret dockerhub-creds-configjson

Resolution Strategies

- Validate Jenkins credentials
- Confirm network access to registry

- Fix Dockerfile build errors
- Ensure jq/sed metadata injection steps work
- Retry build after fixing dependency issues

11.2 Image Pull Issues

Image pulls failures occur when Kubernetes cannot fetch container images from the registry.

Common Symptoms

- ErrImagePull
- ImagePullBackOff
- Failed to pull image
- Digest mismatch

Root Causes

- Incorrect image tag
- Missing digest
- Registry authentication issues
- Image not pushed successfully
- Network connectivity problems

Debugging Steps

kubectl describe pod <pod-name> -n <namespace>

Check events for:

- Unauthorized errors
- Not found errors
- TLS errors
- DNS issues

Fixes

- Validate image tag and digest
- Ensure registry secret exists
- Confirm image push success in CI logs
- Update deployment manifest with correct tag/digest

11.3 ArgoCD Sync Failures

ArgoCD sync issues occur when Git manifests cannot be applied to the cluster.

Common Causes

- Invalid Kubernetes manifests
- Image tag mismatch
- Git repository authentication errors
- RBAC permission issues
- Resource conflicts
- Namespace not found

Debugging Approach

ArgoCD UI

- Check application health
- Review sync status
- Inspect resource events

CLI / kubectl

kubectl get events -n <namespace>

Resolution Strategies

- Fix invalid YAML syntax
- Validate image tags
- Ensure namespace exists
- Re-sync application
- Review ArgoCD logs

kubectl logs deployment/argocd-server -n argocd

11.4 Kubernetes Pod Failures

Pod failures may arise due to scheduling, runtime, resource constraints, or container configuration errors.

Common Pod Failure Types

- CrashLoopBackOff
- Pending
- OOMKilled
- CreateContainerConfigError
- ContainerCreating delays

Debugging Steps

kubectl describe pod <pod-name> -n <namespace>

kubectl logs <pod-name> -n <namespace>

kubectl get events -n <namespace>

Root Causes

- Missing config or secrets
- Invalid environment variables
- Resource limits exceeded
- Node resource pressure
- DNS failures
- Application runtime errors

Resolution

- Adjust resource limits
- Fix environment configuration
- Validate secrets and config maps
- Check node availability
- Debug application logs

11.5 Metadata Mismatch Scenarios

Metadata mismatches can break traceability across CI, CD, and runtime layers.

Example Mismatch Cases

- Image tag mismatch between CI and deployment
- Digest mismatch between build and runtime
- Missing build_id in deployment metadata
- Collector unable to correlate build and deployment
- Timestamp skew
- Multiple pods using different image versions

Debugging Approach

Step 1 — Verify CI metadata

OpenSearch → ci-build-metadata index

Step 2 — Verify deployment metadata

OpenSearch → deployment-metadata index

Step 3 — Compare fields

- image_tag
- image_digest
- commit_id
- build_id
- deployment name
- pod name

Step 4 — Validate collector logs

kubectl logs job/<collector-job> -n <namespace>

Fix Strategies

- Ensure digest injected into metadata JSON
- Confirm deployment uses correct tag/digest
- Validate collector RBAC permissions
- Re-run collector job
- Fix correlation logic in aggregator

Troubleshooting Strategy Summary

The platform troubleshooting methodology follows layered debugging:

1. CI pipeline verification
2. Container registry validation
3. GitOps sync inspection
4. Kubernetes runtime debugging
5. Metadata correlation validation
6. OpenSearch indexing verification
7. Dashboard visualization consistency

This systematic approach ensures rapid root-cause identification while preserving traceability integrity.

13. Conclusion

The Code-to-Container Observability & Traceability Platform successfully demonstrates a modern cloud-native DevOps architecture that enables complete visibility across the software delivery lifecycle — from source code commit to runtime container execution.

By integrating Continuous Integration, GitOps-based Continuous Delivery, Kubernetes runtime orchestration, metadata aggregation, and observability dashboards, the platform provides a unified traceability model that bridges traditionally siloed stages of application delivery.

The implemented architecture ensures:

- Immutable and reproducible container builds using digest-based deployments
- Automated GitOps delivery with ArgoCD ensuring drift detection and self-healing
- Deployment metadata capture through custom collectors for runtime visibility
- Centralized storage of CI and deployment telemetry using OpenSearch
- End-to-end correlation across commit, build, image, deployment, and runtime layers
- Real-time monitoring and analytics using Prometheus and Grafana dashboards
- Enhanced debugging, auditability, and release impact analysis capabilities

This solution highlights best practices aligned with cloud architecture, DevOps, SRE, and platform engineering principles, including:

- Infrastructure as Code
- Immutable infrastructure strategy
- Event-driven deployment automation
- Observability-first system design
- GitOps-driven operational governance
- Metadata-driven traceability architecture

From a business and operational perspective, the platform improves release confidence, incident response speed, deployment transparency, and audit readiness, thereby enabling faster and safer software delivery.

Overall, the project demonstrates how a well-designed cloud-native ecosystem can provide deep operational intelligence and traceability, making it highly relevant for modern enterprise environments adopting Kubernetes, GitOps, and microservices architectures.