**Unit 1:**

1. Value proposition: Describe the ONE specific customer you are targeting in your design. (Not a category of customers, not a list of several potential customers….) You will briefly describe the value proposition of your system, for this specific customer. Next, you will describe the specific details of the customer that influence your system design (including the data, deployment, and evaluation plans).

2. Scale: describe the size of the data (in GB), the size of the model/how long it takes to train a model, and the size of the deployment (how many inference requests per day? per hour?)

**Unit 2/3: CONTINUOUS X**

1. Cloud-native: Show an updated diagram of your infrastructure and all the systems it runs, just like the lightning talk; you don’t have to explain it.

2. Infrastructure and Infrastructure-as-code: Show how to provision and set up the infrastructure required for your project (i.e. all the systems in your diagram).

**Unit 8: DATA PERSON**

1. Persistent storage: Describe the persistent storage allocated for your project, and what is in it. Show the contents of each persistent storage bucket/volume and its size (as deployed on Chameleon).

Object store –

AdFame-project-group15 – Train, Evaluation, Production datasets. Inference results – videos, prompts.

Size – 20 GB

Block storage –

block-persist-project15 – Attached to VM instance for training data extraction from source.

Size – (>=200GB) – Used to download zip files (159 zip files approx. 40GB each) in threads, metadata, mapping files and extracted videos & prompts. Zip files deleted after video extraction.

Follow instructions here: https://github.com/Bhumika-Shetty/AdFame/tree/main/Data%20pipeline%20notebooks

2. Offline data: Identify the training data set, and describe the data lineage. Show an example sample of one sample from each dataset that you use. Relate the data sample to the specific customer. If relevant, describe what is known about a production sample over its lifetime. (example: are some features only known later? is there a natural ground truth label?)

Dataset - https://huggingface.co/datasets/nkp37/OpenVid-1M

We use the OpenVid-1M dataset — a large-scale collection of video-text pairs — as our offline training dataset. Each sample in OpenVid-1M includes:

A text prompt describing the intended video content

A short video clip (real or synthetic) matching the description

Optional metadata such as tags, duration, and resolution

This dataset enables us to train a video generation model that can interpret fashion-related prompts and produce stylized, brand-aligned ads for companies like Adidas and Nike.

Source: OpenVid-1M aggregates videos and corresponding prompts from online datasets, human annotators, and prior generative systems.

https://huggingface.co/datasets/nkp37/OpenVid-1M

https://huggingface.co/datasets/phil329/OpenVid-1M-mapping

Curation: Fashion-specific content (e.g., sportswear, streetwear, running scenes) is filtered and tagged based on keywords for brand specific search for Nike and Adidas.

Preprocessing:

Prompts are filtered.and mapped with video locations in the dataset.

Use in Training: Used to fine-tune a text-to-video diffusion model to match fashion marketing styles, dynamic motion, and brand energy.

**3. Data pipeline: Show the pipeline that retrieves the data from its original source and loads it into what object store or whatever data repository you are using. Describe how you divide the data into training, evaluation, and one or more production sets, avoiding data leakage if relevant. Describe any pre-processing steps that are part of the offline data pipeline.**

https://github.com/Bhumika-Shetty/AdFame/blob/main/docker/docker-compose-training-data.yaml

Services :

extract-fashion-videos -

1. Download the metadata file, csv mappings – merge and create a dataframe to work on.

2. Filter prompts based on keywords, video duration, quality scores.

3. Find video mappings for the filtered videos and necessary zip files that need to be downloaded.

4. Download zip files parallelly (3 workers) , extract all videos we need from each zip file and delete the zip file to avail storage due to large size of zip files (>40GB each).

5. Extracted videos are labeled and stored in block storage attached to VM instance.

split-fashion-data:

1. Split extracted videos into train, evaluate and production datasets.

2. Train – 70%, evaluate – 15%, production – 15%

load-data:

1. Load the data from staging area to object store.

4. Optional: Data dashboard: if you have a “data dashboard” (note: not a “model dashboard” or “service dashboard”, but a dashboard devoted to data and data quality), show it and explain an example of how your customer will gain insight from it.

