

One Compliance Task, 847 Pods, Three Cascading Failures

Governing dynamic, multi-tier AI agent workloads on Kubernetes

847 cumulative Pod executions over 23 hours – (not 847 simultaneously running Pods)

... speaker.identity

Neeraj Pandey

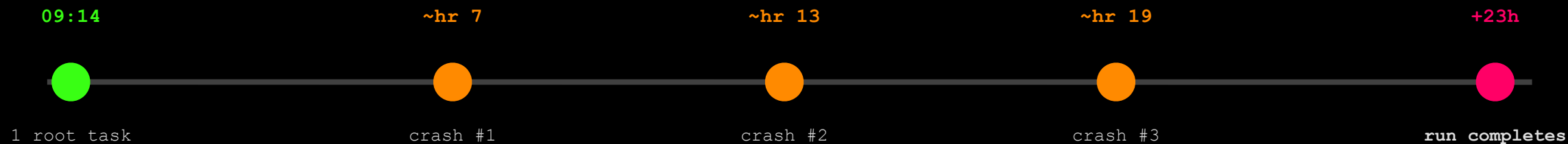
Co-founder & CTO, Lyntcube

AI-native digital workforce for real estate



09:14 → +23H: ONE RUN, THREE CRASHES

A single ESG compliance task began a runtime-discovered execution graph.



847

CUMULATIVE EXECUTIONS

≈200

GPU WORKERS ORPHANED

+2h

PAST SLA

340%

OF ESTIMATED COST

WHAT TURNED SCALE INTO AN INCIDENT

Runtime branching created legitimate scale. Crashes and recovery overlap left prior-generation work active, producing orphaned GPU workers and uncontrolled cost.

847 is cumulative execution volume—not 847 duplicates or simultaneously running Pods.

KUBERNETES CAN CREATE DYNAMIC WORK.

• • • upstream.primitives

WHAT KUBERNETES MODELS WELL

- ▶ Controllers can create workloads dynamically
- ▶ Schedulers reason over declared objects and resources
- ▶ Garbage collection follows valid Kubernetes ownership

• • • application.semantics

WHAT IT CANNOT INFER FOR US

- ▶ Durable root ownership for the original business task
- ▶ One budget across the runtime-discovered execution tree
- ▶ Execution generation: is this child still authorized?

Kubernetes knew whether a Pod was running—but not whether it was still authorized by the original business task.

ONE BUSINESS TASK; A GRAPH DISCOVERED AT RUNTIME

CONTROL PLANE // FIELD NOTES

500-PROPERTY EU TAXONOMY COMPLIANCE PORTFOLIO

DOCUMENTS COMPLETE

OCR → DNSH legal review → done

EPC MISSING / AMBIGUOUS

OCR → data-gap → estimation → legal review

FLOOD RECORD MISSING

OCR → web research → legal review

INTERPRETATION

The graph explains the scale. Missing governance explains the incident.

THIS IMPLEMENTATION

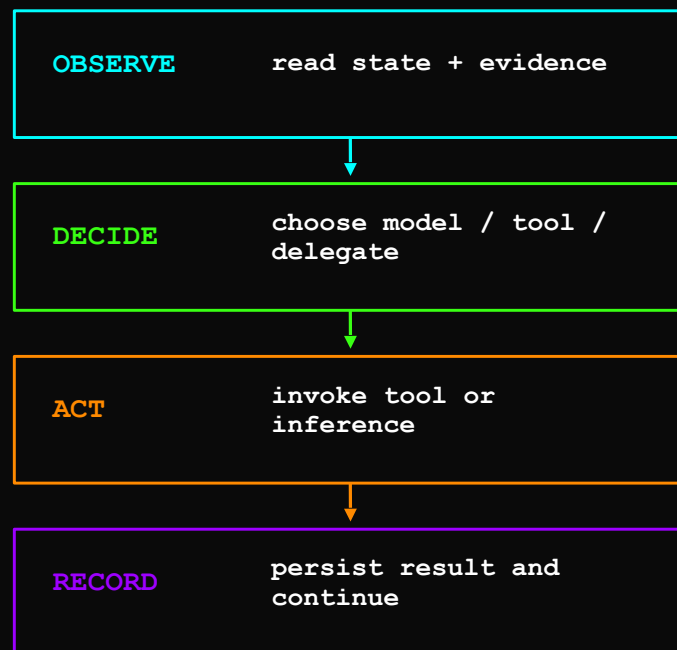
~700 base-path Pods + ~147 evidence-triggered Pods =
847

