Roboflow-like System — Full System Design Document

Version: 1.0
Prepared for: Rithvik
Date: 2025-12-01

Executive Summary

This document provides a complete system design for a Roboflow-like platform tailored for a single developer to build and operate, with clear technology choices, component responsibilities, data models, APIs, deployment plan, security posture, monitoring, scaling strategies, testing, and an actionable roadmap. The platform supports image ingestion, annotation (bounding boxes, polygons, points), augmentation & preprocessing, and multi-format export (COCO, YOLO, Pascal VOC).

Goals & Scope

Goals:
- Provide an internal-grade annotation and dataset pipeline that supports annotation, augmentation, preprocessing, and export formats.
- Be maintainable by a single developer and easily extensible.
- Secure enough for internal customers and junior staff use.

Scope (MVP):
- User auth (username/password, JWT)
- Dataset CRUD, image upload, thumbnail generation
- Annotation editor: bounding boxes, polygons, points
- Augmentation pipeline that also transforms annotations (albumentations)
- Preprocessing (resize, normalize, crop)
- Export: COCO, YOLO, Pascal VOC
- Deployment using Docker Compose (later production: Kubernetes)

High-Level Architecture

The system is a modular monolith for MVP with the following logical layers:
- Frontend: React SPA using React-Konva for canvas annotation.
- Backend API: FastAPI (Python) exposing REST endpoints.
- Storage: Image store (S3 or local disk), Metadata DB (SQL Server for production or Postgres/SQLite for MVP).
- Processing: Albumentations + OpenCV for augmentation & preprocessing; background worker (Celery/RQ) for heavy tasks.
- Exports: Server-side exporters that generate COCO/YOLO/VOC and package as ZIP.
- Deployment: Docker Compose for dev; Docker + Kubernetes for production (optional).

Technology Stack (Recommended)

Frontend:
- React (v18+)
- react-konva + konva (canvas rendering)
- use-image hook for image loading
- Tailwind CSS for rapid UI styling
- Zustand (or Redux) for lightweight state management
- Axios for HTTP

Backend:
- Python 3.11+
- FastAPI (ASGI) for REST endpoints

- Uvicorn / Gunicorn for production workers
- SQLAlchemy + Alembic for DB ORM and migrations
- Celery (with Redis) or RQ for background tasks
- PyJWT for authentication tokens
- Pillow / OpenCV / albumentations for image ops

Database & Storage:
- Primary choice for enterprise: Microsoft SQL Server (supported via pyodbc and sqlalchemy)
- Alternative (open-source): PostgreSQL (recommended for cloud portability)
- Image store: S3-compatible object storage (AWS S3, MinIO) or local disk for MVP
- Cache/broker: Redis (caching, celery broker, signed URLs)

CI / CD / Infra:
- Docker, Docker Compose (dev)
- GitHub Actions for CI
- Kubernetes (EKS/GKE/AKS) for production (optional)
- Nginx as reverse proxy; CloudFront/Cloud CDN in front of images (optional)

Monitoring & Observability:
- Prometheus + Grafana (metrics)
- Sentry for error tracking
- ELK / OpenSearch for logs (or hosted LogDNA)
- Health endpoints (liveness/readiness)

SQL Server: Integration & Considerations

SQL Server is fully supported via SQLAlchemy using pyodbc. Recommended connection string format:
- mssql+pyodbc://:@:/?driver=ODBC+Driver+18+for+SQL+Server

Driver installation:
- On Linux: install msodbcsql17 (or 18) packages and unixODBC
- On Windows: use ODBC Driver 17/18 package from Microsoft

Key considerations:
- Use connection pooling (SQLAlchemy) and configure pool_pre_ping for robustness.
- For migrations use Alembic with SQLAlchemy models or generate migration scripts via tools.
- Prefer UTC timestamps and set database collation and timezone handling explicitly.
- For backups: configure automated backups (native SQL Server backup or storage snapshots).

Component Responsibilities

Frontend (React):
- Image browser, upload UI, dataset management.
- Annotation canvas with tools: bbox, polygon, point, freehand optionally.
- Label management, keyboard shortcuts, undo/redo, zoom & pan.
- Trigger augmentation/export jobs and download results.

