Text-to-Video API Challenge

Objective

Build a text-to-video API (over HTTP) that accepts a text prompt and generates a video using the open-source Genmo Mochi-1 Preview model.

System Environment

The solution is expected to run on:
• 8 × H100 GPUs
• ~18TB cumulative NVMe storage (not mounted by default)
• 2 CPUs (~124 cores)
• Kubernetes (K8s) cluster with a worker node & a control plane (accessible via kubectl only)

Requirements

1. Asynchronous Job Management
• Submit Job API: Accept a new video generation request & return a job ID immediately.
• Get Job Status API: Query job status (pending, processing, completed, failed).
• List Jobs API: Retrieve all submitted jobs, with filtering & pagination support.
• Get Output File API: Download the generated video once a job completes.

2. Video Generation
• Use Genmo Mochi-1 Preview exclusively for text-to-video generation.
• Ensure support for concurrent video processing across all available GPUs.

3. Scalability & Concurrency
• The system should process multiple requests concurrently.
• Workloads must be distributed across GPUs for optimal throughput.

4. User Interface
• Provide a basic frontend that allows:
• Submitting text prompts
• Tracking job status
• Downloading completed videos

5. Deployment
• Deploy the entire service on Kubernetes.
• Ensure high availability & efficient GPU utilization:
• Minimum 2 replicas of the video generation service for redundancy.
• At least 2 GPUs per replica allocated for processing.

Constraints

• No reliance on employer resources.
• Use of public/open-source tools & references is allowed.

Assumptions (allowed)

- Video length
- Resolution/quality
- API schema (must be JSON in/out & sensibly structured)

Expected Deliverables

- A working demo of the service (MVP first, then expanded features).
- Documentation of:
- Planning & design decisions
- Debugging & troubleshooting
- Options considered & tools used

Text-to-Video API – MVP & Production Design Document

Author: Keerthana Purushotham

Date: 2025-08-08

Purpose: This document outlines the design for a Kubernetes-deployed Text-to-Video API service using the Genmo Mochi-1 model to solve the problem of scalable, asynchronous, prompt-driven video generation.

1. Problem Statement (The Why)

Customers: Developers, researchers, and creative teams who need a scalable, programmatic text-to-video generation service.
Pain Points: Current video generation tools are often single-instance, blocking, and lack scalable API endpoints. Customers require asynchronous, concurrent, multi-GPU processing to handle high request volumes.
Urgency: Demand for generative AI video content is growing rapidly; this solution enables fast iteration and deployment.

2. Proposed Solution (The What)

Goal is to build an asynchronous text-to-video API using the Genmo Mochi-1 model hosted on an 8×H100 GPU Kubernetes worker node. The backend will handle job submission, tracking, and retrieval via JSON-based endpoints. A basic React-based frontend will allow prompt submission, status monitoring, and file downloads. The system will be deployed on Kubernetes (K8s) with GPU resource allocation, multi-replica redundancy, and horizontal scaling. Non-Goals: This MVP will not include advanced scheduling algorithms, RBAC, LLM-based load estimation, or zero-knowledge security layers - those are reserved for post-MVP.

3. Success Metrics (The How do we know it worked?)

1. MVP Success:
   - ≥95% job success rate.
   - P95 end-to-end latency ≤10 min.
   - Queue wait P95 ≤2 min.
   - Throughput ≥4 parallel jobs.
   - API availability ≥99% during demo.
   - 100% output artifact validity.
2. Production Success:
   - API availability ≥99.9%.
   - P95 latency ≤6 min, P99 ≤10 min.
   - Job retries <1%, DLQ <0.1%.
   - GPU utilization 70–90%.
   - Auth coverage 100%.
   - 0 critical CVEs in running images.

4. Open Questions & Assumptions

Considerations and Estimations:

1. Load visualization for video length vs prompt length:
    a. Img
2. Scale: Deployment patterns to prevent DoS by region, user-group etc with rollback, canary testing, retries, rate-limits etc.
3. Exceptions:
    a. Buggy prompt context from user – poor quality / lack of response
    b. Prompt work load exceeds resource allocation thresholds
    c. Infra security breaks -> retry / log relevant details
4. Are all tools compatible with potential upgrades and tool integrations without high refactoring costs? (ensure the OOPS aspects optimize computation without logical gaps or duplicate calculations
5. Concurrency handled by python orchestration over encapsulated, asynchronous Rust worker modules that run atomized request threads that close by virtue of Rust's memory/garbage management semantics that ensure that failed jobs do not break the validity of the session

Assumptions:
- Video length ≤10s for MVP.
- Resolution ≤768p.
- API structure is REST over JSON.
- External object storage (S3/MinIO) is available.

