# Anthropic's Kubernetes Scheduler for Frontier AI



**Decheng Dai**
Anthropic

KubeCon + CloudNativeCon North America
Salt Lake City, November 2026

The scale

# Our compute landscape



We run on several clouds and chip types, all on Kubernetes.

The scale

# Our compute landscape

Clouds

AWS, Google Cloud, Azure, xAI, and more

Chips

NVIDIA GPUs
AWS Trainium
Google TPUs
AMD GPUs

Regions

U.S. + international

Physical scale

## A very large fleet

of chips across our clouds

Kubernetes scale

## Very large clusters

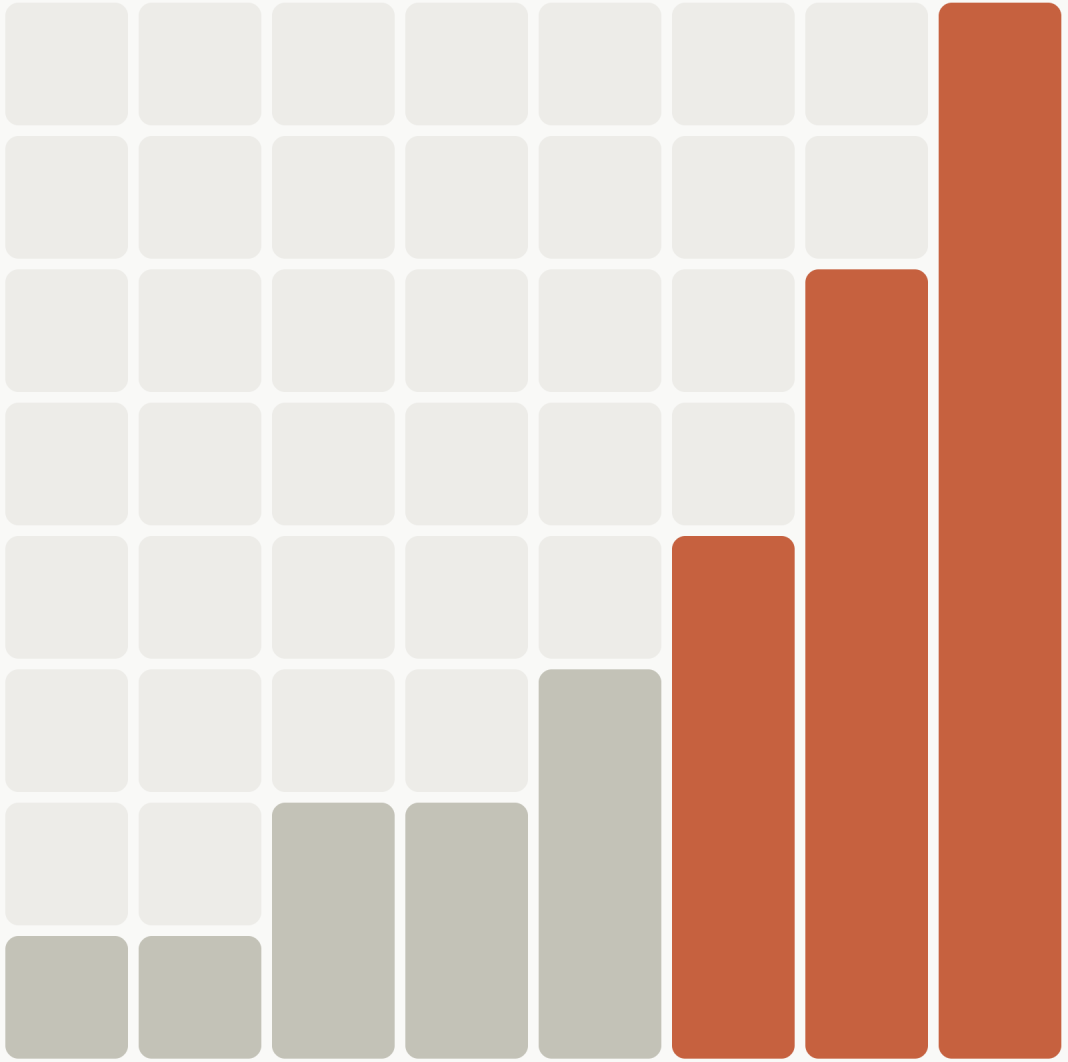
That's what this scheduler has to handle.

