

Distributed htrflow

Pipelines plus campaigns

Start from how htrflow works

pipeline.yaml – an htrflow pipeline, unchanged

```
steps:
- step: Segmentation
  settings:
    model: yolo
    model_settings:
      model: Riksarkivet/yolov9-regions-1
- step: Segmentation
  settings:
    model: yolo
    model_settings:
      model: Riksarkivet/yolov9-lines-within-regions-1
- step: TextRecognition
  settings:
    model: TrOCR
    model_settings:
      model: Riksarkivet/trocr-base-handwritten-hist-swe-2
```

```
htrflow pipeline pipeline.yaml images/
```

One folder of page images in, one folder of ALTO and PAGE out, on one GPU.



From one machine to many

	 one machine	 many nodes
 where it runs	your machine, its GPU	whichever node has a GPU free — chosen for you
 the pages	a folder on its disk	fetches over the web — from a IIIF manifest or plain image URLs — by whichever node runs the volume
 the models	downloaded to that disk	a shared cache every node mounts
 the results	a folder next to the pages	a bucket every node writes to and every browser reads from
 when a machine fails	you start again	the volume restarts on another node and resumes from the bucket
 how you start it	a command on that machine	a file in git — you never name a machine

"Volume" here is an **archival volume** — a bound unit of pages with a reference code such as R0001203, the batch one run works through — never a Kubernetes volume, which is a disk.



The campaign file

by reference code

```
pipeline: demo-v1
volumes:
  - R0001203
  - R0001204
```

by IIIF manifest

```
pipeline: demo-v1
volumes:
  - id: loc-mal2459400
    manifest: https://.../manifest.json
```

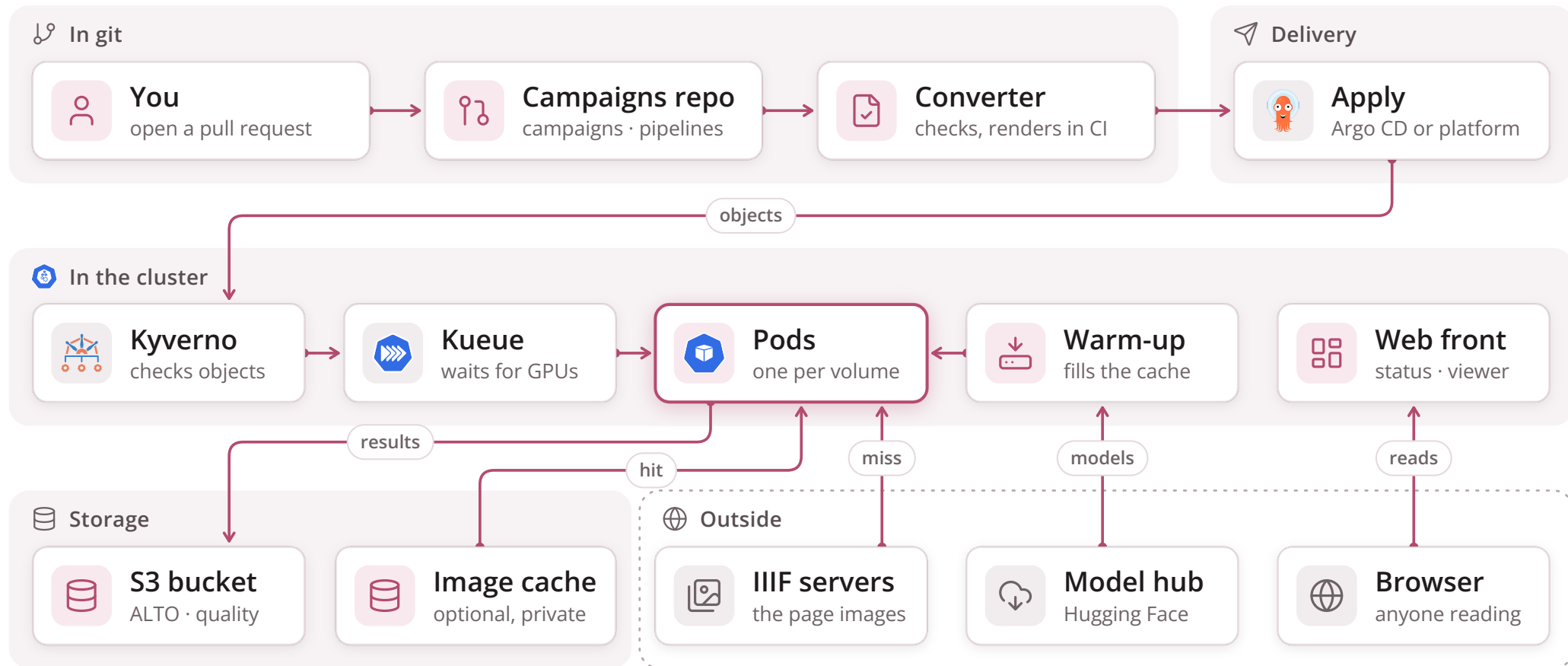
by image URLs

```
pipeline: demo-v1
volumes:
  - id: loose-scans
    images:
      - https://.../scan-0001.jpg
      - https://.../scan-0002.jpg
```

This is how you ask for a run: one file per campaign, in git — a **pipeline** to run, and the archival **volumes** to run it on. A reference code alone is enough: the platform turns it into the archive's IIIF manifest.



Rough architecture



You only open a pull request. The converter checks and renders it in CI; apply — run by Argo CD or the platform team, never by you — sends it to the cluster.



What the platform is made of

BUILT IN HTRFLOW-BATCH



wrapper
runs htrflow in each pod



converter
validate · render · apply



web front
status page · run log



Helm charts
install the platform

PROJECTS WE BUILD ON



htrflow



Kubernetes



Kueue



Kyverno



Universal
Viewer



NVIDIA GPU
stack



S3 store

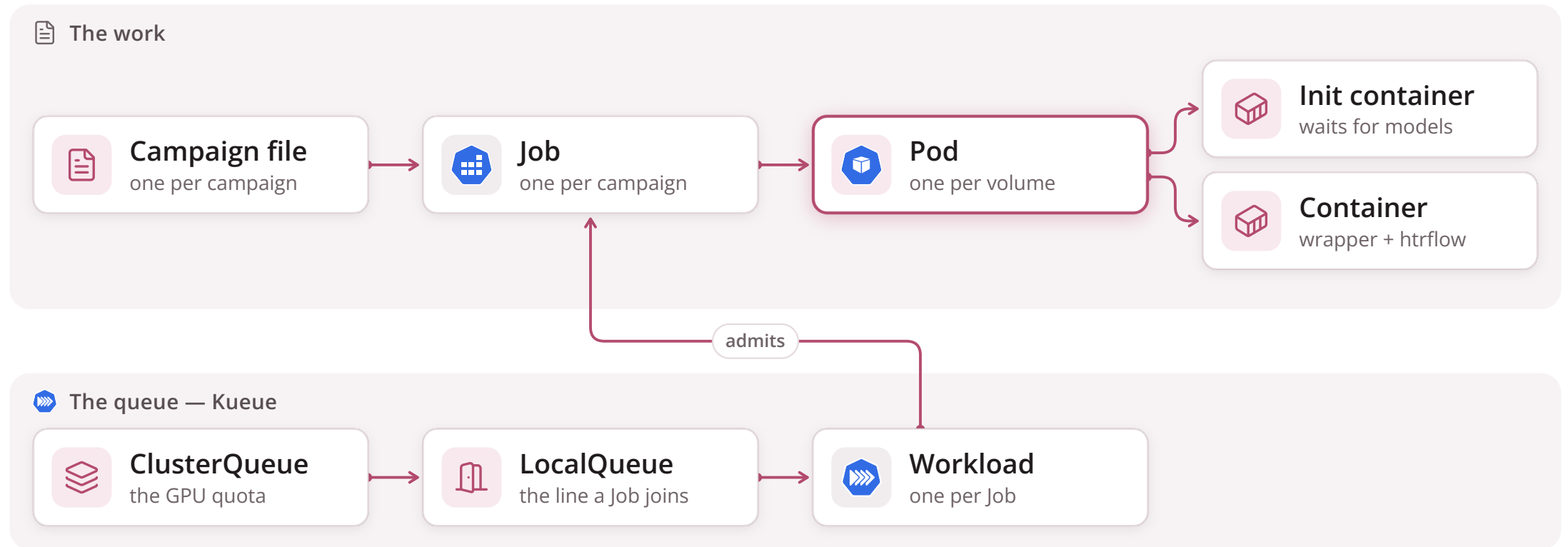


Argo CD
optional

We built the four pieces on top. Everything below is an existing project we use as it is — htrflow included, driven as a library.