The evidence-dependent branches were not automatically errors; missing execution governance made recovery unsafe.

AN AI AGENT IS A CONTROL LOOP – NOT A POD

LOGICAL AGENT

role + policy + state + tools



SYSTEM ROLES

different responsibilities

ORCHESTRATOR

owns task + graph

AGENT

makes bounded decisions

WORKER

performs a unit of work

MODEL SERVER

executes inference

POSSIBLE RUNTIME MAPPINGS

chosen by isolation, scale and resources

CPU PROCESS

inside an existing service

QUEUE ITEM

shared worker pool

JOB / POD

isolated execution unit

INFERENCE REQUEST

shared GPU model server

GPU JOB

dedicated or distributed model work

A sub-agent becomes a separate Kubernetes Job only when isolation, scaling or resource needs justify it.

FIVE QUESTIONS, FIVE LAYERS

01 SHOULD IT (child) EXIST?

→ Agent Budget Controller

02 CAN IT RUN?

→ Kueue

03 WHERE SHOULD IT RUN?

→ kube-scheduler

04 WHICH DEVICE DOES IT GET?

→ DRA / GPU stack

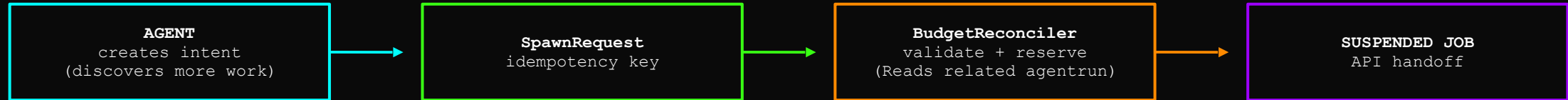
05 IS IT STILL VALID?

→ Lifecycle Controller

ONE CHILD REQUEST, TRACED THROUGH REAL OBJECTS

AGENT CONTROL PLANE — OURS

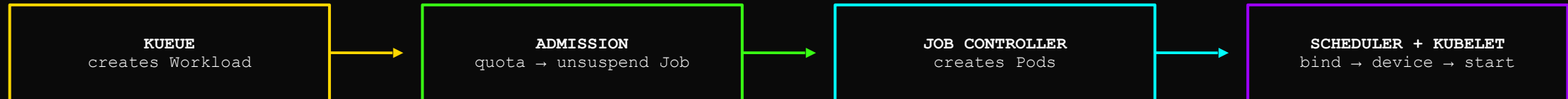
authority · epoch · depth · economic envelope



OBJECT HANDOFF

KUBERNETES CONTROL PLANE — UPSTREAM

Kueue admission · Job materialization · Pod placement · node execution



DEFAULT-DENY EXECUTION AUTHORITY

Agents may create `SpawnRequest`—not `Pod`, `Job`, `Workload` or `ResourceClaim`.

JOB LIFECYCLE

Kueue unsuspends Jobs; the Job controller creates Pods.