AWS is Anthropic's primary cloud provider.

The history

# How we got here

What we tried with the default scheduler before building our own.



The history

# How we got here

We started with the default scheduler and worked around its limits.

A

## Micro priorities

Give each job a slightly different priority, so its pods stay together in the queue

B

## Scheduling gates

Hold pods back from the default scheduler until our own code has picked their nodes

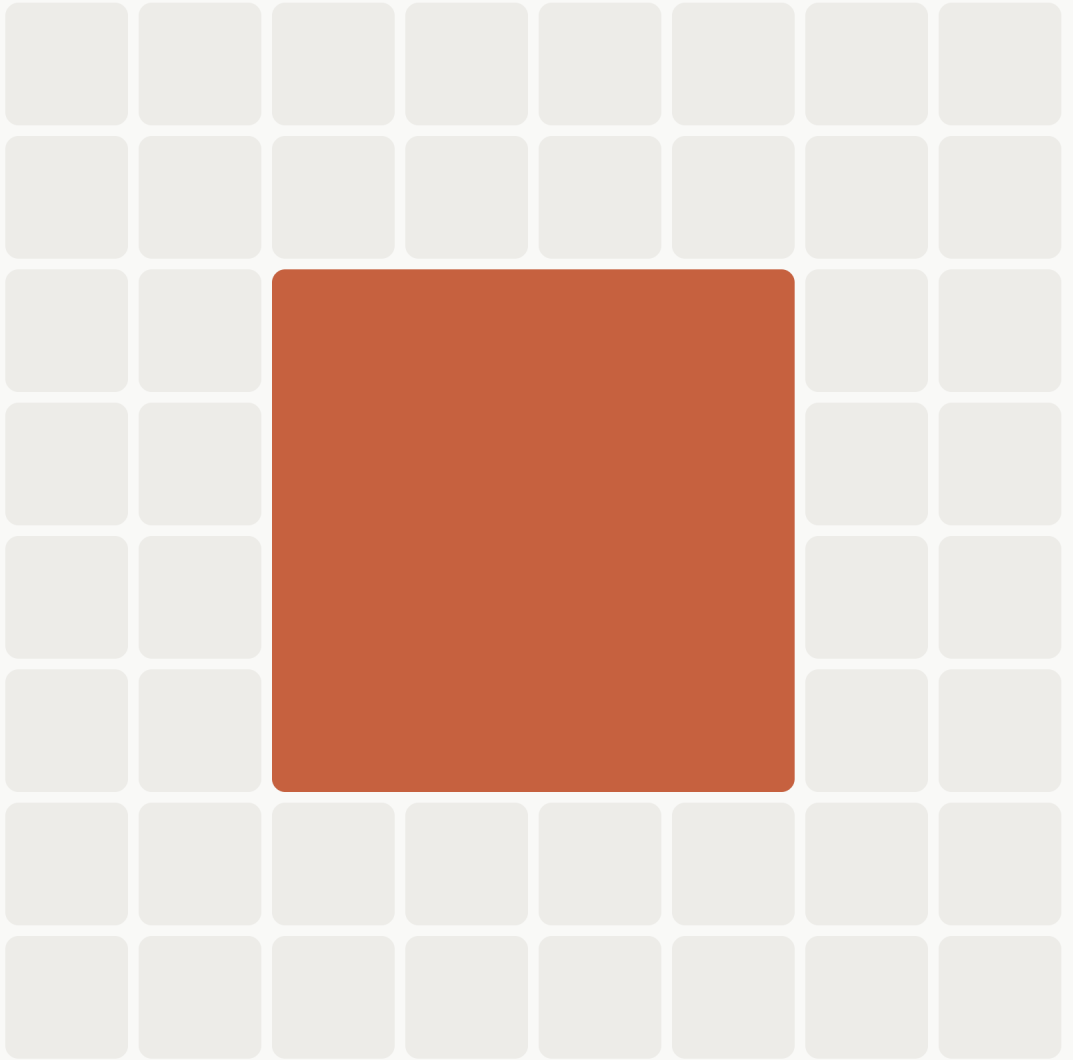
C

## TPU topology scheduling

Figure out where each chip is, then place jobs based on that

The problems

# Gang and topology



A job needs all its pods at once, in the right part of the network.

