Containerization Incident Management Runbooks

Overview and Purpose

This document provides a comprehensive explanation of containerization technology, its purpose and function, and incident management runbooks for commonly encountered container failure modes. These runbooks are formatted consistently and include DevOps and Site Reliability Engineering (SRE) considerations to guide effective response in both development and production environments.

Containerization: Purpose and Function

Containerization is the process of packaging an application along with all its dependencies, configuration files, libraries, and runtime into a single unit called a container. Containers are lightweight, portable, and consistent across environments, making them ideal for microservices and scalable deployments.

Key Benefits

- Consistent behavior across development, staging, and production

- Efficient resource usage and fast startup times

- Simplified CI/CD and DevOps workflows

- Improved scalability and fault isolation

Why These Failure Modes Were Chosen

The failure modes selected are common in both cloud-native and hybrid container environments. They were chosen based on:
- Frequency in real-world DevOps and SRE incidents
- Potential impact on system availability and team productivity
- Representing a cross-section of infrastructure, application, and orchestration-level issues

Selected Failure Modes

- Container Fails to Start (Development)

- Container Crash in Kubernetes (CrashLoopBackOff)

- Image Pull Failure

- Network Partition

- Resource Exhaustion (CPU/Memory)

1. Development Environment Runbook: Container Fails to Start

Failure Mode

Container does not start successfully after build or during integration testing.

Symptoms

- Build pipeline fails at the container startup step

- Error logs or exit code immediately after container launch

- Missing environment variable or command not found errors

Diagnostics

- Check container logs: docker logs <container>

- Validate Dockerfile and ENTRYPOINT

- Inspect .env and docker-compose overrides

- Run container interactively using docker run -it <image> /bin/sh

Causes

- Incorrect or missing environment variables

- Invalid ENTRYPOINT or CMD configuration

- Dependency services not available locally

Actions

- Ensure environment variables are passed correctly

- Update or correct Dockerfile ENTRYPOINT/CMD

- Mock or stub external services

- Run container interactively to debug startup logic

Remediation

- Create and maintain .env.example files

- Add linting/validation steps to CI for Dockerfile and ENV

- Use standard base images and local test services

DevOps/SRE Notes

- Ensure developers validate configs with a local test harness

- Monitor recurring failure patterns in pre-prod pipelines

- Automate container health checks in CI

2. Production Environment Runbook: Container CrashLoopBackOff

Failure Mode

Container crashes repeatedly and enters a CrashLoopBackOff state in Kubernetes.

Symptoms

- Pod stuck in CrashLoopBackOff

- Increasing container restart count

- Application never becomes ready

Diagnostics

- kubectl logs <pod-name>

- kubectl describe pod <pod-name>

- Check health probe settings (liveness/readiness)

- Inspect config maps and secrets

Causes

- Application crashes due to bad configuration or dependency failure

- Invalid startup command or health check probe fails

- Missing secrets or runtime parameters

Actions

- Validate and fix configuration/environment values

- Patch deployment with correct health probes

- Temporarily disable failing liveness probe for debugging

- Roll back to a known good deployment

Remediation

- Include real probe checks in pre-production testing

- Use readiness probes to prevent traffic until ready

- Integrate crash-detection logic into CI/CD

DevOps/SRE Notes

- Set alerts for CrashLoopBackOff in Prometheus/Grafana

- Store previous deployment versions for rollback

- Include structured error handling in app for graceful exits

3. Production Environment Runbook: Image Pull Failure

Failure Mode

Container fails to start because the image cannot be pulled from the registry.

Symptoms

- Pod stuck in ImagePullBackOff or ErrImagePull

- kubectl describe pod shows pull failure

- Logs indicate unauthorized, not found, or timeout

Diagnostics

- Inspect pod events: kubectl describe pod

- Attempt manual pull from node using docker pull

- Check imagePullSecrets and registry credentials

Causes

- Incorrect image name or tag

- Private registry credentials missing or expired

- Image deleted from registry or DNS issue

Actions

- Correct the image tag and verify it exists

- Recreate or refresh imagePullSecret

- Test access from node to registry

Remediation
- Use immutable image tags (e.g., SHA digests)

- Centralize imagePullSecrets with automation

- Mirror critical images internally for reliability

DevOps/SRE Notes
- Monitor image registry availability

- Validate container image references during CI

- Implement fallback registries if mission-critical

4. Production Environment Runbook: Network Partition

Failure Mode
Pods cannot reach services or other pods due to networking issues.

Symptoms
- Connection timeouts

- 5xx errors from service mesh or sidecars

- DNS resolution failures

- Service endpoints missing or unresponsive

Diagnostics
- kubectl exec <pod> -- curl <target>

- Check CNI plugin status and logs

- Validate NetworkPolicy objects

- Inspect DNS: kubectl logs -n kube-system coredns-*

Causes
- Broken CNI plugin or misconfigured policy

- DNS outage or misrouting

- Cloud or hybrid networking issues

Actions
- Restart CoreDNS or network daemonset

- Relax NetworkPolicy for isolation testing

- Trace routes and ping service IPs or FQDNs

Remediation
- Validate network policies in staging before release

- Use service mesh (e.g., Istio) for routing observability

- Periodically test service-to-service communication

DevOps/SRE Notes
- Visualize pod-to-pod and service-to-service graphs

- Alert on DNS resolution failures or high 5xx rates

- Include basic curl/ping checks in health probes

5. Production Environment Runbook: Resource Exhaustion

Failure Mode
Container or node exceeds memory or CPU limits, resulting in OOMKilled or evictions.

Symptoms
- Pod status OOMKilled or Evicted

- Node status shows MemoryPressure

- Application slowdowns or unresponsive behavior

Diagnostics
- kubectl top pod and kubectl top node

- Check Prometheus/Grafana for memory/CPU trends

- Inspect logs around crash time

Causes
- Application memory leaks or spikes

- Lack of defined limits or requests

- Too many containers on under-provisioned nodes

Actions

- Define proper resource requests/limits in manifests

- Kill and restart leaking container

- Reschedule pods or scale up node pool

Remediation

- Run memory profiling and load testing

- Use HPA (Horizontal Pod Autoscaler) or VPA

- Reserve headroom in production clusters

DevOps/SRE Notes

- Monitor memory/cpu saturation trends

- Alert on container throttling or memory pressure

- Test under load in pre-production before releases