KUBERNETES 1.37 STRENGTHENS SCHEDULING – NOT AGENT AUTHORITY

UPSTREAM CONTROL PLANE

capacity, placement and device health

Workload + PodGroup	v1beta1
Gang scheduling	BETA
Workload-aware preemption	BETA
CompositePodGroup	ALPHA
DRA device taints	STABLE

APPLICATION CONTROL PLANE

recursive causal + economic authority

DYNAMIC CHILD SET
discovered while documents execute
ROOT ENVELOPE
GPU-seconds · cost · depth · deadline
EXECUTION AUTHORITY
should this specific child exist?
CAUSAL LIFECYCLE
epoch · orphan cleanup · cancellation
PRECISE LIMIT
CompositePodGroup adds alpha hierarchy; it does not discover children or enforce cost and depth.

Upstream owns workload scheduling. AgentRun owns whether recursively discovered work has authority to exist.

01

PROBLEM 01

Hierarchical Budget Control

Capping agent trees at the root envelope, not the flat namespace limit.

A NAMESPACE QUOTA CAN'T SEE WHAT WE NEED IT TO SEE

```
gpu-quota.yaml
01  apiVersion: v1
02  kind: ResourceQuota
03  metadata:
04    name: gpu-quota
05    namespace: customer-a
06  spec:
07    hard:
08      requests.nvidia.com/gpu: "32"
09      pods: "500"
9 LINES // REVIEWED
```

WHAT THE OBJECT SEES

Task A: 16 GPUs · Task B: 2 · Task C: 4 · aggregate only

WHAT IT CANNOT MODEL

run ancestry · recursion depth · deadline · economic authority

ResourceQuota has no field for 'should this exist' — only 'is there room.'

```
root.execution.envelope
```

THE REAL BUDGET FOR THIS RUN

MAX CONCURRENT GPUS	24
MAX GPU-HOURS	12
MAX CHILDREN	900
MAX EXECUTION DEPTH	5
COST CAP / DEADLINE	€3,200 · 09:14Z

None of this is expressible in a ResourceQuota object.

THE ROOT EXECUTION ENVELOPE

PROBLEM 1 · BUDGET CONTROL

CONTROL PLANE // FIELD NOTES

● ● ● compliance-run-8841.yaml

YAML

```
01 kind: AgentRun
02 metadata:
03   name: compliance-run-8841
04 spec:
05   budget:
06     gpuSeconds: 43200
07     maxConcurrentGpu: 24
08     maxChildren: 900
09     maxDepth: 5
10     costCapEUR: 3200
11     deadline: "2026-08-15T06:14:00Z"
12 status:
13   reservedGpuSeconds: 8200
14   consumedGpuSeconds: 17600
15   runningChildren: 214
16   phase: Active
```

16 LINES // REVIEWED

SPEC = AUTHORITY

GPU-seconds · concurrency · depth · children · cost ·
deadline

STATUS = LEDGER

reserved → consumed / released / expired

STATUS OWNERSHIP

Only the controller patches accounting state

PERSISTENCE

Budget survives controller restart and leader change

AGENTS NEVER PATCH `AgentRun.status`

WHERE KUEUE'S OBJECTS END AND OUR AGENTRUN BEGIN

```

● ● ● compliance-cq.yaml
01  apiVersion: kueue.x-k8s.io/v1beta2
02  kind: ClusterQueue
03  metadata:
04    name: compliance-cq
05  spec:
06    cohort: real-estate-ai
07    resourceGroups:
08    - coveredResources: ["nvidia.com/gpu"]
09      flavors:
10      - name: h100-mig-3g40gb
11        resources:
12        - name: nvidia.com/gpu
13          nominalQuota: 24
  
```

13 LINES // REVIEWED

CLUSTERQUEUE

capacity envelope

LOCALQUEUE

tenant entry point

WORKLOAD

admission unit

RESOURCEFLAVOR

GPU class + placement constraints

- **LocalQueue**: namespace-scoped entry point referencing a ClusterQueue.
- **ClusterQueue**: cluster-scoped capacity, quota, flavor, cohort, and policy boundary.
- **Cohort**: permits controlled borrowing/lending and fair sharing among ClusterQueues.
- **ResourceFlavor**: maps resource quota to node labels/taints or a device class; it is not itself a physical GPU.