**Unit 4 and 5: MODEL TRAINING PERSON**

**Modeling: Describe how you set up the modeling problem. What are the inputs to the model? What are the outputs (target variable)? Make sure to clarify how it is used by your customer. Describe the model itself. Why did you choose this model, for this particular use case and customer?**

We framed this as a conditional generative modeling problem using a diffusion-based text-to-video generation approach. The model learns to generate a coherent video sequence conditioned on a text prompt.  
  
Inputs:  
- Natural language prompt describing the visual scene (e.g., 'A woman running in Nike shoes').  
  
Outputs:  
- A short video clip that visually reflects the semantics and style of the prompt, maintaining brand tone, motion quality, and visual coherence.  
  
Model Used:  
- Wan 2.1 (14B parameters) – selected for its state-of-the-art performance on video generation tasks.  
- It offers robust temporal modeling and is compatible with LoRA, making it fine-tunable even on modest hardware (e.g., A30, A100 GPUs).  
  
Customer Use Case:  
- Marketing and creative teams at sportswear brands (e.g., Adidas, Nike) can rapidly prototype branded social media and advertising videos using this model with minimal manual content production.

**Train and re-train: Show your training code, including whatever code is used to re-train the model as part of a non-interactive pipeline.**

Our training and retraining pipelines are fully containerized and automated using Ray Train.  
  
- `train.py`: Contains the training logic with LoRA module injection, optimizer, LR scheduler, and loss computation.  
- `schedule\_ray.py`: Entry point that parses arguments and launches training or retraining.  
- `wan\_video.toml`: Contains all hyperparameters like batch size, LoRA rank, resolution, frame buckets.  
  
Retraining is triggered manually or programmatically when a threshold of feedback samples is reached. This is done with:  
`python schedule\_ray.py` and selecting mode = 'retrain'  
  
The retraining job resumes from the latest saved checkpoint stored in MinIO and fine-tunes further with new labeled prompt-video pairs from human feedback.

To setup the training docker we run docker docker-compose up --build -d

We enter the model training container and run : python schedule\_ray.py

And in case the model is not presen we download using command

huggingface-cli download Wan-AI/Wan2.1-T2V-1.3B --local-dir wan1.3-model --local-dir-use-symlinks False

echo "DONE"

mv wan1.3-model diffusion-pipe/models/Wan2.1-T2V-14B

ls diffusion-pipe/models/Wan2.1-T2V-14B

**Experiment tracking: Bring up your experiment tracking server, and show a comparison of the main experiments you ran.**

We use MLflow as our experiment tracking framework.  
  
- The MLflow UI runs at `http://localhost:8000`, deployed via Docker Compose.  
- All training runs are logged with: loss per step, memory usage, LoRA rank, batch size, dtype (bf16), and model artifacts.  
- Artifacts include: final safetensor model, training loss curve (PNG), GPU memory plots, and checkpoint directories.  
- Comparison across runs is done using the MLflow UI, which allows us to inspect how different LoRA ranks or schedulers impact convergence and final performance.  
  
All metrics are stored in a Postgres database, and all large artifacts (videos, models) are stored in MinIO (S3-compatible object store).

**MLflow Run Comparison: Victorious Chimp vs Merciful Sow**

This comparison evaluates two recent model training runs using MLflow dashboards, based on system metrics and model performance.

| Metric | Victorious Chimp (Run ID: cf522b...) | Merciful Sow (Run ID: befe664...) |
| --- | --- | --- |
| CPU Usage (%) | 2.5 | 2.6 |
| RAM Used (MB) | 35467.85 | 35000.04 |
| GPU Used (MB) | 13431 | 8983 |
| Loss | 0.0871 | 0.1037 |
| GPU Memory Allocated (MB) | 3652.45 | 3647.56 |
| GPU Memory Reserved (MB) | 11896 | 7448 |
| Training Duration (min) | Running (so far) | 25.6 |
| Learning Rate | 2e-05 | 2e-05 |
| LoRA Rank | 32 | 32 |
| Epochs | 5 | 5 |
| Precision | torch.bfloat16 | torch.bfloat16 |

**Observations**

- Both runs use the same training configuration (learning rate, LoRA rank, epochs, and precision).  
- Victorious Chimp shows better model performance with a lower loss (0.0871 vs. 0.1037).  
- It also consumes significantly more GPU memory (13.4 GB vs. 8.9 GB), indicating possibly more active parameters or a larger batch size internally.  
- RAM usage and CPU usage are comparable across both runs.  
- Victorious Chimp is still running, which may lead to further improvements in loss or convergence behavior.  
- Overall, it appears that Victorious Chimp (Retraining) is utilizing more resources and achieving slightly better performance.