From campaign file to container



The work and the queue meet at the **Workload**: Kueue admits a Job's Workload, and only then does the Job create its pods.



Kueue

A job queue for Kubernetes. It decides **when** a Job may start, from a counted budget of GPUs.

Here it:

- holds a campaign until its GPUs are free
- lets higher priority go first
- pauses and resumes campaigns

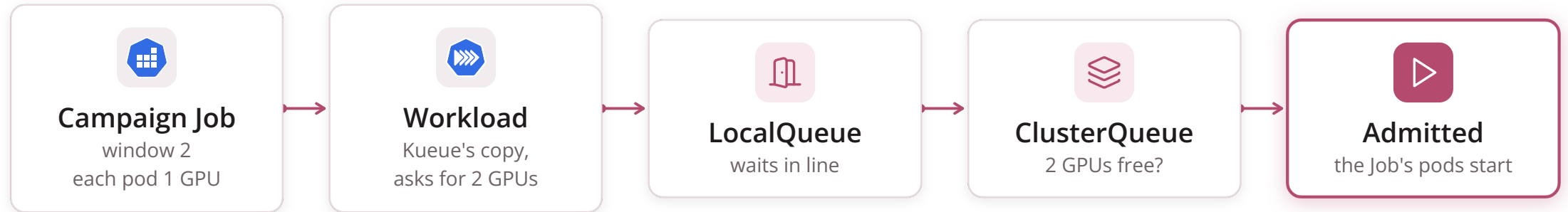
It can also:

- **preemption** — stop lower-priority work to make room
- **fair sharing** — divide idle GPUs by weight between teams

Why: without it, Kubernetes starts every pod it can, whoever asks first takes every GPU, and the rest pile up half-started.



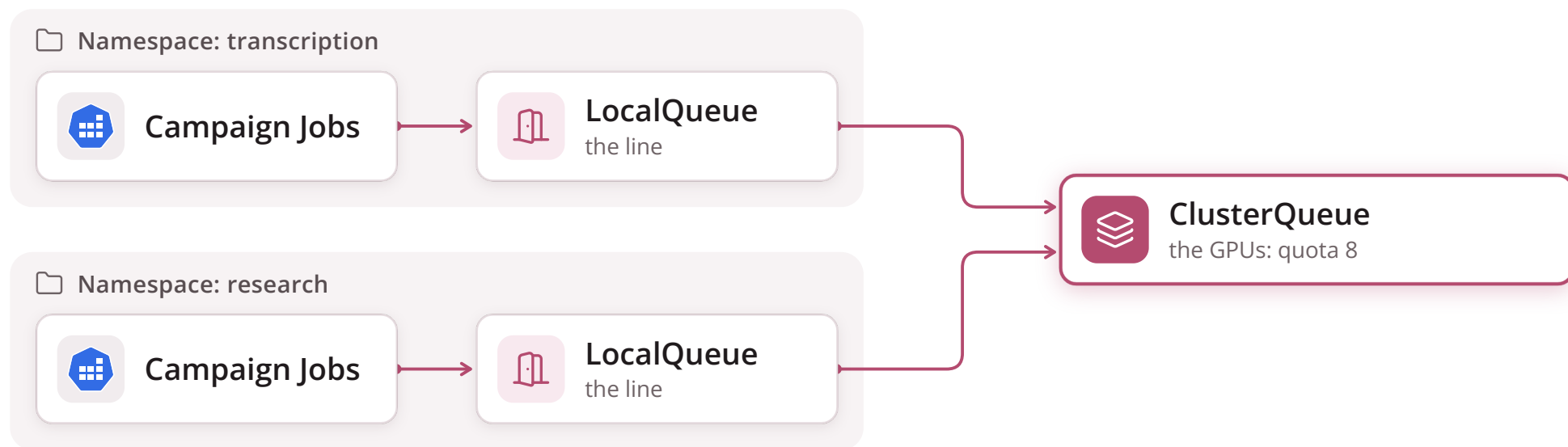
The Workload



A Workload is what Kueue puts in the queue. It says what the campaign needs: two pods, one GPU each, so two GPUs. Kueue writes it when the Job is created. The pods start when both GPUs are free.



LocalQueue and ClusterQueue

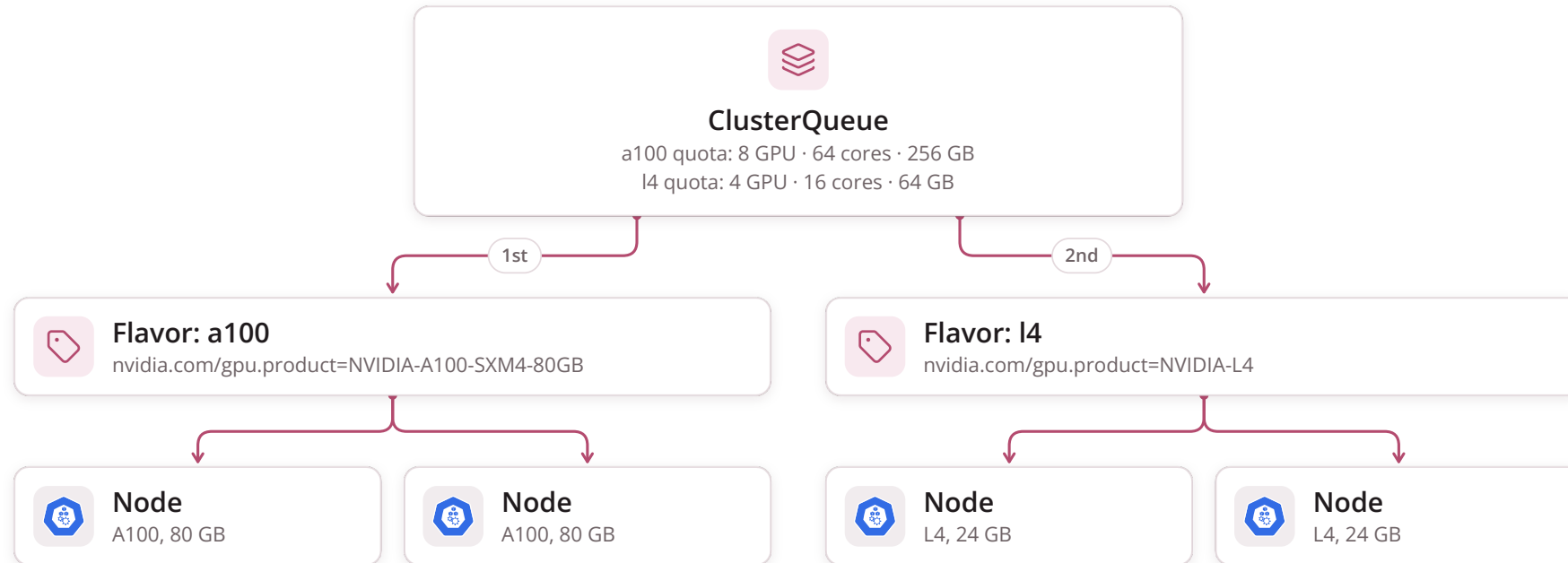


LocalQueue — the door. It lives in a team's namespace, and a Job names it to get in line. It holds no GPUs of its own.

