Deployment Documentation

Overview

This document describes the deployment setup of the system, including its core services, infrastructure components, and runtime environment. The system is deployed as a set of loosely coupled services that communicate over well-defined interfaces. Containerization is used to ensure portability, reproducibility, and simplified deployment across environments.

The system can be deployed using the following command in the project root LiDARC:

docker compose up -d

Deployment Architecture

The system is deployed using a container-based approach. Each major component runs in its own container and can be deployed, scaled, and restarted independently.

Deployed components include:

- Frontend (Web Application)
- Backend (API & Orchestration Service)
- Message Broker (RabbitMQ)
- Worker Services (Metadata, Preprocessing, Comparison, Visualization)
- Object Storage (MinIO)
- Relational Database (PostgreSQL)
- In-Memory Store (Redis)

All components communicate over a shared virtual network.

Containerization Strategy

Each service is packaged as a Docker container with its own runtime environment and dependencies. This ensures consistent behavior across development and production systems.

Key characteristics:

- One service per container
- Stateless services where possible
- Configuration via environment variables

- Persistent data stored in external volumes

Service Configuration

1 Frontend

- Deployed as a static web application served via a web server container
- Communicates with the backend over HTTP(S)
- No persistent state stored locally

2 Backend

- Exposes REST and WebSocket endpoints
- Handles orchestration, and job coordination
- Publishes messages to the message broker
- Reads and writes metadata to PostgreSQL and Redis

3 Message Broker (RabbitMQ)

- Central asynchronous communication component
- Hosts queues for each processing stage
- Configured for durable queues to prevent message loss
- Configured through definitions.json

4 Worker Services

- Deployed as independent python containers
- Consume jobs from RabbitMQ
- Publish into queues if jobs finished
- Perform specialized processing tasks
- Scale horizontally based on workload (in Kubernetes deployment)

5 Storage Services

- **MinIO**: Stores uploaded and intermediate binary data (fully compatible with AWS S3)
- **PostgreSQL**: Stores persistent metadata and relational data
- **Redis**: Stores visualization data and caches files between workers

Persistent volumes are attached to storage containers to ensure data durability across restarts.

Environment Configuration

Configuration is managed via environment variables, including:

- Database connection details
- Message broker credentials
- Object storage access keys
- Service endpoints

Sensitive configuration values are not hard-coded and can be replaced per environment.

Scaling Strategy

- Worker services can be scaled independently by increasing the number of container instances depending on open jobs
    - Currently only employed for Preprocess worker
- Message queues buffer workload and prevent overload

Monitoring and Logging

- Each container writes logs to standard output
- Logs can be aggregated using container logging drivers

Kubernetes (K8S)

LiDARC is deployed on the EODC using K3S (lightweight Kubernetes) with a single cluster. Unlike other services, Minio is excluded due to the existing EODC S3.

Deployment is handled by a custom GitLab CI runner on the EODC, which runs `skaffold.yaml` (https://skaffold.dev/) with the "prod" profile. This process builds container images, pushes them to the defined registry, and K3S uses them to apply the StatefulSets (Postgres, Redis, etc.) and Deployments (Backend, Frontend, Workers).

The `preprocess-worker` autoscales from 2 to 6 pods based on the RabbitMQ queue load.

Currently, resource limits are undefined for Deployments, and PersistentVolumes for the Backend, Postgres, and Redis are limited to 20GB. The Frontend is exposed via an Ingress.

Summary

The container-based deployment approach ensures modularity, scalability, and resilience. By separating concerns across independent services and using asynchronous communication, the system can be deployed and operated efficiently while remaining flexible for future extensions.