Backend (FastAPI):
- Auth, dataset, image, annotation CRUD APIs
- Image ingestion pipeline (validation, metadata extraction, thumbnail generation)
- Synchronous lightweight processing (resize, compress)
- Orchestration for long running tasks (Celery tasks to perform augmentation and export)
- Exporters for multiple formats, zipping and serving artifacts

Worker (Celery/RQ):
- Perform augmentation that modifies annotations
- Generate large exports (COCO/YOLO)

- Generate thumbnails, batch preprocessing
- Optionally run model-based auto-labeling (future)

Storage/DB:
- Store image files in object storage
- Store metadata and annotations in SQL Server (or Postgres)
- Store export artifacts and temporary files in object storage or local disk

Data Model (Simplified)

Tables (key columns):
- users: id, username, password_hash, role, created_at
- datasets: id, name, owner_id, metadata, created_at
- images: id, dataset_id, filename, path, width, height, sha256, uploaded_at
- annotations: id, image_id, label, shape_type, geometry (JSON), created_by, updated_at
- exports: id, dataset_id, format, path, created_at, status

Geometry formats (store as JSON):
- bbox: { "bbox": [x, y, w, h], "coords_unit": "px" }
- polygon: { "points": [x1,y1,x2,y2,...], "coords_unit": "px" }
Always store original image width/height in image row to enable normalization on export.

API Contracts (Essential Endpoints)

Auth:
- POST /api/auth/login -> { token }
- POST /api/auth/register -> { user }

Datasets:
- GET /api/datasets
- POST /api/datasets { name }
- GET /api/datasets/{id}
- DELETE /api/datasets/{id}

Images:
- POST /api/datasets/{id}/images (multipart) -> { image metadata }
- GET /api/datasets/{id}/images?page=&limit;=
- GET /api/images/{image_id} -> metadata
- GET /api/images/{image_id}/file -> binary image (serve via CDN or presigned URL)

Annotations:
- GET /api/images/{image_id}/annotations
- POST /api/images/{image_id}/annotations {label, shape_type, geometry}
- PUT /api/annotations/{id}
- DELETE /api/annotations/{id}

Augmentation / Preprocessing:
- POST /api/datasets/{id}/augment { spec } -> starts job, returns job_id (or sync returns ZIP)
- GET /api/jobs/{job_id} -> status, result link

Export:
- POST /api/datasets/{id}/export { format: coco|yolo|voc } -> job_id -> ZIP result
- GET /api/exports/{id} -> download link

Workflows (detailed)

Image Upload:
1. Frontend uploads via multipart POST.
2. Backend validates, stores file to object storage, creates DB row, generates thumbnail via worker.
3. Frontend receives image id and navigates to annotation page.

Annotate & Save:
1. Frontend loads image + annotations.
2. User edits; UI keeps local draft with undo stack.
3. Save action posts annotation(s) to backend; DB updated; frontend marks as saved.

Augment (train-ready):
1. Frontend posts augmentation pipeline spec.
2. Backend enqueues Celery job that:
- Loads image and annotation pairs
- Uses Albumentations with bbox_params and keypoint_params to transform annotations accordingly
- Writes augmented images and JSON annotations to storage
3. On completion, worker zips artifacts and stores export record; frontend polls job status and downloads ZIP

Export:
1. Export job queries DB for images+annotations
2. Exporter normalizes coordinates as required and writes format files
3. ZIP created and stored; download link returned

Deployment & Infrastructure

Development:
- Docker Compose (backend, frontend, db, redis)
- Local S3 emulator (MinIO) optional
- Run Uvicorn with reload for dev

Production (recommended):
- Kubernetes cluster (EKS/GKE/AKS)
- Backend: Deployment with HPA, service, ingress via Nginx Ingress Controller
- Workers: Separate Deployment/StatefulSet for Celery workers
- Database: Managed SQL Server (Azure SQL) or self-managed on VMs with automated backups
- Object Storage: S3 (AWS) or Azure Blob Storage
- CDN: CloudFront or Azure CDN for images
- Use Kubernetes CronJobs or a separate job system for scheduled tasks

Networking:
- TLS termination at ingress (cert-manager or Let's Encrypt)
- Internal network for DB and Redis (no public access)
- Use VPC/subnet isolation

Security, Authentication & Compliance

Authentication & Authorization:
- JWT tokens with short expiry + refresh tokens
- Role-based access (admin, annotator, viewer)
- Dataset-level ACLs (dataset.owner_id and collaborators list)