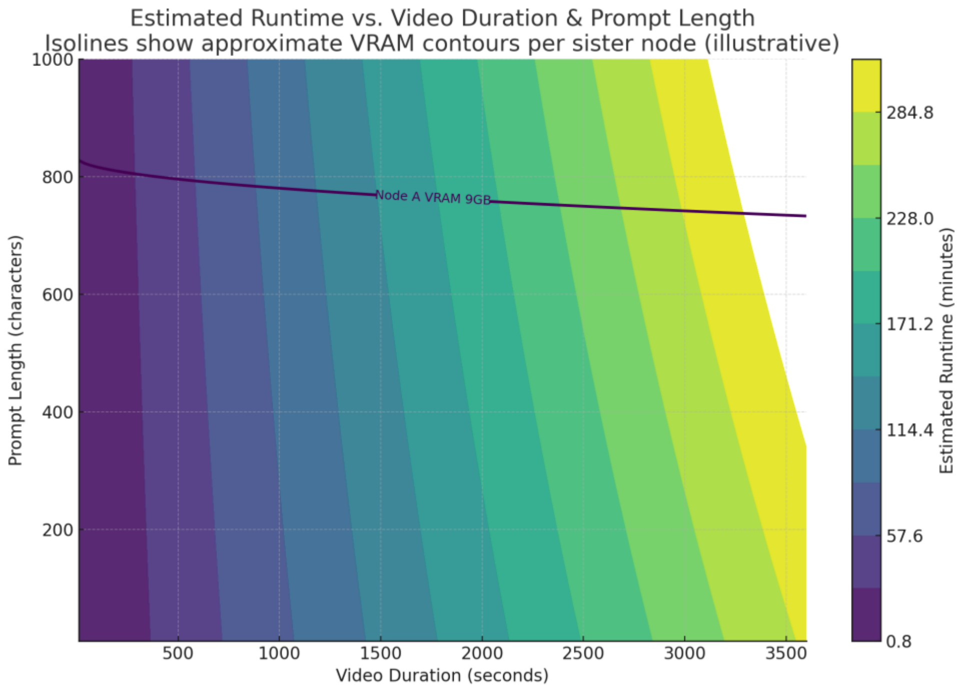
Open Questions:
- Will the control plane ELB DNS be stable for external access? (known to cause costly DoS across regions resulting in downtime and loss)
- Expected concurrency limits at demo vs production scale?
- Any constraints on video length/quality &/or time limits from stakeholders?
- Complex multi-part prompts requiring state management, explicit network hardening (over sandboxing) plus encryption.

Other Corner Cases:

1. Pre-signed URL misuse / role changes mid-job / token skew.
    - recover any state from persistent storage and retry
2. Hot-keying in rate limiter; retry storms; DLQ loops.
    - (fastapi-limiter &/or redis). Link: stackoverflow & documentation for the caveat described below

- NOTE: FastAPI doesn't natively support this, but it's possible with a few libraries such the ones below, but will usually require some sort of database backing(redis, memcached, etc), although slowapi has a memory fallback in case of no database.
- https://pypi.org/project/fastapi-limiter/
- https://pypi.org/project/slowapi/

  ** In order to use fastapi-limiter, as seen in their documentation: You will need a running Redis for this to work.



3. Starvation of long jobs; convoy effects; head-of-line blocking.
    - dedicated long-task exception handler node with critical Alarm if task still fails;
    - some job length estimator module with dashboard tracking accuracy growth (for “is this potentially a long task based on context, linguistics, user/env metrics? Yes/No”
    - ….alarms at sev-2.5 if accuracy (true positives and negatives) consistently falls over time (false values are increasing. Check estimator logic), alarm at sev 2 if it falls immediately)
4. Log PII, high-cardinality labels; sampling hiding tail latency.
    - outliers and adversarial samples. Check if data corruption occurred via access/edit-logs, stack trace, etc to ensure no security exploitation broke the ML model.
5. Split-brain deploys across regions; partial rollbacks.
    - rollout
    - set
    - Scale
    - Autoscale
    - Auth
6. Model nondeterminism vs “golden” tests; flaky perf from noisy neighbors.