● ● ● boundary

KUEUE

capacity, fairness, admission

AGENTRUN

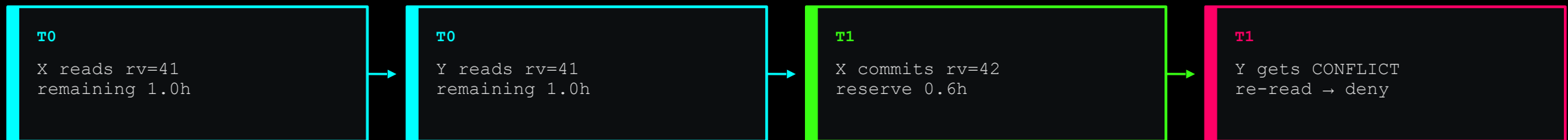
depth, cost, deadline, authority

Kueue arbitrates scarce capacity across workloads. AgentRun arbitrates authority inside one execution tree.

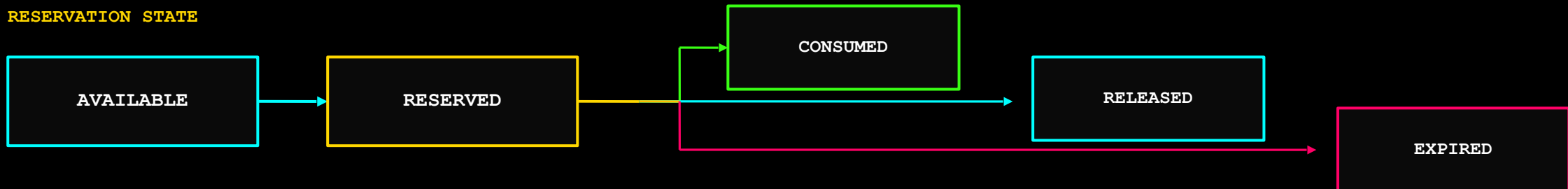
WHERE A BUDGET CONTROLLER CAN QUIETLY DOUBLE-BOOK ITSELF

$$\text{CONSUMED} + \text{RESERVED} + \text{REQUESTED} \leq \text{ROOT GPU-SECOND BUDGET}$$

SAME INITIAL READ; ONLY ONE COMMIT



RESERVATION STATE



WHAT MAKES IT SAFE

resourceVersion CAS prevents conflicting commits. Reservation TTL recovers budget if the reserver dies. Epoch fencing rejects stale leaders.

Leader election reduces duplicate work; CAS and fencing protect correctness when leadership is ambiguous.

CORRECTNESS IS FIVE INVARIANTS — NOT FIVE COMPONENTS

CONTROL-PLANE CONTRACT

properties that survive concurrency and process failure

01 **BUDGET** `reserved + consumed ≤ root.gpuSeconds`

02 **CONCURRENCY** `running GPU children ≤ maxConcurrentGpu`

03 **DEPTH** `child.depth ≤ root.maxDepth`

04 **ONE-TO-ONE** `one admitted child ↔ one reservation`

05 **FENCING** `request.epoch < currentEpoch → reject`

ENFORCEMENT MAP

mechanisms may change; guarantees may not

ROOT ENVELOPE

`budget + depth`

CAS

`one reservation commit`

KUEUE

`cluster capacity`

EPOCH

`stale writer rejection`

RECONCILE

`eventual convergence`

Mechanisms can change. These invariants are the contract.

02

PROBLEM 02

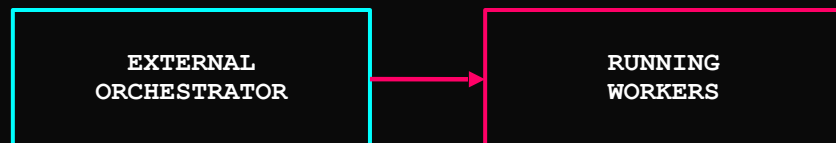
Orphan Detection & Cleanup

Recovering execution chains when the orchestrator that spawned them is gone.