The problems

# Gang scheduling

Most large ML jobs need many pods running at the same time. If only some get placed, the job holds chips and does nothing.

## Full gang



All the pods have to be running or the job can't start. Here two are missing, so the other six sit idle.

## Independent replicas



Lots of small gangs. If a pod is missing, only its replica waits. The rest keep running.

The problems

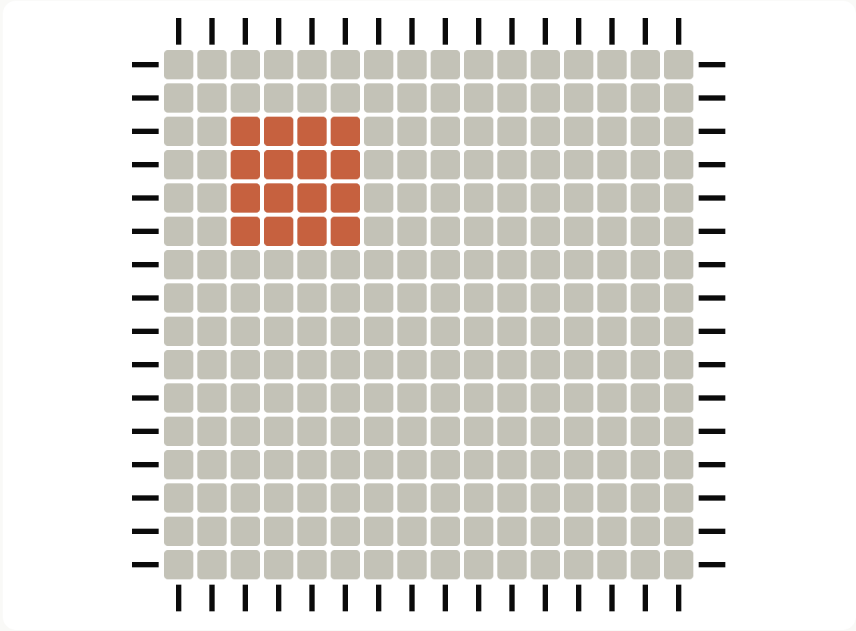
# Topology

A gang wants its pods close together on the network, but "close" means something different on every platform.

| | Google TPU (Ironwood) | AWS Trainium2 | NVIDIA GB200 |
|---|---|---|---|
| Node | **4 chips**<br>Per host | **16 chips**<br>4x4 2D torus | **4 GPUs**<br>Per node |
| Fast domain | **64 chips**<br>4x4x4 cube. Optical switches connect up to 144 cubes, and slices are formed at run time. | **64 chips**<br>UltraServer: 4 nodes joined by NeuronLink in a ring. | **72 GPUs**<br>NVL72 rack over NVLink. Cliques split it into smaller domains. |
| Network above | **Superblocks**<br>Google's Jupiter network | **Network layers**<br>3 or 4 layers of EC2 network nodes | **Data halls**<br>For example, xAI Colossus |

The problems

# TPU v5e topology



Orange: a 4x4 slice. Black ticks: wraparound links, only on the full 16x16. Based on Google DeepMind, “How to Scale Your Model.”

A 4x4 job needs 16 connected chips, not just any 16 free chips.