- Validation – check if things are right
- Sanity – ensure wrong things can't happen
- Unit – cover as many test cases, corner cases and outlier cases
- Integration – check if cross-tool features are ok
- Regression – ensure that new changes don't break existing functionality

5. Stakeholders & Next Steps

Key Stakeholders:
- Users: API consumers (developers, researchers)
- Tech Support: Handles incidents & outages
- Developers: Build & maintain backend/frontend
- Vendor Organization: Voltage Park infrastructure team
- Network Peers: Any API gateway/CDN providers
- Node Cluster: K8s worker node (8×H100)
- Control Plane: Managed by vendor, not directly accessible

Next Steps:
- Deploy initial API & worker pods on K8s.
- Implement asynchronous endpoints.
- **Finalize v1 prod features critical to release for enterprise scale.**
- Build basic React frontend.
- Integrate Prometheus/Grafana monitoring.
- Conduct load test for target throughput.
- Prepare for demo & stakeholder review.

Appendix: team input – Vote and choose v1 release features for prod.

Post-MVP Features – Prioritization Matrix

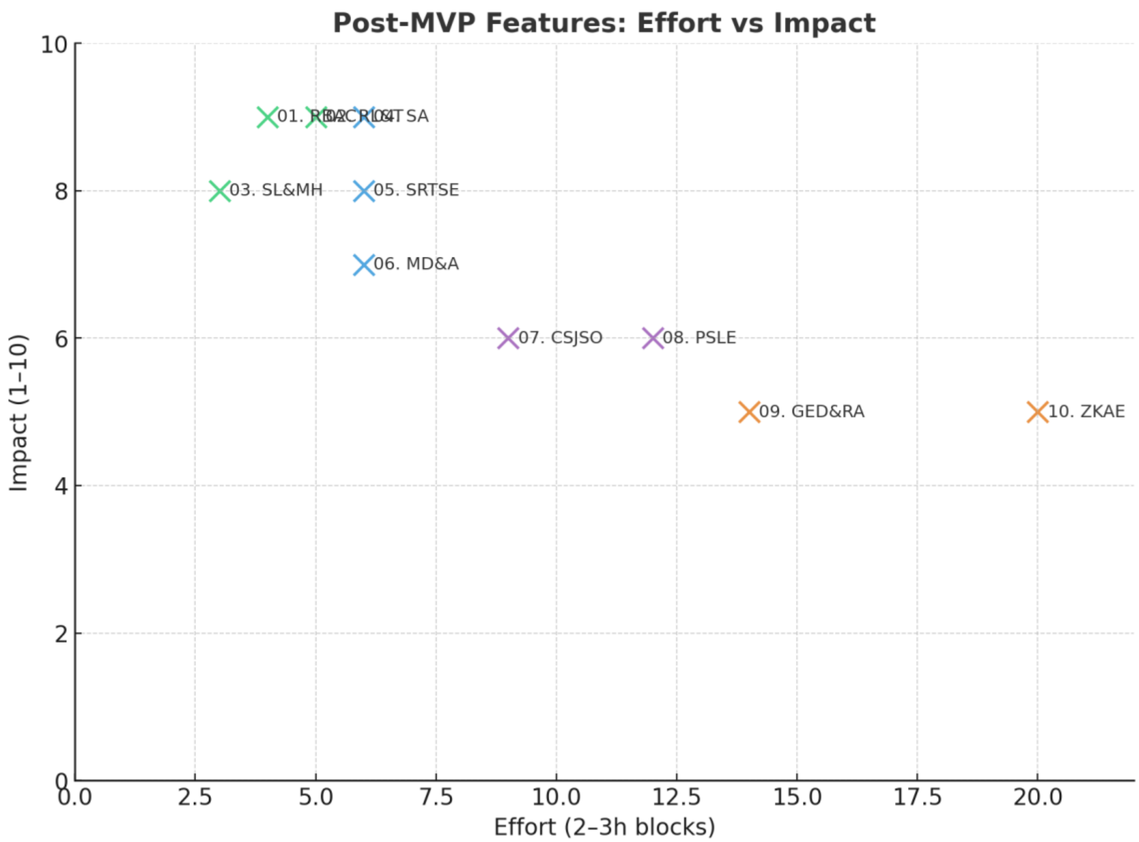
Purpose: Enable the team to quickly assess, vote, and sequence high-impact improvements after the MVP launch.
Voting Format: ✅ = must-have next, 🕒 = later, ❌ = not now.