USE ownerReferences WHERE VALID—LINEAGE WHERE THEY AREN'T

ORIGINAL FAILURE

external root had no Kubernetes UID



NO VALID OWNER UID

A crashed external root gave Kubernetes nothing to cascade-delete.

TTL DOES NOT APPLY

`ttlSecondsAfterFinished` cleans terminal Jobs;
It starts only after a Job becomes Complete or Failed,
while these workers were still Running.

CORRECTED DESIGN

native ownership plus independent recovery signals



PRIMARY

`ownerReferences`: AgentRun → Job → Pods wherever scope rules permit.

DEFENSE IN DEPTH

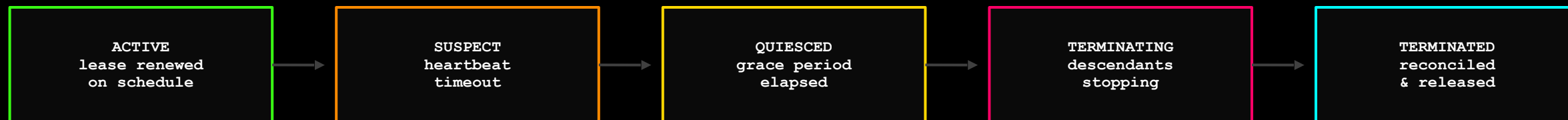
`run-id + parent + epoch labels` · Lease · periodic sweeper

SCOPE RULE

Cross-namespace owner references are invalid; do not replace ownership with labels when valid ownership is available.

Labels preserve causal identity. ownerReferences drive native garbage collection. The lifecycle controller handles what neither can infer.

DETECTING 'THE ORCHESTRATOR IS GONE,' CORRECTLY



FENCING, NOT JUST TIMEOUT — 'NO HEARTBEAT FOR 30S' ISN'T AN ANSWER ON ITS OWN

```

orchestrator-lease.yaml
01  apiVersion: coordination.k8s.io/v1
02  kind: Lease
03  metadata:
04    name: run-8841-orchestrator
05    annotations:
06      agents.lyntcube.io/epoch: "18"
07  spec:
08    holderIdentity: orchestrator-b
09    renewTime: "2026-08-14T09:41:07Z"

ACTIVE → SUSPECT → QUIESCED → TERMINATING → TERMINATED
9 LINES // REVIEWED
  
```

```

fencing.go
01  func fenced(incoming, current int64) error {
02    if incoming < current {
03      return ErrStaleEpoch
04      // orchestrator-a talking after
05      // orchestrator-b took over
06    }
07    return nil
08  }
8 LINES // REVIEWED
  
```

LIVENESS SIGNAL

Lease expiry moves ACTIVE → SUSPECT. Timeout starts investigation; it does not grant takeover authority.