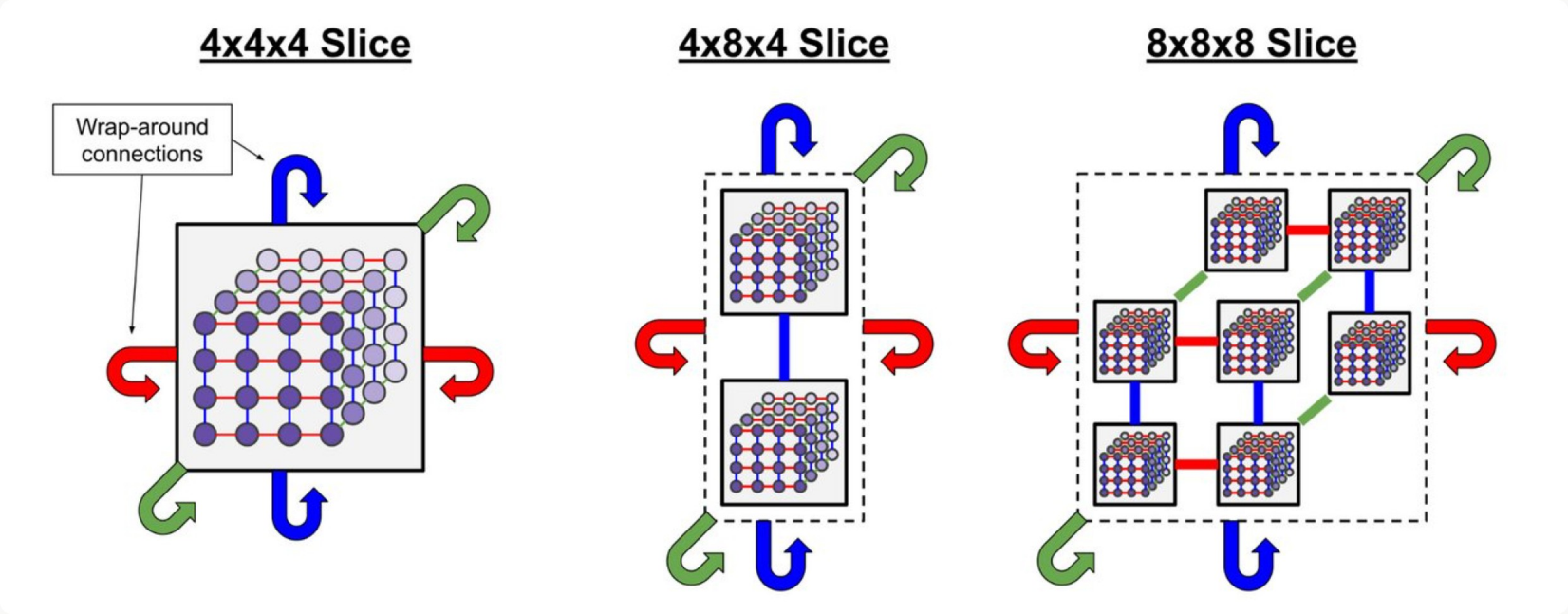
| Slice | Chips | Hosts |
|---|---|---|
| 4x4 | 16 | 4 |
| 8x8 | 64 | 16 |
| 16x16 | 256 | 64 |

4 chips per host. Each chip links to up to 4 neighbors.

The problems

# TPU v7 topology

A job asks for a shape of connected chips, like 4x8x4. Dynamic slicing wires up that shape when the job is scheduled.




**4**
chips per host

**64**
chips per 4x4x4 cube

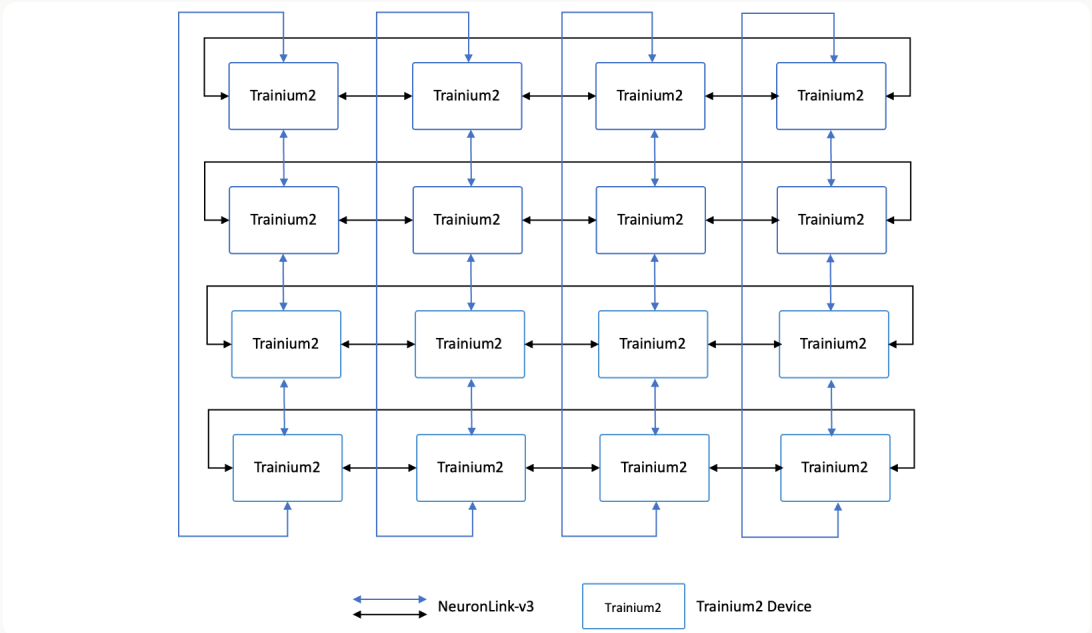
**9,216**
chips per Pod, 144 cubes

Source: Google Cloud, “Inside the Ironwood TPU codesigned AI stack”