#	Acronym	Full Name	Stakeholders	Dependencies	Corner Cases	Effort (2–3h blocks)	Priority
☐01	RBAC	Role-Based API Access Control	user, dev, vendor org, control plane	Auth (04)	service-to-service calls, URL misuse	3–4	✅
☐02	RL&T	Rate Limiting & Throttling	user, tech-support, dev, vendor org, network peers	RBAC (01), Redis, Ingress	retry storms, hot keys	3–5	✅
☐03	SL&MH	Structured Logging & Monitoring Hooks	tech-support, dev, vendor org	none	PII leaks, log storms	2–4	✅
☐04	SA	Secure Authentication (JWT + OIDC)	user, dev, vendor org, control plane	IdP, RBAC	token expiry, revocation	4–6	✅
☐05	SRTSE	Stress & Regression Test Suite Expansion	dev, vendor org, node cluster	CI w/ GPU	flaky perf, model variance	4–6	🕒
☐06	MD&A	Minimal Dashboard & Alarms	user, tech-support, dev, vendor org	SL&MH (03)	alert fatigue, tenant leaks	4–6	🕒
☐07	CSJSO	Case-Specific Job Scheduling Optimizations	user, dev, vendor org, node cluster	job metadata	starvation, convoy	6–9	🕒
☐08	PSLE	Prompt-Specific Load Estimation	user, dev, vendor org	CSJSO (07)	bias, cold-start	8–12	❌
☐09	GED&RA	Global Edge Deployment & Rollback Automation	user, tech-support, dev, vendor org, control plane	MD&A (06), SL&MH (03)	split-brain, data residency	10–16	❌
☐10	ZKAE	Zero-Knowledge Architecture Expansion	user, vendor org, dev	KMS, TEE	key loss, debug blind	14–22	❌

Effort/Impact Visualization

> Documentation > Design.doc
, 1 second ago | 1 author (You)

#	Acronym	Full Feature Name	Stakeholders	Dependencies
01	RBAC	Role-Based API Access Control	user, dev, vendor org, control plane	Auth (04)
02	RL\&T	Rate Limiting & Throttling	user, tech-support, dev, vendor, network peers	RBAC (01), Redis, Ingress
03	SL\&MH	Structured Logging & Monitoring Hooks	tech-support, dev, vendor org	none
04	SA	Secure Authentication (JWT + OIDC)	user, dev, vendor org, control plane	IdP, RBAC
05	SRTSE	Stress & Regression Test Suite Expansion	dev, vendor org, node cluster	CI w/ GPU
06	MD\&A	Minimal Dashboard & Alarms	user, tech-support, dev, vendor org	SL\&MH (03)
07	CSJSO	Case-Specific Job Scheduling Optimizations	user, dev, vendor org, node cluster	job metadata
08	PSLE	Prompt-Specific Load Estimation	user, dev, vendor org	CSJSO (07)
09	GED\&RA	Global Edge Deployment & Rollback Automation	user, tech-support, dev, vendor, control plane	MD\&A (06), SL\&MH (03)
10	ZKAE	Zero-Knowledge Architecture Expansion	user, vendor org, dev	KMS, TEE



I’ve mapped each feature into an **Effort vs Impact** matrix so it’s easy to see trade-offs:

- **Green** = Immediate High-Impact / Low Effort (01, 02, 03)
- **Blue** = High-Impact / Medium Effort (04, 05, 06)
- **Purple** = Medium Impact / Higher Effort (07, 08)
- **Orange** = Niche Impact / High Effort (09, 10)

Single Node Text-to-Video API

-> see design document for more info

Overview

This project implements a **Kubernetes-deployed asynchronous text-to-video API** using **FastAPI**, **Celery**, **Redis**, and GPU-bound **Rust workers** orchestrated by Python.
It satisfies the Voltage Park take-home assignment for a single node with **8× H100 GPUs**.

Core idea:

- **FastAPI** serves as the HTTP API gateway.
- **Celery** manages job orchestration and retries.
- **Redis** acts as the broker/result backend and status cache.
- **Rust GPU workers** run the `genmo/mochi-1-preview` model.
- **PVC** provides fast local scratch space for video generation.
- **MinIO** stores completed videos for retrieval via presigned URLs.

High-Level Architecture

- See README

Components

1. FastAPI — API Gateway

- Async, high-performance Python web framework.
- Handles:
  - **POST** `/jobs` → Create a job, return `job_id`
  - **GET** `/jobs/{id}` → Status (`pending`, `processing`, `completed`, `failed`)
  - **GET** `/jobs` → Paginated job list
  - **GET** `/jobs/{id}/result` → Presigned download URL (MinIO) or direct MP4

Scaling:

- Stateless → scale horizontally (HPA) by CPU/memory or RPS.
- No GPU allocation.