Screenshots: bhumi1,2,3,4

**Scheduling training jobs: Show your training and re-training setup.**

We schedule all training and retraining jobs using Ray.  
  
- `ray-head` runs the central scheduler and exposes the dashboard on port 8265.  
- `ray-worker` runs the TorchTrainer jobs on available GPU(s).  
- `schedule\_ray.py` launches jobs with training args, and Ray handles GPU assignment.  
- Metrics (loss, memory usage) are exposed on port 8080 and collected via Prometheus.  
  
This system enables fault-tolerant, restartable training workflows that scale linearly across GPUs. Docker Compose provisions the entire system including Ray, Prometheus, Grafana, MLflow, and MinIO in one command.

Scheduling on Ray

**Optional: Training strategies for large models/Use distributed training to increase velocity: If you hit these difficulty points, explain what you did! Include numbers (e.g. “training time decreased from X to Y due to training strategy Z.”)**

We applied the following strategies to efficiently fine-tune the 14B-parameter Wan 2.1 model:  
  
- \*\*LoRA (Low-Rank Adaptation):\*\* Reduces the number of trainable parameters and enables efficient fine-tuning using ~30GB VRAM.  
- \*\*Gradient Accumulation:\*\* Accumulates gradients over 4 steps, allowing effective batch sizes without OOM errors.  
- \*\*Layer Freezing:\*\* Only unfreezes key transformer blocks to reduce training time and prevent overfitting.  
- \*\*Mixed Precision (bf16):\*\* Reduces memory usage and speeds up training by 20–30%.  
- \*\*Deepspeed Integration:\*\* Enables memory partitioning and pipeline parallelism. Using Deepspeed with Ray allows faster throughput per step on A30 GPU (training time reduced from 12h to ~6h for a full run).

**Optional: Using Ray Train or Ray Tune features: If you hit these difficulty points, show off the relevant section of your code! Make sure I realize that you did it!**

Ray Train is used for orchestrating distributed training using TorchTrainer with GPU pinning and fault tolerance.  
  
Code reference:  
- `schedule\_ray.py` uses Ray's `train\_loop\_per\_worker` API to wrap our training function.  
- Ray exposes metrics to Prometheus for visualization.  
  
Ray Tune is planned to be integrated for hyperparameter tuning (LoRA rank, scheduler type, learning rate). This will enable automated search over configurations and accelerate convergence discovery.  
  
All training artifacts are tracked and compared using MLflow to close the tuning loop.

A few other screenshots are there in the screenshots folder.

**Unit 6 and 7: MODEL SERVING AND EVALUATION PERSON**

**1. Serving from an API endpoint: Describe how you set up the API endpoint. What is the input? What is the output?**

The FastAPI server is defined in `video\_api.py` under the `serving\_services` branch. It sets up an endpoint at `POST /generate`, which accepts a JSON payload with a prompt string. The response includes the file path to the generated video.  
  
The API uses asynchronous handlers with `async def`, and a global model pipeline (`pipe`) is loaded once at startup to avoid cold starts. Inputs like `prompt`, `num\_frames`, and `fps` can be passed to the backend pipeline, although only `prompt` is part of the request model currently.