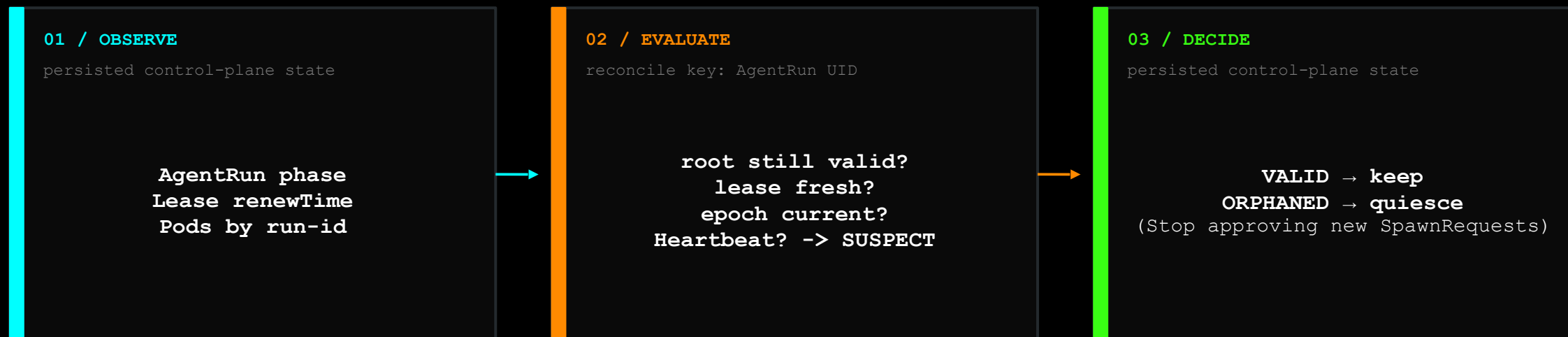
AUTHORITY SIGNAL

Epoch fencing rejects writes from orchestrator-a after orchestrator-b takes over at epoch+1.

BEFORE 200 orphaned workers, found manually.

AFTER SUSPECT within one lease interval; reconciled in AgentRun.status.

FROM ANCESTRY TO CLEANUP: WHAT ACTUALLY WATCHES



BOUNDED, IDEMPOTENT CLEANUP



SAFETY RULE // timeout creates SUSPECT; persisted phase + current epoch authorize destructive cleanup

Ownership and labels identify descendants; Lease + epoch establish authority; the reconciler performs cleanup.

03

PROBLEM 03

GPU Burst Scheduling

Correlated demand from concurrent agent populations, not independent arrivals.

'THE SCHEDULER TOOK 40 MINUTES' WAS THE WRONG METRIC

01

AUTHORITY + ADMISSION

child ready → budget → Workload → quota → admitted

BudgetReconciler + Kueue

02

PLACEMENT + PROVISION

Pod eligible → node selected → node ready

scheduler + node autoscaler

03

DEVICE PREPARATION

GPU allocation → container start

device plugin / DRA + kubelet

04

MODEL WARM-UP

container started → inference ready

application-not Kubernetes

TIMESTAMP EACH OWNERSHIP BOUNDARY; OPTIMIZE THE COMPONENT THAT OWNS THE DELAY

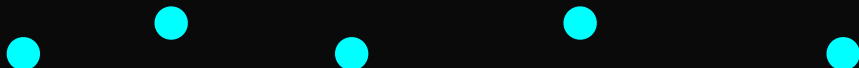
kube-scheduler compute time was a rounding error; uncontrolled admission created the queue.

THE DEMAND PATTERN THAT BREAKS FLAT ADMISSION

• • • normal.web.traffic

NORMAL WEB TRAFFIC

- ▶ Independent users
- ▶ Roughly Poisson-distributed arrivals
- ▶ One user's request tells you nothing about the next one



• • • compliance.run.burst

THIS COMPLIANCE RUN

- ▶ One OCR pass flags a batch of properties as ambiguous
- ▶ Their Legal + Inference agents fire in the same window
- ▶ ~300 correlated GPU requests land together, not spread out



The requests aren't independent — they share a common ancestor and a common trigger.

ADMIT A SLICE. QUEUE THE REST AS LOGICAL WORK.

... before.optimistic-admission

CREATE 300 PODS — LET K8S HANDLE IT

API server, scheduler queue, autoscaler and
quota all contend simultaneously.