---

2. Celery — Job Orchestration

- Mature Python task queue framework.
- Handles retries, routing, and concurrency limits.
- Queue separation:
  - `short-low` — quick jobs

    - `long-high` — heavy/long-running jobs

Scaling:

- HPA based on `queue_depth`.
- Workers pinned to GPU resources via Kubernetes.

---

3. Redis — Broker + Cache

- Fast, in-memory datastore.
- Stores:
    - Celery task queue
    - Task results
    - Job status cache

Scaling:

- Single instance with AOF persistence for MVP.
- Optional Redis Cluster for high concurrency.

---

4. Rust GPU Workers

- Long-running processes in GPU pods.
- Host the mochi-1 model for text-to-video.
- Python task (Celery) calls Rust via **gRPC** (preferred) or **PyO3 FFI**.

GPU Binding:

```
resources:
  limits:
    nvidia.com/gpu: 2
```

- Each worker pod processes tasks sequentially per GPU to avoid memory contention.

5. PVC — Persistent Scratch Storage

- **Local NVMe-backed** (or fast network) persistent volume for:
    - Temporary frames
    - Partial encodes
- Prevents data loss on pod restarts.

---

6. MinIO — Artifact Storage

- **S3-compatible** object store deployed in the cluster.
- Stores final videos.

- API returns **presigned URLs** so frontend can download directly without routing through the API pods.

---

gRPC API Specification (Python ↔ Rust Workers)

The Rust workers expose a **gRPC service** consumed by the Python Celery tasks. This keeps orchestration in Python while GPU-heavy work stays in Rust.

Proto file: `video_generator.proto`

```
syntax = "proto3";

package video;

service VideoGenerator {
  // Submit a text-to-video generation request
  rpc Generate (GenerateRequest) returns (GenerateResponse);

  // Check job status (optional if Celery handles status)
  rpc GetStatus (StatusRequest) returns (StatusResponse);
}

message GenerateRequest {
  string job_id = 1;              // UUID from Celery
  string prompt = 2;              // User's text prompt
  int32 resolution_width = 3;     // e.g., 1920
  int32 resolution_height = 4;    // e.g., 1080
  int32 duration_seconds = 5;     // Clip length
  string quality = 6;             // "low", "medium", "high"
}

message GenerateResponse {
  bool accepted = 1;              // Whether the worker accepted the job
  string message = 2;             // Info or reason for rejection
}

message StatusRequest {
  string job_id = 1;
}

message StatusResponse {
  string job_id = 1;
  string state = 2;                // "pending", "processing",
"completed", "failed"
  string output_path = 3;          // Path on PVC or S3 URL
  string error_message = 4;        // If failed
}

# gRPC API & Kubernetes Deployment Specification (Python ↔ Rust Workers)
```

````
**Goal:** Keep orchestration in Python (FastAPI + Celery) while
offloading GPU-heavy work to Rust workers exposed via **gRPC**. Workers
run in the same pod as the GPU process (or alone) and write
intermediates to a PVC, then upload final artifacts to MinIO.

---

## Architecture Overview

```text
Clients ──(HTTPS)──▶ FastAPI (video-api)
                    │
                    ├─▶ Redis (task queue / status cache)
                    │
                    ├─▶ Celery worker (Python orchestrator) ──(gRPC)──▶
Rust worker(s)
                    │                                               │
                    │                                               ├─▶
NVIDIA GPU
                    │                                               ├─▶
PVC scratch (RWO/RWX)
                    │                                               └─▶
MinIO (S3-compatible)
                    │
                    └─▶ API serves status & presigned URL
````

Flow:

1. FastAPI enqueues a Celery job with `job_id` and request payload.
2. Celery task dials the Rust worker **gRPC** service (`Generate`) inside the cluster (same namespace).
3. Rust worker loads the model (on first call), generates video to a **PVC scratch** path.
4. Worker uploads final video to **MinIO** and records job state (Redis update or callback to API).
5. API exposes status + **presigned URL** to clients.

Advantages:

- **Language isolation:** Rust can evolve independently of orchestration.
- **Clear contracts:** gRPC schema is the source of truth.
- **Scales cleanly:** Add workers; API/Celery remain unchanged.

---

gRPC API

`.proto` (tonic-compatible)