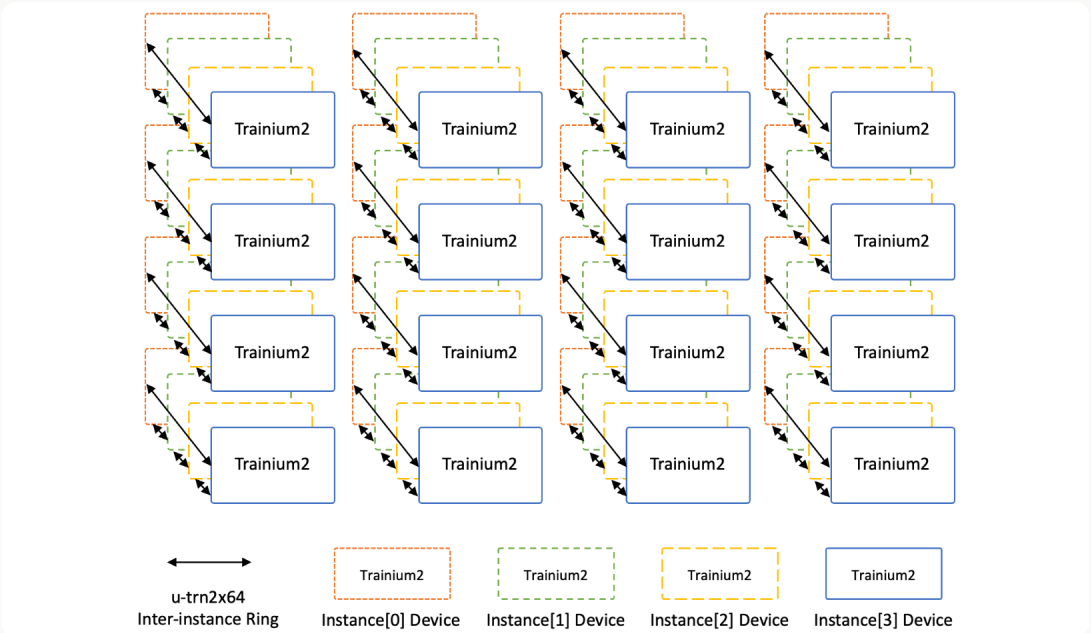
The problems

# Trainium topology

Four trn2u nodes make one UltraServer.




One node: 16 chips in a 4x4 2D torus

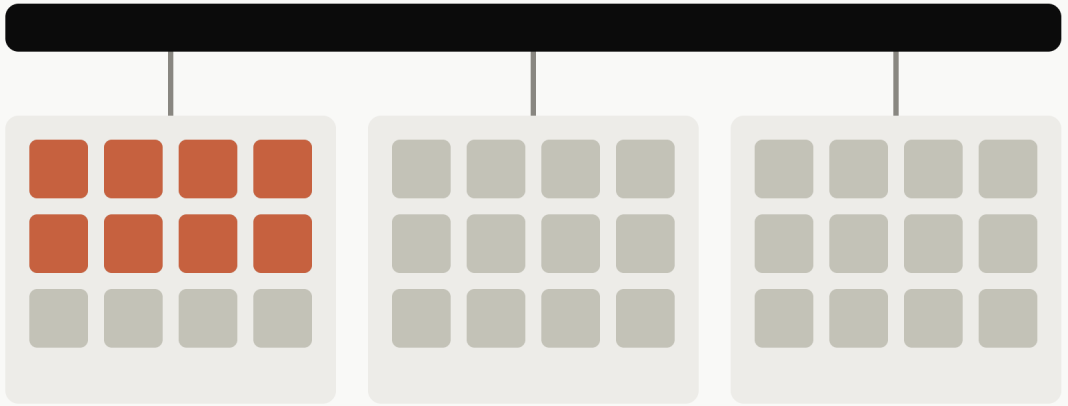



One UltraServer: 4 nodes, 64 chips, joined in rings

Source: AWS Neuron documentation, “Amazon EC2 Trn2 Architecture”

