Video Generation Model Deployment

By Harshavardhan Kalalbandi (hkala002@ucr.edu)

Setup And Configuration

Initial Setup

Installed Docker and NVIDIA Container Runtime
Setup the initial system through Docker Compose

K3S Orchestration

Installed K3S and configured to listen on all network interfaces to allow access from external clients

Installed appropriate NVIDIA drivers and container toolkit to enable GPU workloads in K3S.

Created secrets (configuration) containing database credentials and connection strings before pod creation.

Application Build and Deployment

Created an automated deployment script that sets up K3s cluster, GPU support, builds the video generation app, and deploys all components with validation testing.

Build the video gen application Docker container and directly import to K3S container registry

Infrastructure Deployment

ConfigMap: application configuration
PersistentVolumeClaim (PVC): persistent storage volumes (for data storage)
Postgres: DB
Redis: Message Queue / caching

Application Deployment

api-deployment.yaml: FastAPI Server Deployment (default docker command *uvicorn*)

worker-deployment.yaml: Celery Worker deployment (run through k3s command override celery -A api.celery_app worker …)

api-service.yaml: Network access to the API

Validation and Testing

Check status through *kubectl rollout status* and verify deployment (after a timeout)
Test GPU access to the worker pod. *kubectl exec …. –nvidia-smi*

Networking: Port forward for local dev, and NodePort External IP access for external clients

K3S Manifest Files Details

Configuration Layer

Configmap.yaml: stores configs that all components share. Referenced by API and Worker using *configMapRef*.

Storage Layer

Creates a 50G persistent storage for video output files.

Pvc.yaml: *ReadWriteOnce* - can be mounted by multiple pods on the same node at once. Acts as storage between API (serves files) and Worker (creates files). Both mount at */app/output*.

Database Layer

Postgres.yaml: Single postgres-16 instance with 8G storage. Separate PVC. Exposes postgres:5432 for internal access only. Credentials pulled from videogen-pg-auth secret.
Strategy: recreate (Postgres security issues, locking - need to understand more)

API connects with CRUD operations. Worker updates job results.

Redis.yaml: redis-7 with persistence enabled –append-only**.** *Redis-master* connection string.
Strategy: rollingUpdate maxSurge 1 maxUnavailable 0

API queues video generation job, Worker (Celery) consumes it.

Application Layer

Api-deployment.yaml: User facing REST (FastAPI) for video generation requests.
Replicas: 3 (high availability and load distribution)
Health checks: */health* endpoint for load balancer checks
Strategy: Rolling Updates MS: 1 MU: 0 (0 downtime)

Worker-deployment.yaml: Video generation using GPU acceleration (Asynchronous job processing)
Replica: Single replica (GPU context issues with multiple Celery workers on a single node - need to understand more). Solo pool concurrency is 1
GPU Runtime: runtimeClassName: nvidia enables GPU access (K3S spec). Access 'all' GPUs
Strategy: Recreate (cannot have multiple nodes on a single GPU given Celery/GPU limitations)

api-service.yaml - Network Access

Internal Service (`videogen-api`): ClusterIP (internal only). Used by other services within the cluster

External Service (`videogen-external`): NodePort type exposes port 30080 externally. Allows direct access from outside the cluster

System Design and Components

FastAPI with uvicorn: accept requests, create jobs, serve status and file downloads. Stateless.
Pros: simple, fast, type-safe (pydantic), great for JSON + async IO.
Cons: Python isn't the absolute fastest at raw IO.
Future Improvements: Rate-limit per user/IP; enforce request body limits,

Celery Worker: dequeue jobs, run CogVideoX on H100, write MP4, update DB.

Pros: included retries, acks, time limits, visibility; large ecosystem
Trade-offs: not GPU-aware scheduling

Different Systems Evaluated and why Celery:

1. **Redis Queue (RQ):**
   Pros: Lightweight job queue by Redis. simple API; has job dependencies; supports retries and (via rq-scheduler)
   Cons: fewer orchestration primitives than Celery (routing, rate limits, time limits, chords)

2. **Ray / Ray Serve**
   Pros: Distributed execution with resource aware scheduling (e.g., `num_gpus`), retries, and autoscaling; Serve adds fractional GPU serving.
   Cons: I considered it overkill for a single-GPU, single-worker pipeline today. Setup seemed slightly more complex.

3. **Celery (chosen):**
   Pros: task queue with rich reliability semantics; works with Redis. **Built-in retries** and backoff; easy to declare *max_retries*. **Late acknowledgements** so tasks are re-delivered if a worker dies mid-run
   Cons: not GPU-aware scheduling (I serialize on one GPU worker to keep it safe)

Redis Broker: task queue between API and worker
Pros: simple to run on K3s; Celery-native; good for single-node.
Failure modes: broker down → no new work

Postgres Backend: Stores system of record. Job metadata, status, output paths, errors, timestamps
Pros: relational integrity, simple joins, easy migrations.
Cons: Right now single instance; requires care for long-running connections

File System Storage: (PVC, shared volume at `/app/output`)
Pros: trivial to mount for both pods in single-node K3s
Cons: ties you to node; capacity limits;
Future Improvements: Move to S3 or similar object store for production. Presigned S3 Urls for better performance

K3s (single node / single GPU): lightweight k8s distro orchestrating API, worker, DB, Redis, PVCs.

Cons: single point of failure; NodePort exposure; manual image push

Future Improvements: Snapshot DB; backup PVC; node monitoring. Switch NodePort → Ingress + TLS; use a registry + CI/CD (Helm/ArgoCD).

H100 GPU: (in worker pod via `runtimeClassName: nvidia`)
Accelerate CogVideoX inference
Future Improvements: If we add GPUs: 1 worker per GPU or model server that multiplexes safely.

Database Schema

PostgreSQL (primary store): holds job metadata (application table) and Celery task/result state (metadata tables).

Application table — video_generation_jobs: tracks each job's lifecycle and output path.

Columns: id (UUID, PK), celery_task_id (text, unique), user_id (text/UUID), prompt (text), status (pending|processing|completed|failed), output_file_path (text), file_size_bytes (bigint), video_duration_seconds (numeric/float), error_message (text, nullable), submitted_at (timestamp), updated_at (timestamp), completed_at (timestamp, nullable).

Cons: unbounded growth; slow queries without indexes.
Future Improvements: TTL + archival/GC (also delete files); indexes on celery_task_id, user_id, status, submitted_at; optional partitioning.

Celery system tables: celery_taskmeta (task state/result/traceback/date_done), celery_tasksetmeta (groups/chords).
Cons: not for long-term retention; duplicates some info with app table; can bloat DB.

Celery configuration: Broker = Redis; Result backend = PostgreSQL via db+postgresql://… (auto-creates Celery metadata tables).

Observability and Monitoring: Simple /health, /metrics endpoint (in-memory)
Pros: Simple setup
Cons: Not detailed enough, no historical data
Future Improvements: Move to Prometheus + Grafana or similar alternatives for robust solutions.

Video Generation Model

Used a simple small model zai-org/CogVideoX-2b to run the assignment.

Future Improvements: Need to look into best configuration for better generation and faster inference.