```
syntax = "proto3";
package video;
```
```

```
service Generator {
  rpc Generate(GenerateRequest) returns (stream GenerateEvent);
  rpc GetJob(GetJobRequest) returns (GetJobResponse);
}

message GenerateRequest {
  string job_id = 1;                // Idempotency key
  string prompt = 2;                // Text or JSON-encoded params
  string model = 3;                 // e.g. "mochi-1-preview"
  string scratch_dir = 4;           // Mounted PVC path (pod-local)
  map<string,string> options = 5;   // width, height, fps, seed,
duration, etc.
}

message GenerateEvent {
  oneof event {
    Progress progress = 1;
    Result result = 2;
    Error error = 3;
  }
}

message Progress {
  string job_id = 1;
  int32 step = 2;
  int32 total_steps = 3;
  string message = 4;
  double gpu_util = 5;              // Optional, for live UX/metrics
}

message Result {
  string job_id = 1;
  string artifact_path = 2;         // Final local path before upload
  string s3_url = 3;                // s3://bucket/key
  string etag = 4;
  int64 bytes = 5;
}

message Error {
  string job_id = 1;
  int32 code = 2;                   // Application error codes
  string message = 3;
}

message GetJobRequest { string job_id = 1; }
message GetJobResponse {
  enum Status { UNKNOWN = 0; QUEUED = 1; RUNNING = 2; SUCCEEDED = 3;
FAILED = 4; }
  string job_id = 1;
  Status status = 2;
  string s3_url = 3;                // filled if SUCCEEDED
  string error = 4;                 // filled if FAILED
```

```
  int32 progress = 5;              // 0..100
}
```

Rust (tonic) – service skeleton

```
use tonic::{Request, Response, Status};
use tonic::codegen::futures_core::Stream;
use tokio_stream::wrappers::ReceiverStream;

pub struct GeneratorSvc { /* state: model cache, redis, minio, etc. */ }

#[tonic::async_trait]
impl video::generator_server::Generator for GeneratorSvc {
    type GenerateStream = ReceiverStream<Result<video::GenerateEvent,
Status>>;

    async fn Generate(
        &self,
        req: Request<video::GenerateRequest>,
    ) -> Result<Response<Self::GenerateStream>, Status> {
        // load or get cached model
        // spawn task producing Progress/Result/Error events
        // stream back via mpsc channel
        todo!()
    }

    async fn GetJob(
        &self,
        req: Request<video::GetJobRequest>,
    ) -> Result<Response<video::GetJobResponse>, Status> {
        // read from Redis/state store
        todo!()
    }
}
```

Python client (Celery task)

```
import grpc
from video_pb2 import GenerateRequest
from video_pb2_grpc import GeneratorStub

@celery.task(bind=True, acks_late=True)
def generate_task(self, job_id: str, prompt: str, options: dict):
    chan = grpc.insecure_channel("video-worker:50051")  # or mTLS
    stub = GeneratorStub(chan)
    req = GenerateRequest(job_id=job_id, prompt=prompt, model="mochi-1-
preview", scratch_dir="/scratch", options=options)
```

```
    for event in stub.Generate(req):
        if event.HasField("progress"):
            redis.hset(f"job:{job_id}", mapping={"status": "RUNNING",
"progress": pct(event.progress)})
        elif event.HasField("result"):
            # upload may be done in Rust; optionally verify ETag
            redis.hset(f"job:{job_id}", mapping={"status": "SUCCEEDED",
"s3_url": event.result.s3_url})
        elif event.HasField("error"):
            redis.hset(f"job:{job_id}", mapping={"status": "FAILED",
"error": event.error.message})
            raise Exception(event.error.message)
```

API semantics:

- **Idempotency:** `job_id` is unique; repeated `Generate(job_id)` should resume or no-op.
- **Deadlines/timeouts:** client sets gRPC deadline; server respects and continues in background if desired.
- **Backpressure:** streaming progress avoids long-polling and supports cancellation.
- **Observability:** include step counts and GPU util in events for metrics.

Kubernetes

> The examples assume namespace `video` and GPU scheduling via the NVIDIA Device Plugin/Operator.