ClusterQueue — the pool. It holds the GPU quota, for the whole cluster. Many LocalQueues can point to one ClusterQueue, and share its GPUs.



ResourceFlavor

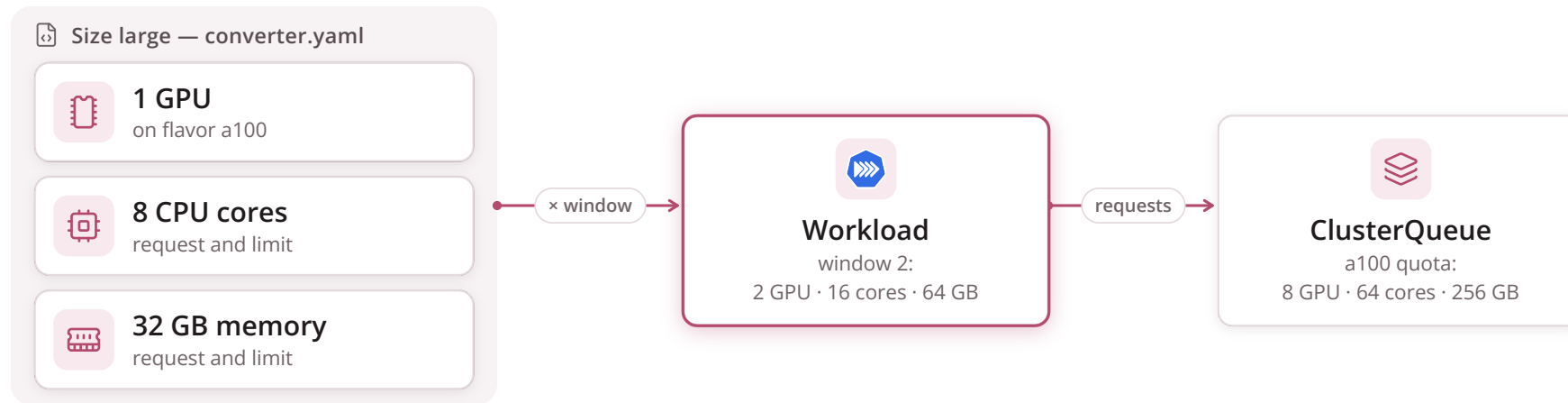


A kind of GPU. A flavor names nodes by their label — the GPU model NVIDIA's feature discovery writes on each node, A100 or L4 — and Kueue adds that label to the pods it admits, so they land on the right machines.

A quota per flavor. A ClusterQueue promises GPUs, cores and memory on each flavor, and tries them in order: a pod that may use either takes an A100 if one is free, an L4 if not.



GPU, cores and memory



```
# converter.yaml — the operator names sizes
sizes:
  small: { flavor: l4,   gpu: 1, cpu: 4, memory: 16Gi }
  large: { flavor: a100, gpu: 1, cpu: 8, memory: 32Gi }

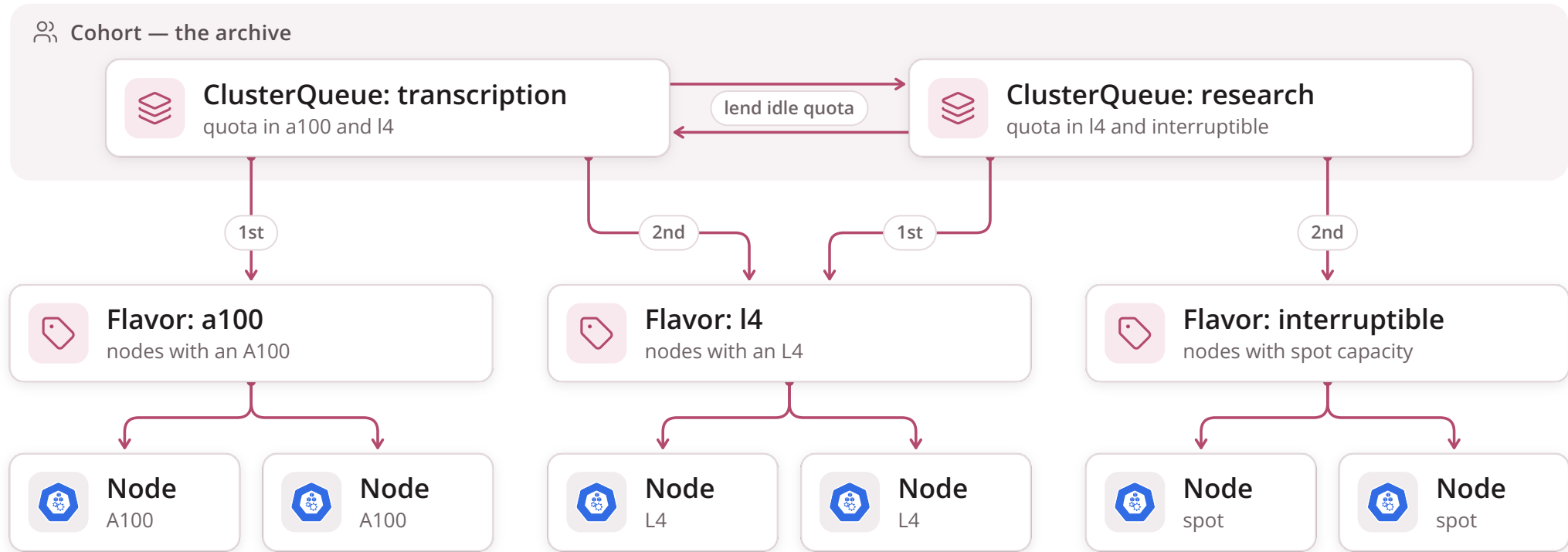
# pipelines/demo-v1.yaml — the recipe picks one
size: large
```

You pick a size by name. The platform team decides what **large** means: how many GPUs, how many cores, how much memory, and which kind of GPU. **validate** refuses any other name.

Two pods of **large** need twice that. The campaign asks for 2 GPUs, 16 cores and 64 GB. It waits until the quota has room.