Data Security:
- Encrypt data at rest (S3 server-side encryption, DB TDE for SQL Server)
- Use HTTPS for all endpoints
- Signed URLs for image access where needed (presigned S3 URLs)

Operational Security:
- Rotate secrets via vault (Hashicorp Vault or cloud secret manager)
- IAM roles for services
- Network ACLs and security groups

Logging & Audit:
- Audit logs for annotation create/update/delete
- Access logs for API endpoints
- Retain logs according to policy

Monitoring, Metrics, and SLOs

Metrics:
- Request latency, 5xx rate, queue lengths (Celery), worker success/failure rates
- Disk usage for storage, DB connection counts

SLOs (suggested):
- API availability: 99.9%
- Annotation save latency: < 300ms for single annotation (fast path)
- Export job completion: depends on dataset size; measured and SLOs set per tier

Tools:
- Prometheus exporter for FastAPI (metrics)
- Grafana dashboards
- Sentry for exceptions

Testing Strategy

Unit Tests:
- Backend: pytest for API, DB layer, exporter logic
- Frontend: jest + react-testing-library for critical components

Integration Tests:
- End-to-end tests with Playwright or Cypress (upload -> annotate -> export)

Load Tests:
- Use k6 or Locust to simulate concurrent uploads and export generation

Security Tests:
- Dependency scanning (Snyk)
- Static analysis (Bandit for Python, ESLint for JS)

CI/CD Pipeline (example)

- GitHub Actions pipeline:
- Lint and unit tests (frontend & backend)
- Build docker images and push to registry on merge to main
- Deploy to staging (helm or kubectl apply)
- Run integration tests in staging
- Manual approval -> deploy to production

- Use image tags (SHA) and immutability
- Rollback strategy via Kubernetes deployments (previous replicaset)

Cost & Sizing Guidance

Small team / internal (MVP):
- Single t3.medium or equivalent VM running docker-compose for dev/testing.
- Storage: S3 with lifecycle rules; expect costs for storage & egress.
- SQL Server: use managed Azure SQL if on Azure (cost depends on tier). For cost-sensitive, use PostgreSQL.

Production (recommended minimum):
- 2 backend replicas (for HA)
- 2 worker replicas
- Managed DB (db.t3.medium equivalent or managed tier)
- S3 storage
- Redis (small instance)

Estimate monthly (rough):
- Compute: $100-500
- Storage: $10s-$100s depending on dataset sizes
- DB managed: $100-500+

Implementation Roadmap (12 weeks)

Phase 0 (Prep): 1 week
- Repo scaffold, CI baseline, infra IaC basics (docker-compose, k8s manifests)
Phase 1 (Core MVP): 3 weeks
- Backend: auth, datasets, image upload, images API, DB models
- Frontend: image list, upload, basic annotator (bbox), save/load
Phase 2 (Features): 4 weeks
- Polygon support, undo/redo, label management
- Exporters: YOLO, COCO, VOC
- Augmentation pipeline with annotation transforms (synchronous)
Phase 3 (Scale & Prod): 4 weeks
- Async workers (Celery), S3 integration, Postgres/SQL Server integration
- CI/CD, monitoring, security hardening, performance tuning

Delivery Checklist

- Repo scaffold (frontend/backend)
- API docs (OpenAPI)
- Annotator with bbox & polygon
- Exporters: YOLO/COCO/VOC
- Augmentation pipeline that handles annotations
- CI pipeline & Docker images
- Deployment manifests or docker-compose
- Monitoring & alerts (basic)
- Automated backups for DB & storage

Appendix: Example Config Snippets

FastAPI + SQL Server connection sample (SQLAlchemy):
DATABASE_URL =
"mssql+pyodbc://user:pass@host:1433/roboflowdb?driver=ODBC+Driver+17+for+SQL+Server"

Celery sample (Redis broker):
CELERY_BROKER_URL = "redis://redis:6379/0"
CELERY_RESULT_BACKEND = "redis://redis:6379/1"

Albumentations sample:

```
A.Compose([
A.HorizontalFlip(p=0.5),
A.Rotate(limit=15, p=0.5),
A.RandomBrightnessContrast(p=0.5),
], bbox_params=A.BboxParams(format="pascal_voc", label_fields=['category_ids']))
```