Namespace & RBAC

```yaml
aPiVersion: v1
kind: Namespace
metadata:
  name: video
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: runtime
  namespace: video
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: runtime-role
  namespace: video
rules:
  - apiGroups: [""]
    resources: ["pods", "pods/log", "secrets", "configmaps"]
    verbs: ["get", "list", "watch"]
---
```

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: runtime-rb
  namespace: video
subjects:
  - kind: ServiceAccount
    name: runtime
    namespace: video
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: runtime-role
```

Config & Secrets

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: s3-credentials
  namespace: video
stringData:
  AWS_ACCESS_KEY_ID: "minio"
  AWS_SECRET_ACCESS_KEY: "<redacted>"
  AWS_S3_ENDPOINT: "http://minio.video.svc.cluster.local:9000"
  AWS_S3_BUCKET: "videos"
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
  namespace: video
data:
  REDIS_URL: "redis://redis-master.video.svc.cluster.local:6379/0"
  SCRATCH_DIR: "/scratch"
  MODEL_NAME: "mochi-1-preview"
```

PVC (scratch)

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: video-scratch
  namespace: video
spec:
  accessModes: ["ReadWriteOnce"]        # Use RWX if multiple pods read
same path
  resources:
```

```yaml
    requests:
      storage: 500Gi
  storageClassName: fast-nvme
```

API Deployment + Service + HPA

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: video-api
  namespace: video
spec:
  replicas: 2
  selector:
    matchLabels:
      app: video-api
  template:
    metadata:
      labels:
        app: video-api
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/port: "8000"
    spec:
      serviceAccountName: runtime
      containers:
        - name: api
          image: your-dockerhub/video-api:latest
          ports:
            - containerPort: 8000
          envFrom:
            - configMapRef: { name: app-config }
            - secretRef: { name: s3-credentials }
          readinessProbe:
            httpGet: { path: /healthz, port: 8000 }
            periodSeconds: 5
          livenessProbe:
            httpGet: { path: /livez, port: 8000 }
            periodSeconds: 10
---
apiVersion: v1
kind: Service
metadata:
  name: video-api
  namespace: video
spec:
  selector:
    app: video-api
  ports:
    - port: 80
```

```yaml
      targetPort: 8000
      protocol: TCP
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: video-api
  namespace: video
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: video-api
  minReplicas: 2
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 70
```

Rust Worker Deployment + Service + PDB

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: video-worker
  namespace: video
spec:
  replicas: 4
  selector:
    matchLabels:
      app: video-worker
  template:
    metadata:
      labels:
        app: video-worker
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/port: "9464"      # if exposing metrics
    spec:
      serviceAccountName: runtime
      nodeSelector:
        nvidia.com/gpu.present: "true"
      tolerations:
        - key: nvidia.com/gpu
          operator: Exists
          effect: NoSchedule
```

```yaml
      volumes:
        - name: scratch
          persistentVolumeClaim:
            claimName: video-scratch
      containers:
        - name: worker
          image: your-dockerhub/video-worker:latest
          ports:
            - containerPort: 50051  # gRPC
          resources:
            limits:
              nvidia.com/gpu: 2
            requests:
              cpu: "2"
              memory: "8Gi"
          volumeMounts:
            - name: scratch
              mountPath: /scratch
          envFrom:
            - configMapRef: { name: app-config }
            - secretRef: { name: s3-credentials }
          readinessProbe:
            tcpSocket: { port: 50051 }
            periodSeconds: 5
          livenessProbe:
            tcpSocket: { port: 50051 }
            periodSeconds: 10
---
apiVersion: v1
kind: Service
metadata:
  name: video-worker
  namespace: video
spec:
  selector:
    app: video-worker
  ports:
    - port: 50051
      targetPort: 50051
      protocol: TCP
---
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: video-worker-pdb
  namespace: video
spec:
  minAvailable: 75%
  selector:
    matchLabels:
      app: video-worker
```

Network Policies (lock down traffic)

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: api-allow
  namespace: video
spec:
  podSelector:
    matchLabels: { app: video-api }
  ingress:
    - from:
        - podSelector: {}             # adjust for ingress controller
      ports:
        - protocol: TCP
          port: 80
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: worker-allow
  namespace: video
spec:
  podSelector:
    matchLabels: { app: video-worker }
  ingress:
    - from:
        - podSelector:
            matchLabels: { app: video-api }
        - podSelector:
            matchLabels: { app: celery }   # if separate Celery
deployment
      ports:
        - protocol: TCP
          port: 50051
```

Deployment Plan (cluster-level)

1. **Verify cluster access**

```
kubectl --kubeconfig kubeconfig.yml get nodes
```

2. **Install NVIDIA GPU Operator + device plugin**

```
helm repo add nvidia https://helm.ngc.nvidia.com/nvidia
helm repo update
```

```
helm install gpu-operator nvidia/gpu-operator -n gpu-operator --
create-namespace
```

3. **Install Redis** (Bitnami)

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm install redis bitnami/redis -n video --create-namespace \
  --set architecture=replication --set auth.enabled=false
```

4. **Deploy API**