Cohort

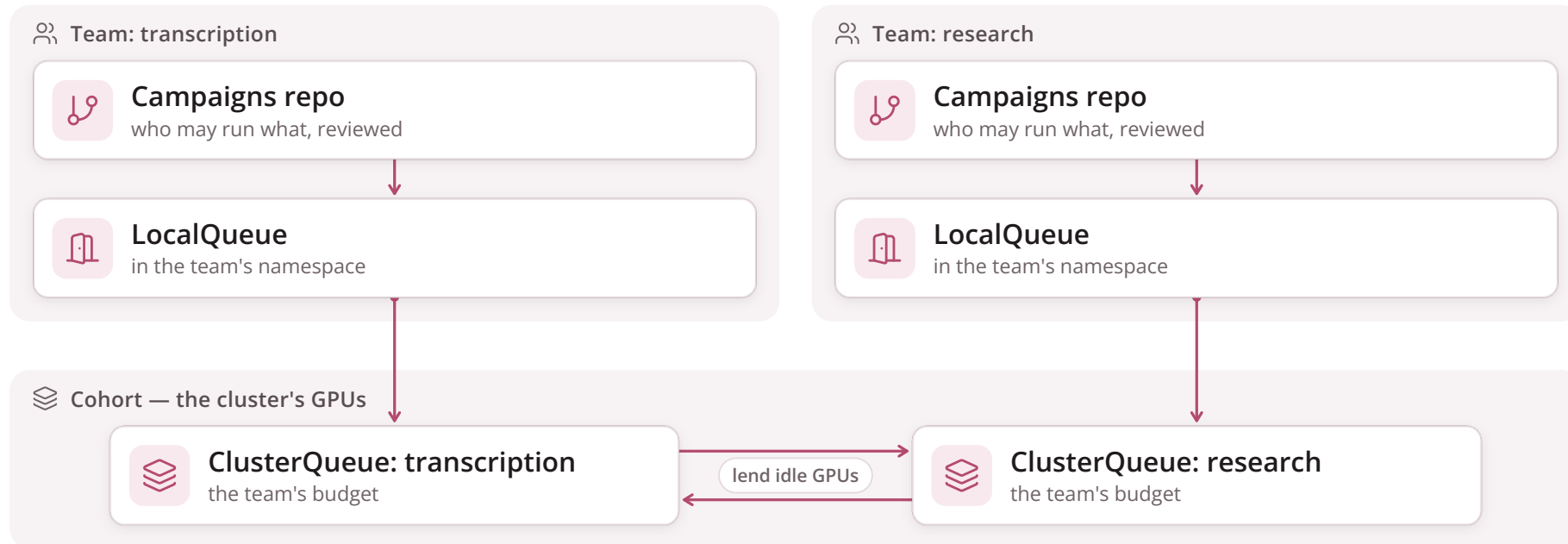


Pools that lend. ClusterQueues in one cohort borrow each other's idle quota, within limits each pool sets — a busy team runs on a quiet team's GPUs, and the owner can take them back.

Here: planned. Today there is one pool.



One team, one repo, one queue



```
# converter.yaml – in the team's repo
namespace: transcription  # where its Jobs go
queue: transcription      # the LocalQueue they name
```

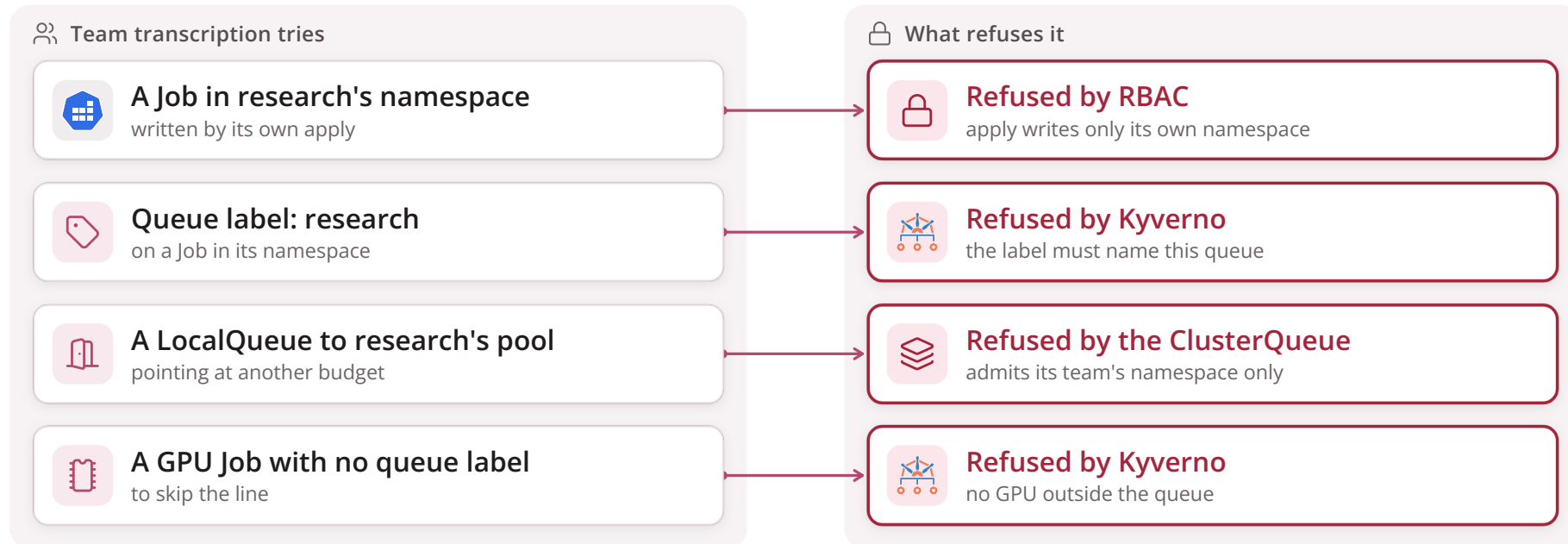
The converter writes both onto every Job; that LocalQueue points at the team's ClusterQueue.

A team is a repository: whoever can merge chooses what runs with the team's credentials.

A budget per team, lent out when idle. Urgency inside a team is **priority**, not another queue.



A team stays in its own queue



RBAC is the wall: each team's apply may write only its own namespace. Kyverno says what went wrong, in one sentence.



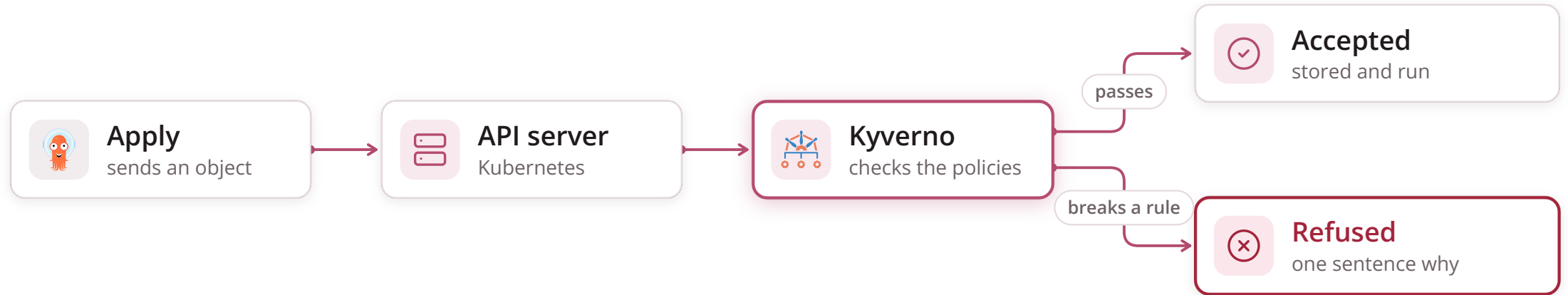
Why Kyverno

	 without a rule on the cluster	 with Kyverno
 an image by tag	someone pushes a new build under the same tag, and every later run silently uses other code	refused unless pinned by digest — the image an ALTO names is the image that ran
 a model without a revision	the author re-uploads it, and the same pipeline gives different text	refused unless pinned to a commit — a pipeline id keeps meaning one set of weights
 an image from anywhere	a merged file can run any code on our GPUs, with the bucket's credentials	refused unless it comes from a registry we allow — optionally, signed by our own build
 a Job sent by hand	anything that skips the converter skips its checks	checked anyway — every object sent to the cluster passes through

It enforces good provenance. When every image is pinned and every model has a revision, what each ALTO says produced it is true — and the cluster enforces that, not every campaigns repository on its own.