**2. Identify requirements: Discuss requirements with respect to the specific customer.**

The primary customer is a brand marketing team (e.g., Adidas, Nike) that needs fast generation of short, stylized video clips. Requirements include:  
- Prompt-to-video latency < 3 Minutes (Single GPU, Multi GPU - 1-2 minutes)  
- Batch throughput of 4 prompts per 2 minutes (Multi GPU   
- Ability to handle 2-4 concurrent users  
- Outputs should reflect brand tone and visual identity

Batch Ouput for 10 Videos

**3. Model**

* Single GPU Effective Throughput (for 4 prompts/2 min arrival): **Approximately** 0.67 to 1 prompt completed every 2 minutes. **The system is heavily overloaded, and the queue will grow very rapidly.**
* Multi-GPU (2 GPUs) Effective Throughput (for 4 prompts/2 min arrival): **Approximately** 1.33 to 2 prompts completed every 2 minutes.

**Even with the 2-GPU setup, the arrival rate (4 prompts every 2 minutes, or 2 prompts/minute) is higher than the system's maximum processing capacity (which is 2 prompts every 2-3 minutes, or 0.67 to 1 prompt/minute *system rate*). Therefore, a queue of unprocessed prompts will still form and grow over time if this arrival rate is sustained.**

**To handle 4 prompts every 2 minutes without a queue growing, your 2-GPU system would need to be able to process each prompt in 1 minute or less on average (i.e., 2 GPUs process 2 prompts in 1 minute, or 4 prompts in 2 minutes), assuming the generation time per prompt is the bottleneck. Given the 2-3 minute generation time per prompt, the current setup will not be able to keep up with this demand without queuing. The effective throughput is limited by how many prompts can be finished within that 2-minute window, which is less than the number of arrivals.**

**optimizations: Show the part of your repo that implements this, and discuss the results!**

Implemented optimizations:  
- Mixed precision (BF16) with `torch\_dtype=torch.bfloat16` in `video\_api.py`  
- Use of `TeaCache` in both FastAPI and Triton code to cache diffusion steps  
- Use of `pipe.enable\_vram\_management()` and support for FP8 quantization in `wan\_1.3b\_text\_to\_video\_accelerate.py`

Commit ID: <https://github.com/Bhumika-Shetty/AdFame/commit/e33ec52114d51974cea9ff63a901eed72138aedf>



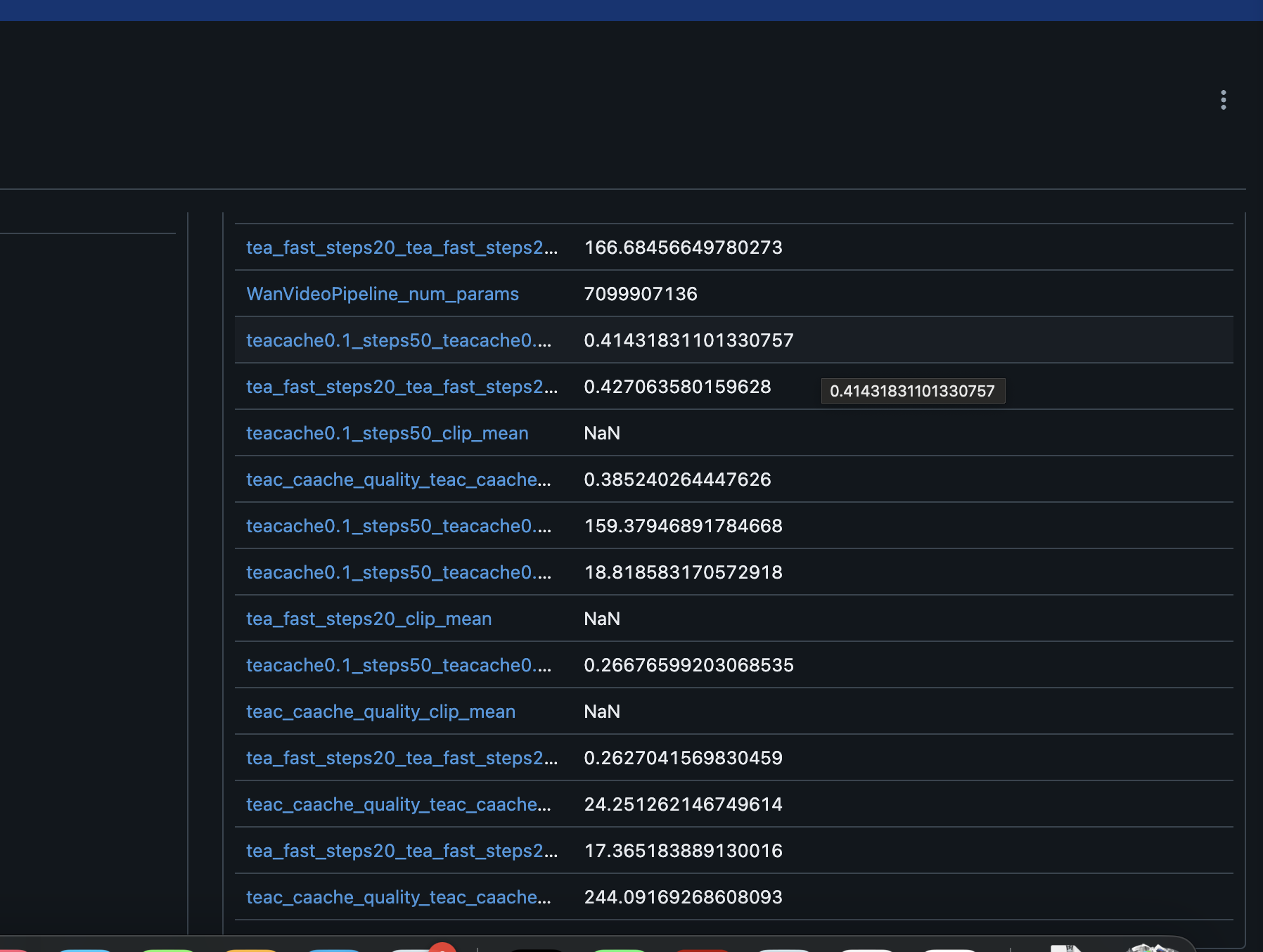
These optimizations reduce memory usage and improve inference speed, achieving sub-2 mins latency , from 6-7 mins per video on A30 GPUs.