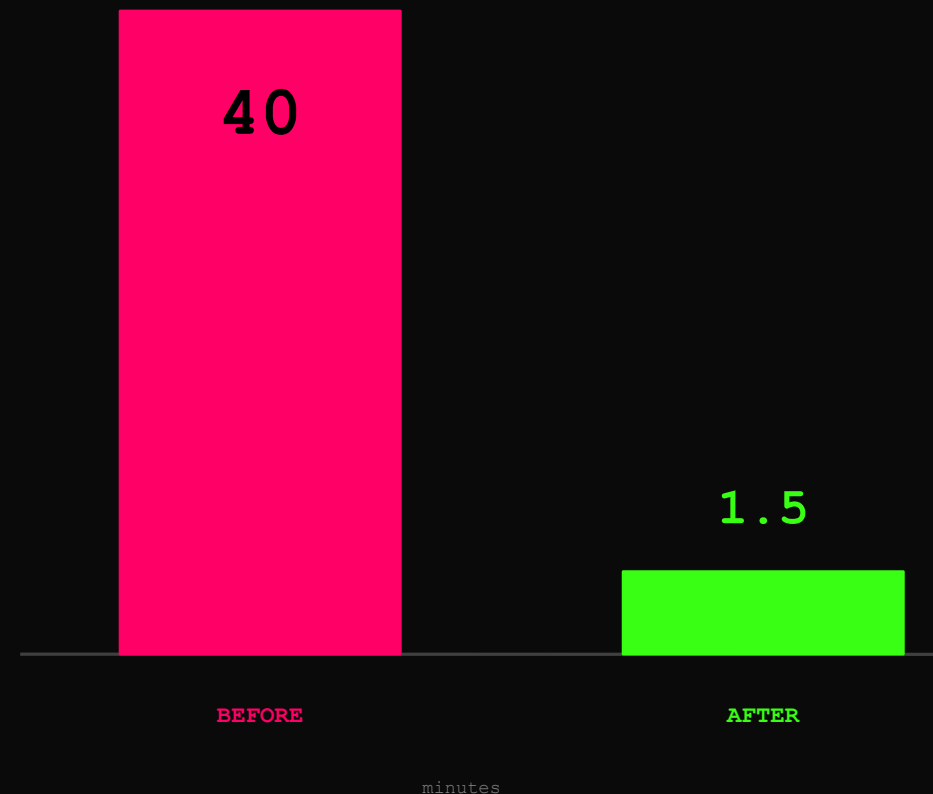
... after.bounded-admission

ADMIT 20 · HOLD 280 AS LOGICAL WORK

The budget controller releases more as GPU
capacity actually frees up.

... submission-to-running.minutes

SUBMISSION → RUNNING: 40 MIN → UNDER 90 SEC



PROBLEM 04

04

MIG Partitioning

A GPU resource and fault-isolation boundary inside a broader multi-tenant security model.

WHERE THE ISOLATION BOUNDARY ACTUALLY SITS



For our threat model, MIG supplied the GPU resource and fault-isolation boundary—alongside host, network, storage and IAM controls.

FULL GPU vs. TIME-SLICING vs. MIG

mode.01

FULL GPU

Exclusive device

Strong isolation
low packing efficiency

mode.02

TIME-SLICING

Shared device

Best packing
no hard memory partition

mode.03

MIG

Dedicated compute + memory

Predictable slices
fixed geometry + fragmentation

REFERENCE H100 PROFILE SIZING

1g.10gb

OCR / extraction

2g.20gb

research / data
gap

3g.40gb

legal reasoning

7g.80gb

large-batch
inference

SIZE FROM WORKING SET

model weights + KV cache(context × batch) + workspace + headroom

DELIVERY MODEL

Static device-plugin profiles today; DRA enables claim-time selection. Dynamic MIG creation remains alpha.

MIG was our device-isolation choice—not a universal regulatory requirement.