Kyverno — how admission works

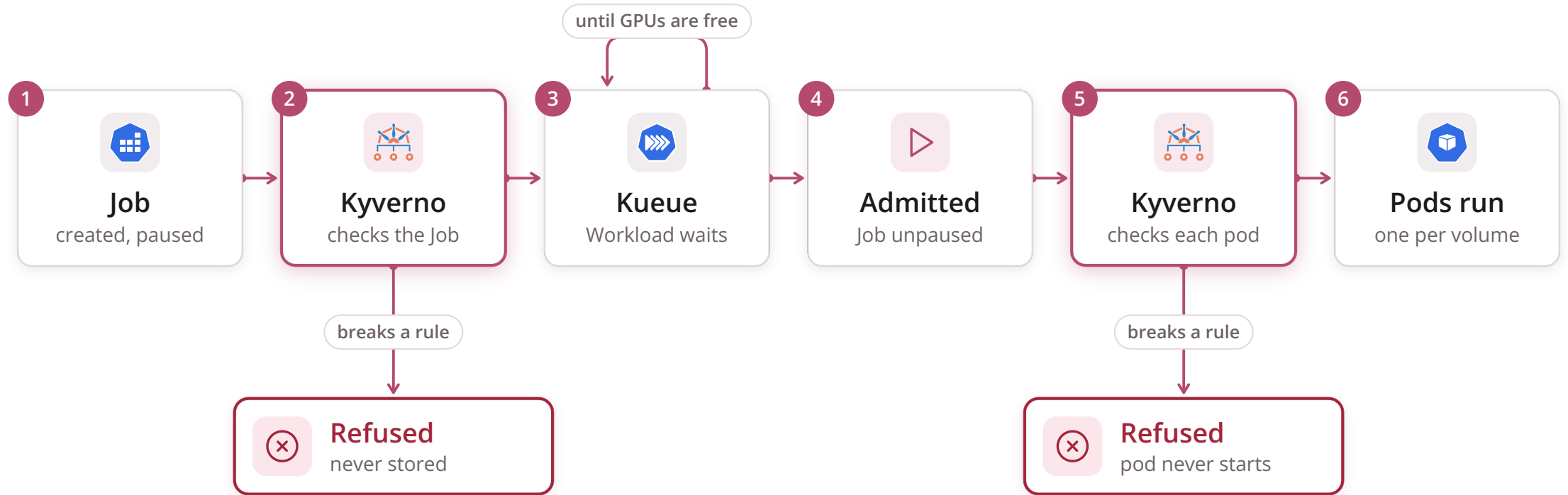


Every object is checked before it exists — once the platform turns the rules on, since they ship switched off — and the same rules run in the pull request, so a bad pipeline usually fails there first:

```
models not pinned to a revision: Riksarkivet/yolov9-regions-1
– add revision: <40-character commit hash> under model_settings
(YOLO) or model_settings.model_kwargs (TrOCR and other Hugging Face models)
```



Kueue — how a campaign gets its GPUs



Kyverno checks twice: the Job before it is stored, so a bad one never reaches the queue — and each pod after admission, where image signatures can be checked.



Stop, remove, restart — all in git

stop

```
pipeline: demo-v1
suspend: true
volumes:
  - ...
```

Running volumes stop, finished ones are kept, the GPUs go back. Delete the line to go on from where it stopped.

Every one is a pull request — reviewed and merged like any other change.

remove

```
git rm campaigns/demo.yaml
```

The Job is removed from the cluster. The results in the bucket stay — nothing cleans up S3 for now, so removing results is a manual step.

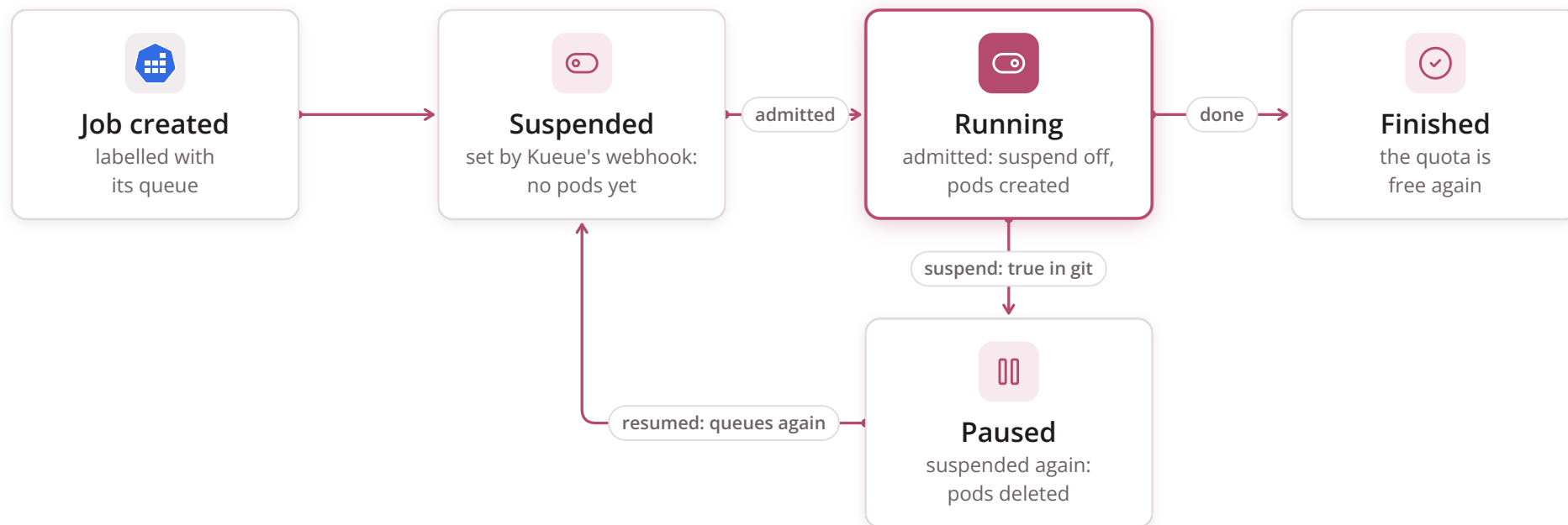
restart

```
git mv campaigns/demo.yaml \
  campaigns/demo-2.yaml
```

A new name runs the campaign again, and skips every page already in the bucket.



Suspend — Kueue's switch on the Job

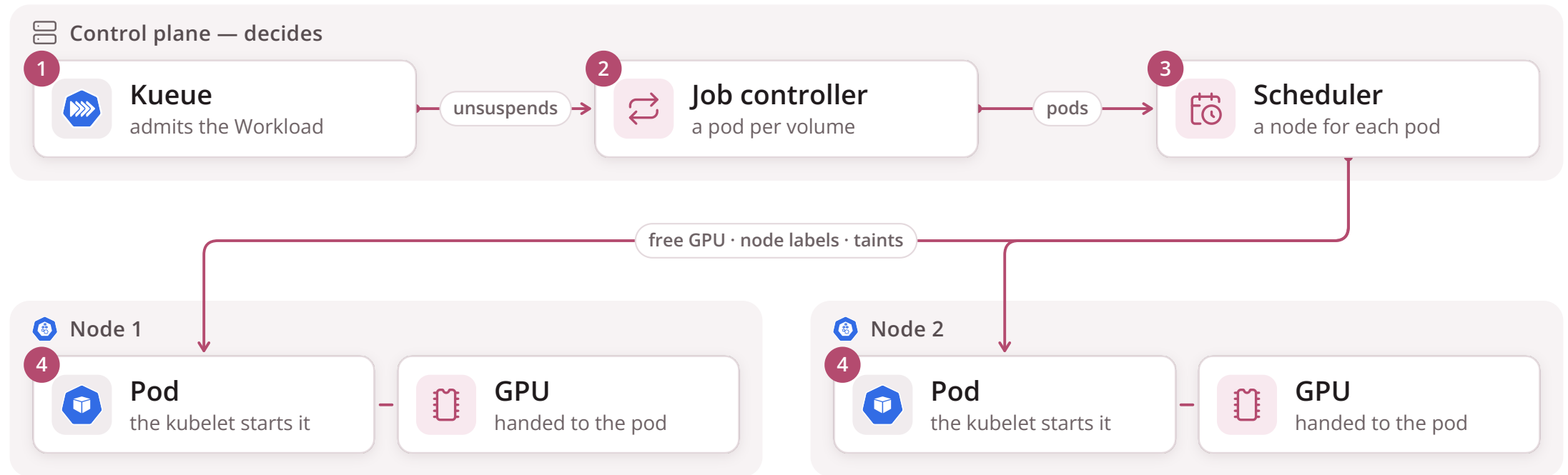


Why a switch. While a Job is suspended its controller makes no pods: nothing holds a node, and a campaign starts whole or not at all. Kueue only decides *when*; the ordinary Job controller still runs the pods.

