OpenShift Kubernetes Cluster

Database & Storage Services

Technical Documentation & Best Practices

On-Premises Environment

Document Version	1.0
Date	February 2026
Environment	On-Premises

Table of Contents

1. Database Containers

Database containers provide a containerized approach to deploying and managing database services within OpenShift Kubernetes clusters. This section covers the deployment, configuration, and management of relational and NoSQL databases in containerized environments.

1.1 Relational Databases (PostgreSQL)

Overview

PostgreSQL is a powerful, open-source relational database management system (RDBMS) that is highly recommended for enterprise applications running on OpenShift Kubernetes. It provides ACID compliance, robust transaction support, and excellent scalability for production workloads.

Deployment Architecture

- StatefulSet Deployment: Use OpenShift StatefulSets to ensure stable pod identities and persistent storage associations
- Persistent Volume (PV) Attachment: Provision and attach PVs to database pods for data persistence
- ConfigMaps and Secrets: Use ConfigMaps for configuration files and Secrets for sensitive credentials

- Service Exposure: Configure ClusterIP or headless services for internal connectivity

Configuration Best Practices

- Memory and CPU Allocation: Set appropriate resource requests and limits based on workload requirements
- Storage Class Selection: Choose appropriate storage classes (fast-ssd, standard) based on IOPS requirements
- Connection Pooling: Configure max connections to prevent resource exhaustion
- Replication Setup: Implement streaming replication for high availability and disaster recovery

Security Considerations

- Network Policies: Implement restrictive network policies to limit database access
- Encryption at Rest: Enable encrypted volumes using storage encryption capabilities
- TLS Connections: Configure TLS/SSL for secure database connections
- RBAC: Enforce role-based access control at the cluster level

Monitoring and Logging

- Prometheus Metrics: Integrate postgres_exporter for metrics collection
- ELK Stack Integration: Configure log aggregation using Elasticsearch, Logstash, and Kibana
- Alerting: Set up alerting for connection pool exhaustion and replication lag
- Health Checks: Configure liveness and readiness probes for automatic failure detection

1.2 NoSQL Database

Overview

NoSQL databases provide flexible schema design, horizontal scalability, and high performance for unstructured or semi-structured data. Common implementations include MongoDB, Redis, Cassandra, and DynamoDB-compatible solutions.

Implementation Options

- MongoDB: Document-based NoSQL database ideal for flexible schemas and rapid development
- Redis: In-memory data structure store for caching, sessions, and real-time analytics
- Cassandra: Distributed database designed for high availability and fault tolerance

Deployment Patterns

- Replica Sets (MongoDB): Deploy MongoDB replica sets with primary and secondary nodes for redundancy
- Sharding: Implement sharding for horizontal scalability across multiple database instances
- Operator-Based Deployment: Use community operators (MongoDB Community Kubernetes Operator) for simplified management

Data Consistency and Availability

- Write Concern Levels: Configure appropriate write concern (acknowledged, journaled, replicated)
- Read Preferences: Set read preferences (primary, primaryPreferred, secondary) based on requirements
- Eventual Consistency: Design applications to handle eventual consistency patterns

Performance Optimization

- Indexing Strategy: Create appropriate indexes to improve query performance
- Query Optimization: Analyze and optimize slow queries using execution plans
- Cache Layer Integration: Implement Redis as a caching layer for frequently accessed data
- Connection Optimization: Tune connection pool settings for optimal throughput

1.3 Database Connection Pooling

Overview

Connection pooling is a critical component for managing database connections efficiently in containerized environments. It reduces the overhead of establishing new connections and prevents connection exhaustion.

Connection Pool Architecture

- PgBouncer (PostgreSQL): Lightweight connection pooler for PostgreSQL databases
- Hikari CP (Java): Industry-standard JDBC connection pool for Java applications
- Application-Level Pooling: Built-in pooling mechanisms in application frameworks
- Proxy-Based Pooling: Use separate proxies deployed as sidecars in pods

Configuration Guidelines

- Pool Size Calculation: Set minimum and maximum pool sizes based on (worker_threads * 2) + spare_connections
- Connection Timeout: Configure appropriate timeout values to prevent hanging connections
- Idle Connection Validation: Enable periodic validation of idle connections
- Statement Caching: Enable statement caching to reduce parsing overhead

Monitoring and Troubleshooting

- Connection Count Metrics: Monitor active, idle, and pending connection counts
- Pool Exhaustion Alerts: Set up alerts when pool utilization exceeds thresholds
- Connection Leak Detection: Implement logging to identify connection leaks
- Performance Impact Analysis: Measure query execution time and connection overhead

2. Persistent Storage

Persistent storage is essential for maintaining data durability and availability in containerized environments. This section covers storage management strategies, backup and restore procedures, and data migration approaches.