```
kubectl apply -f k8s/api.yaml
```

5. **Deploy Workers**

```
kubectl apply -f k8s/worker.yaml
```

6. **Create PVC**

```
kubectl apply -f k8s/pvc.yaml
```

7. **Deploy MinIO**

```
helm install minio bitnami/minio -n video \
  --set resources.requests.memory=1Gi \
  --set mode=standalone \
  --set auth.rootUser=minio,auth.rootPassword=<redacted>
```

Scaling Guidelines

Scenario	API replicas	Worker pods	GPUs/pod	Storage Strategy
Many small jobs	4	4	2	PVC → MinIO
Heavy long jobs	2	4	2 (1 task/GPU)	PVC → MinIO
Mixed	HPA on RPS	Separate queues	1–2	PVC per worker
Very large outputs	2–3	4	2	Direct stream to MinIO

Notes:

- Consider **one task per GPU** to avoid context thrash.
- Use **separate Celery queues** (e.g., `short`, `long`) mapped to different workers.
- For multi-reader workflows, prefer **RWX** PVCs (e.g., NFS/CSI) or remove PVC and **stream directly** to MinIO.

API (FastAPI) – status & presigned URL

```
from fastapi import FastAPI, HTTPException
import redis, boto3, os

r = redis.Redis.from_url(os.environ["REDIS_URL"])
app = FastAPI()

@app.get("/jobs/{job_id}")
def get_job(job_id: str):
    data = r.hgetall(f"job:{job_id}") or {}
    if not data:
        raise HTTPException(404, "unknown job")
    return {k.decode(): v.decode() for k,v in data.items()}

@app.get("/download/{job_id}")
def presign(job_id: str):
    s3 = boto3.client("s3", endpoint_url=os.environ["AWS_S3_ENDPOINT"],
aws_access_key_id=os.environ["AWS_ACCESS_KEY_ID"],
aws_secret_access_key=os.environ["AWS_SECRET_ACCESS_KEY"])
    key = r.hget(f"job:{job_id}", "s3_url")
    if not key:
        raise HTTPException(404, "no artifact")
    bucket = os.environ["AWS_S3_BUCKET"]
    url = s3.generate_presigned_url("get_object", Params={"Bucket":
bucket, "Key": key.decode()}, ExpiresIn=3600)
    return {"url": url}
```

Security

- **mTLS** for gRPC between Celery and Rust workers (SPIRE/certs or cert-manager Issuer).
- **NetworkPolicy** to restrict gRPC to API/Celery only.
- **ServiceAccount** with minimal RBAC; avoid node-wide permissions.
- **Secrets**: use Kubernetes Secrets + optional KMS envelope encryption.
- **Pod Security**: run as non-root, drop `CAP_SYS_ADMIN`, read-only root FS where possible.

```
securityContext:
  runAsNonRoot: true
  runAsUser: 1000
```

```
  fsGroup: 1000
  allowPrivilegeEscalation: false
  readOnlyRootFilesystem: true
```

Observability

- **Metrics** (Prometheus): GPU utilization (DCGM exporter), queue depth, job throughput, success/failure rates, p95 latency, upload time.
- **Logs**: structured JSON logs from API/worker; forward via Fluent Bit or OpenTelemetry Collector.
- **Tracing**: OpenTelemetry SDK in API and Rust worker (OTLP → collector).
- **Alerts**: worker crashloop, PVC > 80% full, queue depth > threshold, high GPU idle.

Example Prometheus annotations are shown on Deployments above.

Development Workflow

1. Stand up local **FastAPI + Celery + Redis**.
2. Integrate **HuggingFace **`` (guard model weights with .gitignore; use HF_TOKEN).
3. Test generation locally with GPU (Docker + `--gpus all`).
4. Build Docker images for API & worker and push to registry.
5. Apply manifests (`kubectl apply -f k8s/`), then `kubectl logs -n video -l app=video-worker`.
6. Submit a job; verify progress and final artifact in MinIO.

Optional Frontend (MVP)

- Minimal SPA or HTML with:
  - Text prompt input
  - Submit button → POST /jobs
  - Status polling → GET /jobs/{id}
  - Download with presigned URL → GET /download/{id}

Hardening Checklist

-

Quick Commands

```
# Port-forward API
kubectl -n video port-forward svc/video-api 8080:80

# Tail worker logs
kubectl -n video logs -f deploy/video-worker -c worker
```

```
# Watch GPU pods
kubectl -n video get pods -l app=video-worker -w
```