Pause is the same switch. `suspend: true` in the campaign file makes the apply set the Workload inactive, and Kueue suspends the Job: its pods go, the quota returns. Remove it and the campaign queues again; each volume resumes from the bucket.



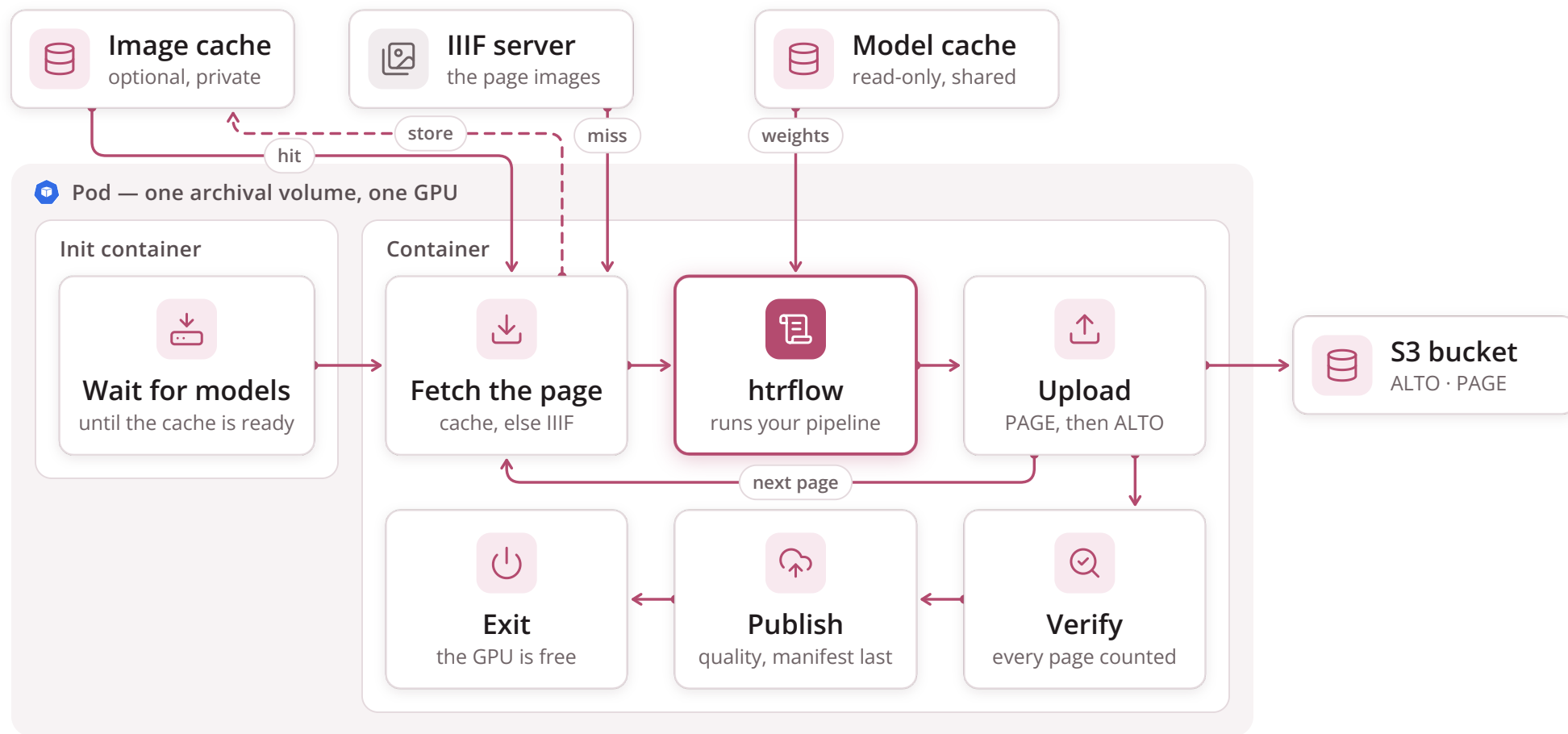
Where a pod runs



The control plane decides, the nodes run. Kueue counts quota, not free GPUs: the scheduler places each pod on a node with a GPU free and the right labels, and a pod with nowhere to go waits *Pending*. You never name a machine.



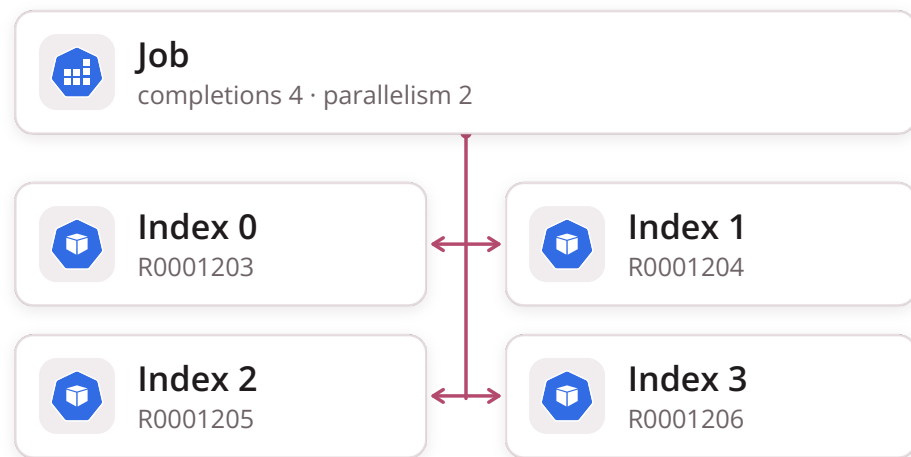
htrflow in a pod



Every pod runs htrflow — your pipeline, unchanged — on one archival volume, page by page.



A campaign is a list of volumes, and one Job



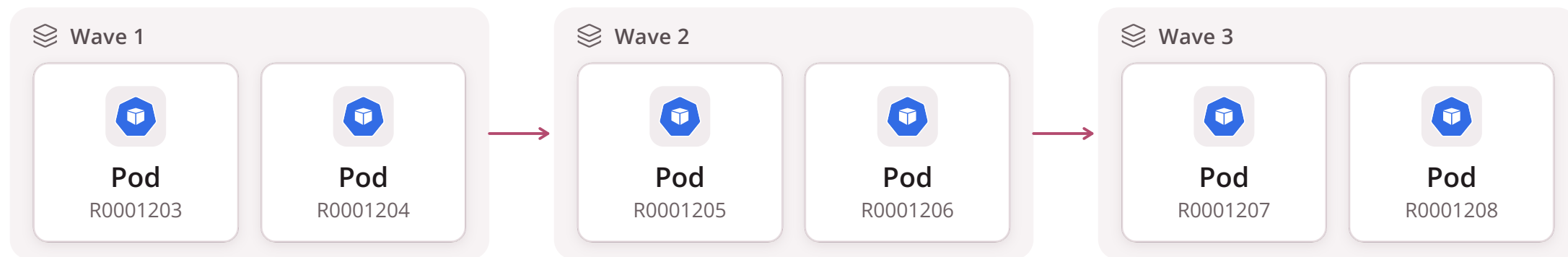
campaigns/demo.yaml

```
pipeline: demo-v1
window: 2          # parallelism
volumes:           # completions = 4
- R0001203
- R0001204
- R0001205
- R0001206
```