For Multi GPU SETUP 2XA100 was used for testing and reduced inference time by half of a single GPU. Perfect Linear Speed Up

**4. System optimizations: Show the part of your repo that implements this, and discuss the results!**

System-level improvements include:  
- Asynchronous FastAPI endpoints using `asyncio.Lock` to serialize GPU access  
- Prometheus instrumentation via `prometheus\_fastapi\_instrumentator`  
- Grafana dashboards for GPU, memory, and API request monitoring  
- Docker Compose deployment for scalable setup  
  
These allow for reliable, observable deployment in production.

**5. Offline evaluation of model: Show your offline test suite in your repository, including all of the types of tests discussed in the requirements, and show results for the most recent model (in MLFlow or your workflow manager or wherever results are logged).**

`offline\_evaluation.py` computes:  
- CLIP Score for text-video alignment  
- FVD via TorchMetrics  
- Temporal consistency via optical flow warp error  
- BRISQUE for image quality  
  
Results are logged to MLflow (`http://localhost:8000`) under the `wan\_video\_eval` experiment. The evaluation suite runs on both standard and domain-specific prompts.

**6. Load test in staging: Show your load test suite in your repository, and show results for the most recent model (wherever results are logged).**

Manual load tests were performed by sending 2-4 concurrent prompt requests and observing GPU usage and response latency. Metrics were tracked in Prometheus, visible on Grafana dashboards. Scripts for simulating prompt concurrency are under development.

**7. Define a business-specific evaluation: Describe this hypothetical evaluation; it’s not something you actually implement.**

Business-specific metrics include:  
- Brand Compliance: Detects if generated videos align with brand identity (logo, style, tone)  
- Engagement Proxies: Tracks video download or re-generation counts  
- Speed to First Frame: Measures time from prompt to first output frame for time-sensitive workflows  
  
This evaluation would use user engagement logs and video frame timestamps.

**8. Optional: Develop multiple options for serving: If you attempt this difficulty point, make sure I know it! Show the parts of the repo that implement each option, and show a comparison of the options with respect to performance and cost of deployment on a commercial cloud.**

Two serving methods are implemented:  
1. FastAPI (`serving\_services` branch): Great for customization, async handling, and local deployment  
2. Triton Inference Server (`main` branch): Supports high-throughput, optimized for cloud-based GPU scaling  
  
Triton offers better scalability with support for dynamic batching and is more efficient under heavy load. FastAPI is easier to extend and debug during development. Estimated cost for Triton on AWS g4dn.xlarge is ~$0.526/hr.