2.1 Docker Volume Management

Volume Types in OpenShift

- EmptyDir: Temporary storage created with pod, deleted when pod terminates
- HostPath: Direct access to host filesystem (not recommended for production)
- PersistentVolumeClaim (PVC): Request for persistent storage backed by Persistent Volumes
- ConfigMap/Secret Volumes: Mount configuration and secret data

Persistent Volume Configuration

- Storage Class Selection: Choose from available storage classes (fast-ssd, standard-rwo, nfs, etc.)
- Access Modes: Configure appropriate access modes (ReadWriteOnce, ReadOnlyMany, ReadWriteMany)
- Capacity Planning: Request appropriate storage capacity with room for growth
- Reclaim Policy: Set appropriate reclaim policies (Retain, Delete, Recycle)

Volume Lifecycle Management

- Dynamic Provisioning: Use StorageClass for automatic PV provisioning
- Resize Operations: Expand PVC size when needed (if supported by storage backend)
- Volume Snapshots: Create point-in-time snapshots for backup and cloning
- Cleanup Policies: Remove unused PVCs and snapshots regularly

Performance Optimization

- IOPS Configuration: Tune storage IOPS for database workloads
- Throughput Tuning: Configure throughput limits and burst capabilities
- Caching Strategies: Leverage storage-level caching for frequently accessed data
- I/O Scheduling: Configure appropriate I/O scheduling policies

2.2 Backup & Restore Procedures

Backup Strategy Framework

- Recovery Point Objective (RPO): Define acceptable data loss window
- Recovery Time Objective (RTO): Define maximum allowed restoration time
- Backup Frequency: Schedule backups based on RPO requirements
- Retention Policy: Maintain backup retention schedules (daily, weekly, monthly, yearly)

Backup Methods

- Logical Backups: pg_dump for PostgreSQL, mongodump for MongoDB (human-readable, platform-independent)
- Physical Backups: PGBACKREST for PostgreSQL, filesystem snapshots (faster, compression-friendly)
- Incremental Backups: Backup only changed data since last backup
- Continuous Replication: Use streaming replication as backup mechanism

Backup Storage

- Local Storage: Fast backup/restore, vulnerable to cluster-wide failures
- Network Storage (NFS): Centralized storage with cross-node access
- Object Storage (S3/MinIO): Off-cluster backup for disaster recovery
- Geographic Replication: Replicate backups to remote locations

Restore Procedures

- Pre-Restore Validation: Verify backup integrity and completeness
- Point-in-Time Recovery: Restore to specific timestamp using transaction logs
- Selective Restoration: Restore specific databases or collections
- Validation After Restore: Verify data integrity and consistency

Backup Automation

- CronJobs: Schedule backup jobs using Kubernetes CronJobs
- Operators: Use specialized backup operators (e.g., Velero for cluster-wide backups)
- Monitoring: Track backup job status and send alerts on failures
- Testing: Regularly test backup integrity with restore drills

2.3 Data Migration Strategies

Migration Planning

- Assessment Phase: Document current system, workload patterns, and dependencies
- Data Volume Analysis: Calculate total data volume and estimate transfer time
- Schema Compatibility: Review schema differences and plan necessary transformations
- Validation Strategy: Define data validation rules and reconciliation procedures

Migration Approaches

- Big Bang Migration: Single-step migration with scheduled downtime
- Phased Migration: Gradual data transfer with parallel operation
- Dual-Write Strategy: Write to both systems during transition period
- Continuous Replication: Use database replication for minimal downtime

Migration Tools

- Native Tools: pg_dump/pg_restore (PostgreSQL), mongodump/mongorestore (MongoDB)
- Replication Tools: Logical replication, WAL-based replication
- ETL Tools: Apache Kafka, Debezium, DMS (Database Migration Service)
- Custom Scripts: Write application-specific migration scripts

Data Validation and Verification

- Row Count Comparison: Verify record counts match between source and target
- Checksum Validation: Compare checksums of migrated data
- Sample Data Verification: Test critical datasets for accuracy
- Query Result Comparison: Run identical queries on source and target

Rollback Planning

- Backup Strategy: Maintain full backup of source system during migration
- Fallback Procedures: Document step-by-step rollback procedures
- Application Rollback: Plan application configuration changes for quick rollback
- Runbook Preparation: Create detailed runbooks for all rollback scenarios

Conclusion

Database and storage services are critical components of enterprise OpenShift Kubernetes deployments. Organizations must carefully plan deployment strategies, implement robust backup and recovery procedures, and establish comprehensive monitoring systems. The best practices outlined in this document provide a foundation for building reliable, scalable, and secure database infrastructure in on-premises environments.

Continuous improvement, regular testing, and proactive monitoring are essential for maintaining high availability and preventing data loss. Organizations should regularly review these practices and adapt them to their specific requirements and evolving infrastructure needs.