- A **Job** is Kubernetes' word for "run this to completion". An *Indexed* Job runs it N times, and hands each pod its number.
- **Pod number 2 reads line 2 of the volume list.** That is the whole trick. No database, no controller of ours, nothing to keep in sync.
- **parallelism** is how many run at once. **completions** is how many there are — and it is fixed the moment the Job is created.



window: how many volumes at once



campaigns/demo.yaml — you set it here, per campaign

```
pipeline: demo-v1
window: 2           # optional; the cluster caps it
volumes:
  - R0001203
  - R0001204
  # ... six in all
```

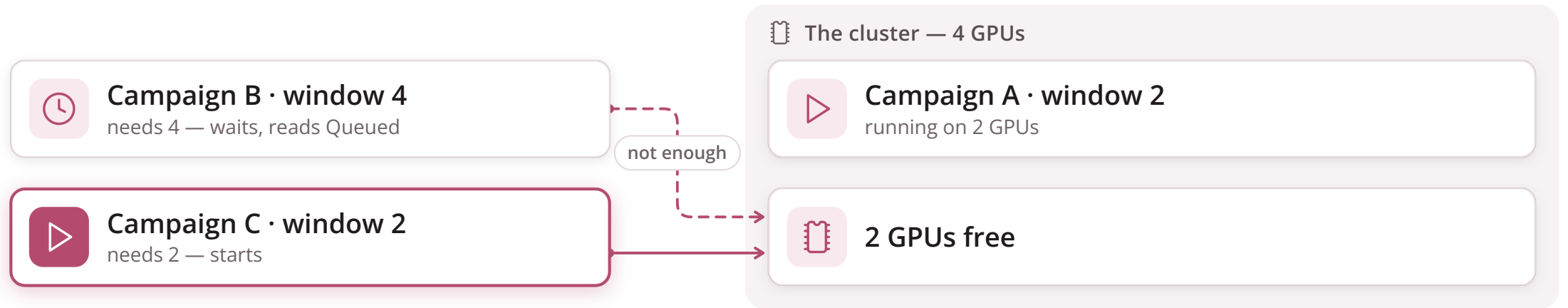
Six volumes, **window: 2**. Two pods at once, each on its own GPU; when one finishes, the next volume takes its place. Nobody plans the waves — the job keeps two indexes busy.

So **window** is the campaign's GPU count — not the number of volumes, not a speed setting. **window: 1** is fine, and takes six times as long.

Asked for as a whole, capped by the cluster. Two GPUs free means it starts; one free means it waits. Above the cap it is clamped; left out, it gets the cap.



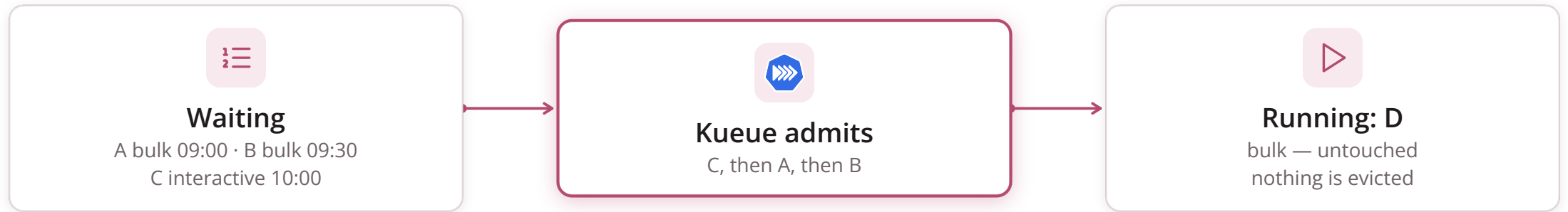
When there are not enough GPUs



A campaign starts only when **all** the GPUs its window asks for are free. Until then its card reads *Queued* — and a smaller campaign that fits may start before it.



priority: who goes first in the line



campaigns/demo.yaml

```
pipeline: demo-v1
priority: htr-interactive # optional; default is htr-bulk
volumes:
  - R0001203
```

Three classes ship with the cluster. **htr-interactive** for a handful of volumes someone is waiting for, **htr-bulk** for the normal campaign, **htr-idle** for work that may wait for the gaps.

Priority orders the queue. Among the campaigns waiting, the higher class goes first; within a class, the older one. That is all it does.

It never evicts. A running campaign keeps its GPUs until its last volume is done, whatever arrives behind it. Preemption is deliberately off.



Results stream into a bucket, and the bucket is the truth

```
htr-batch/demo-v1/R0001203/
  page/0001.xml      PAGE XML, written first
  alto/0001.xml      ALTO – "this page is done"
  page/0002.xml
  alto/0002.xml
  ...
  iiif.json          open it in the viewer, republished every ten pages
  progress.json      pages done and failed, live
  pipeline.yaml      the steps this run used
  manifest.json      written LAST – the only thing that means "done"
```

- **Resume is a list operation.** A pod starts by listing its folder. Every page already done is skipped; the rest are fetched.
- **So a crash costs one page.** Kubernetes restarts the pod, it lists, it carries on.
- **Provenance is in the files.** Every ALTO names the image digest and the model revisions; `manifest.json` names every source URL.
- **The pipeline id is in the path.** A better recipe writes beside the old results, never over them.



Where the state lives

	holds	if it is lost
 git	what <i>should</i> run: campaign files and pipelines — and an audit trail of who changed what, who approved it, and when	nothing running stops, nothing new can be asked for — and every clone is a full copy
 etcd the cluster's database	what <i>is</i> running: Jobs, Workloads, the campaign records the status page reads	rebuilt from git: apply again — campaigns run again, but no page already in the bucket is transcribed again
 S3 bucket	the results: ALTO, PAGE, manifest.json, the run logs	the transcriptions are gone — only running every campaign again brings them back

Only the bucket cannot be rebuilt from the others. That is where replication and backups matter most.

Jobs do not stay in etcd. A finished Job is deleted a week after it ends — its TTL. The campaign's record stays, so the status page still shows it and its results.



Your interface is git



pipelines/demo-v1.yaml – the htrflow pipeline, plus the image that runs it

```
image: docker.io/riksarkivet/htrflow-batch@sha256:637fbe...
steps:
  - step: Segmentation
    settings:
      model: yolo
      model_settings:
        model: Riksarkivet/yolov9-regions-1
# ... the rest of the htrflow steps, unchanged
```