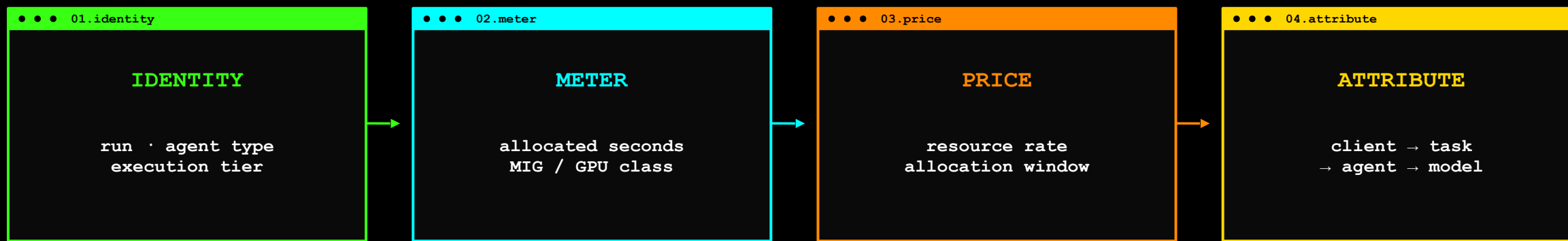
05

PROBLEM 05

Cost Attribution & FinOps

*GPU-hours tagged to client, task, agent type and execution tier –
without cardinality explosion.*

FROM GPU-SECOND TO €—WITH A CAUSAL OWNER



DO NOT COLLAPSE THESE STATES



Idle / warm cost is allocation minus useful usage; every amount rolls up through causal lineage.

ALERT ON BUDGET BURN—WITHOUT EXPLODING CARDINALITY

• • • dont.prometheus-cardinality

DON'T DO THIS

- ▶ task_id, agent_id, document_id, run_id as Prometheus labels
- ▶ 100 clients × 10,000 tasks × 100 agents = millions of series

• • • do.separate-signal-types

DO THIS

- ▶ Metrics: agent_type, resource_class, status, tenant_tier
- ▶ Traces: root run → agent span → child span
- ▶ Logs / DB: task_id, agent_id, document_id

• • • budget-burn.contract

BURN RATIO

actual cumulative GPU-seconds

> 1.4

expected cumulative GPU-seconds at this point

PER-RUN DECISION

Evaluate in the cost ledger or stream processor where run-id cardinality belongs.

FLEET SIGNAL

Export low-cardinality aggregates to Prometheus for operational alerting.

Use rate() for counters—not a fixed estimate gauge. Keep per-run IDs out of Prometheus labels.

CLOSING ANALYSIS

The Postmortem

Three reasonable decisions let legitimate runtime scale become unsafe under crash recovery.

THREE DECISIONS, EACH DEFENSIBLE IN ISOLATION

01

OWNERSHIP OUTSIDE KUBERNETES

WHY The run began in an external workflow system.

IMPACT Crash left no Kubernetes root to cascade-delete 200 running workers.

02

FLAT NAMESPACE QUOTA

WHY It was simple and matched ordinary workloads.

IMPACT Runtime-discovered complexity stayed invisible until GPUs were committed.

03

OPTIMISTIC POD CREATION

WHY It worked while fan-out remained small.

IMPACT ~300 correlated requests contended across API, quota and capacity.

The failure lived in the interaction between locally reasonable decisions.

HOW THREE SOUND DECISIONS BECAME ONE INCIDENT

INCIDENT REPLAY — SYSTEM STATE OVER TIME



COMPOUNDING IMPACT

EXECUTIONS 1 root → 847 cumulative	ORPHANS 0 → 200 GPU workers	COST 100% → 340%	ALERT STATE all objects legitimate
--	---------------------------------------	----------------------------	--

None was wrong in isolation. Together, under correlated load, they compounded.

FIVE DECISIONS. ONE CONTROL PLANE.

SHOULD IT EXIST?

→ Agent Budget Controller

CAN IT RUN?

→ Kueue

WHERE SHOULD IT RUN?

→ kube-scheduler

WHICH DEVICE?

→ DRA / GPU stack

IS IT STILL VALID?

→ Lifecycle Controller

*Kueue manages capacity.
Our controller manages
execution authority.*

DENY is final.
WAIT is provisional.

THANK YOU — QUESTIONS.

Neeraj Pandey · Co-founder & CTO, Lyntcube

The Scheduler Has Never Heard of NVLink

13:30 · Room 603+604