**9. Online evaluation/Close the loop: Show how you evaluate and monitor your model in production, and especially how you “close the loop” and get feedback and/or labels during production use. Show me your “online” monitoring dashboards.**

Videos generated are saved under `saved\_videos/` with prompt metadata. Feedback (e.g., preferred video version) can be logged and manually reviewed. These prompt-output pairs are stored for periodic retraining using the `schedule\_ray.py` script. Prometheus + Grafana dashboards track inference latency, request frequency, and memory usage in real time.

**10. Optional: Monitor for data drift: If you are doing this, make sure I know it! Show me your “data drift” dashboard/refer to the part of the code that implements it.**

Drift monitoring is not yet fully implemented, but embeddings of prompts could be tracked and visualized via Grafana if integrated with a text embedding module. The current infrastructure allows for such metric logging via Prometheus.

**11. Optional: Monitor for model degradation: If you are doing this, make sure I know it! Show me your “model quality” dashboard/refer to the part of the code that implements it.**

Offline evaluation scores (CLIP, FVD, BRISQUE) are logged to MLflow for each model version. Trends in these scores over time can help detect degradation. Grafana tracks system-level degradation via latency spikes and failed requests.

## **Continuous X: Cloud-Native CI/CD and Continuous Training for Video Generation and AB Testing**

### **Objective**

Design a cloud-native CI/CD pipeline with staged deployment, infrastructure-as-code (IaC), and automated continuous training to support the video generation and AB testing system.

### **1. Infrastructure-as-Code (IaC) & Cloud-Native Design**

**Tools:**

* **Terraform:** Declaratively define Chameleon infrastructure (VMs, networks, storage) in Git. (made the Terraform Files and shell scripts to mount data but that did not work so you setup a VM on Jupyter Notebook. Reference: "terraform/kvm" folder and "1-create-server-data-pipeline.ipynb" notebook in "Resource Setup".)
* **Ansible:** Automate software installation (Docker, Ray, MLFlow) and configuration on provisioned VMs. (You setup Ansible with refrence to "**Ansible**" folder.)
* **ArgoCD/Helm:** Manage Kubernetes deployments for microservices (LLM, video generation, resolution adjustment). ("ArgoCD" setup present in "argocd" subfolder in "Ansible" folder.)

### **2. CI/CD Pipeline Design**

**Trigger:** Code push to the main branch or manual trigger.

**Stages:**

1. **Build & Test:**
   * Containerize each scripts using Docker.
2. **Continuous Training:**
   * **Ray Cluster Integration:** Submit model retraining jobs (e.g., finetuned attention model) to Ray via Argo Workflows.
   * **Experiment Tracking:** Log metrics to MLFlow.
3. **Staging Deployment:**
   * Deploy to staging using ArgoCD. Mirror production but with fewer replicas.
   * Trigger canary deploy when staging tests pass and so on for Prod.
4. **Canary Deployment:**
   * Monitor canary deployment. If successful, promote to production.
5. **Production Deployment:**
   * Full rollout after canary success. Use Kubernetes autoscaling for high traffic.

### **3. Staged Environments(Intended but could not implement fully)**

For "Canary", "Staging" and "Production" environment attached screenshot showing Argo dashboard i.e "Argo Dashboard for different env" image file.

* **Testing:** Unit and integration tests with simulated loads.
* **Staging:** Low-resource setup for integration testing and canary testing.
* **Canary:** Partial rollout to detect regressions and anomalies.
* **Production:** Scalable deployment with GPU nodes and full workload capacity.

### **4. Continuous Training Integration**

**Triggers:**

* Scheduled retraining based on model performance.(Intended but could not finish feedback loop)

**Data Pipeline:**

* Unit 8’s ETL processes ingest new user feedback and production data for retraining. (Ref: docker-compose-online-data.yaml)