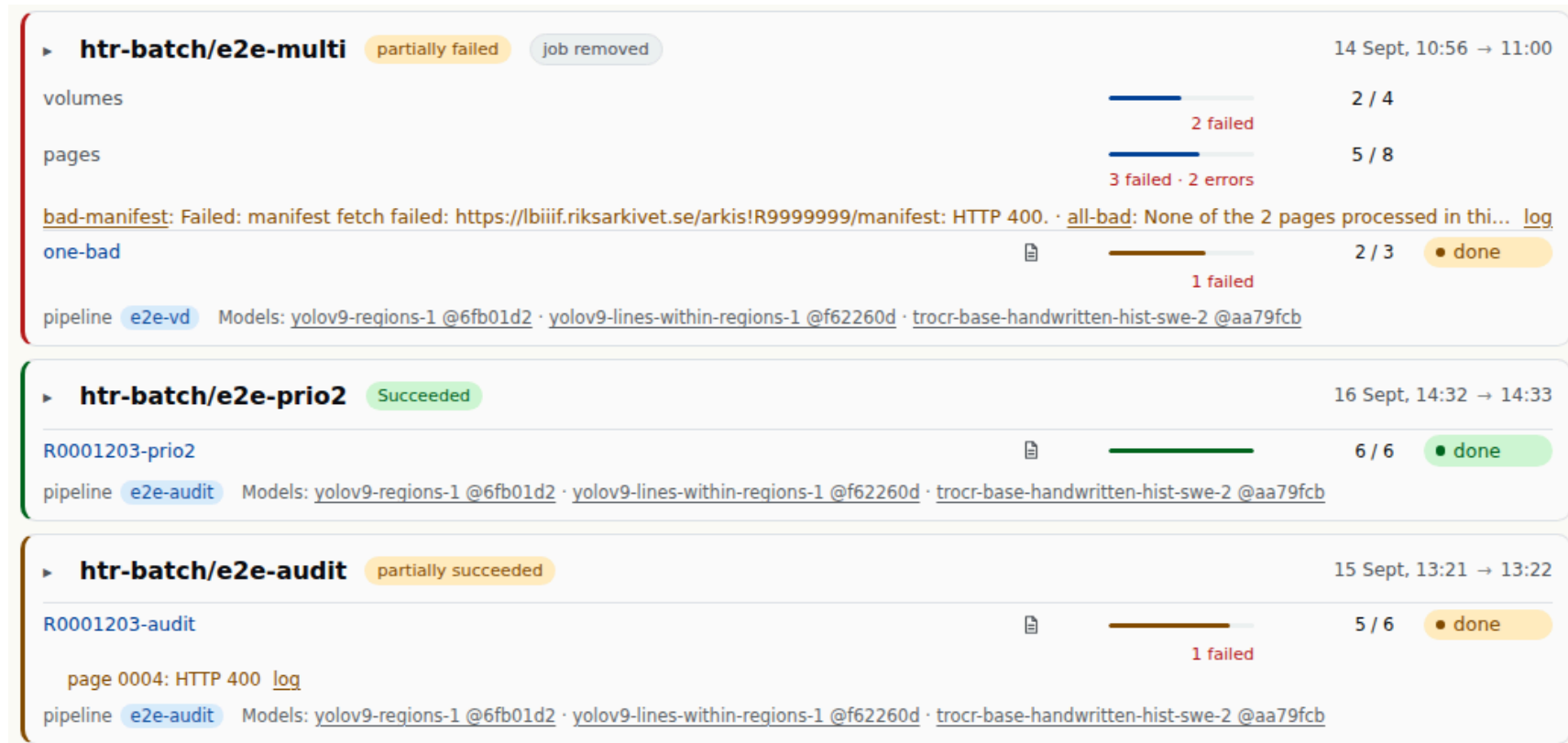
Two files are yours: the campaign, and the pipeline it names — htrflow's `steps:` plus **which image** runs them, pinned by digest.

Validate needs no cluster. The same program runs locally and in the pull request, and says one sentence per problem, naming the file.

Nothing in the cluster reads git. An `apply` renders the repo into Kubernetes objects and sends them. Delete the file, apply with prune, and the Job is gone; the results in the bucket are not.



The status page



One card per campaign, running first, then anything wrong, then finished.



Reading a card

header	the campaign's name, how it stands, and when it was created and finished
state	<i>Queued</i> waiting for GPUs · <i>Running</i> · <i>Paused</i> · <i>Succeeded</i> · <i>partially succeeded</i> some pages lost · <i>partially failed</i> some volumes lost · <i>Failed</i>
extra chips	<i>warm-up</i> the models are not ready yet, or failed to load · <i>job removed</i> finished long ago, results still there
totals	volumes and pages done, with a bar; failures in red under the bar
problems	one sentence per failed volume, saying why
a volume	its name opens the viewer · the braces open its source manifest · the page icon opens its run log · its own bar, count and state · a failed page's reason under the row
footer	the pipeline id and each model with its revision



The run log

[← campaigns](#) raw

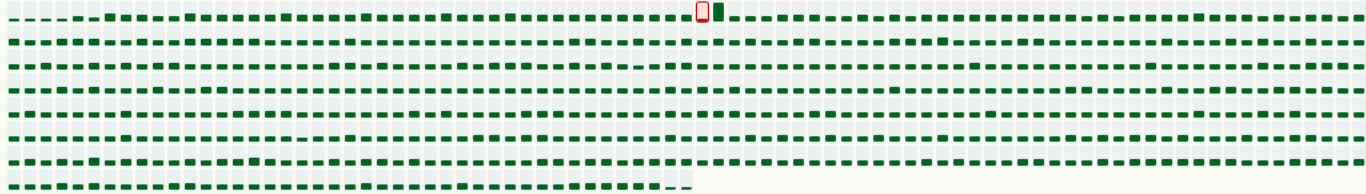
 **Run log · R0001203**

VOLUME	PIPELINE	HTRFLOW	IMAGE	PAGES	TOTAL TIME
R0001203	trocr-large-3char	0.2.6	4d5b93d25390	638 637 ok · 1 failed · 0 skipped	1 h 1 min 23 s wall 1 h 1 min 44 s

PER PAGE · MEDIAN / P95 / MAX
5.7 s / 6.7 s / 18.0 s

SLOWEST PAGES
0045 18.0 s 0018 7.8 s 0032 7.8 s 0007 7.6 s 0526 7.6 s

hover or focus a cell: page id and seconds



1 failed page

0044 page 0044: htrflow's Segmentation (model Riksarkivet/yolov9-lines-within-regions-1) worker thread died; the page is marked failed and the pipeline is rebuilt

[▶ all 638 pages](#)

[▶ pipeline](#)

12:06:17.732 INFO Found credentials in shared credentials file: /secrets/s3/credentials

[▶ 1 HTTP request](#)

12:06:18.060 INFO [R0001203] 638 pages in manifest

Each volume's log: a summary, one cell per page, the failed pages with their reason, and the log itself — updated while the volume runs, and stored in the S3 bucket beside the results for now.

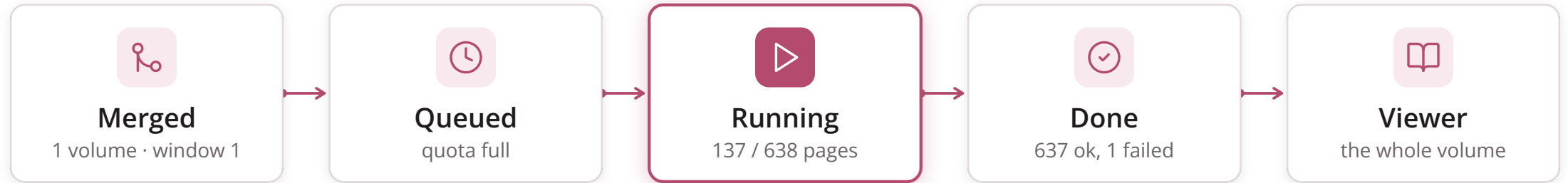


The viewer

Riksarkivet's Universal Viewer 4: the page with every transcribed line outlined, and the text beside it — even while the volume is still running. It ships with the platform's Helm chart, so there is no separate viewer to install.



Follow one campaign



- **Queued.** Another campaign holds the GPUs. The status page shows the campaign with no pod, and says so.
- **Running.** One pod, one GPU. The page count moves every few seconds, and the volume opens in the viewer at page ten.
- **Done, with one failed page.** Page 44 failed and is recorded; the other 637 pages are in the viewer. The card turns amber into green, not plain green, and names the page.
- **What you do about page 44:** nothing, or a new campaign later with a fixed image. Its failure is in `manifest.json` and on the card, and the 637 good pages are in the viewer now.



Any questions?