The problems

# Network topology

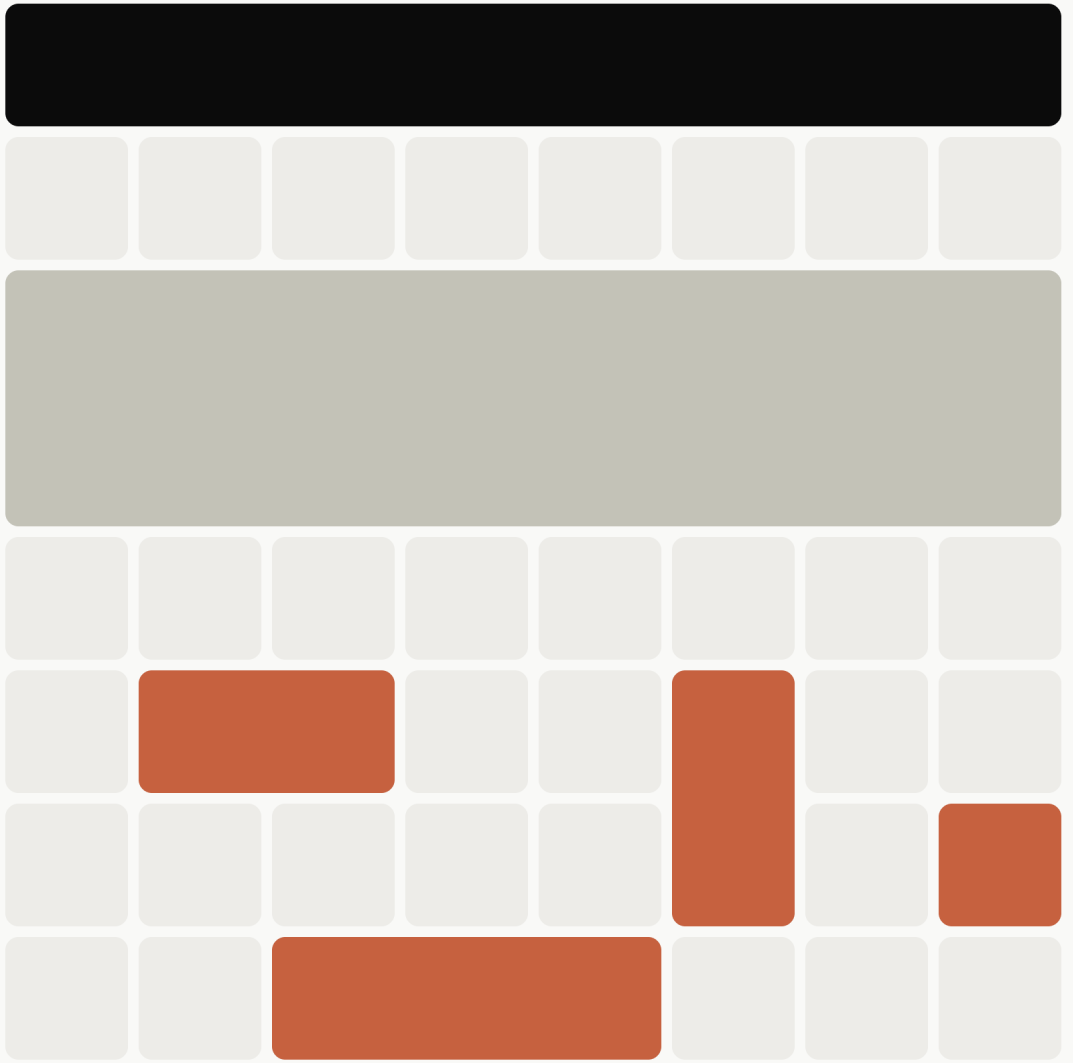


Pods in the same part of the network are fewer hops apart, so we try to keep a gang together.

| | |
|---|---|
| **Google** | Superblocks in the Jupiter network |
| **AWS** | 3 or 4 layers of network nodes |
| **xAI Colossus** | Data halls |

The design

# Architecture

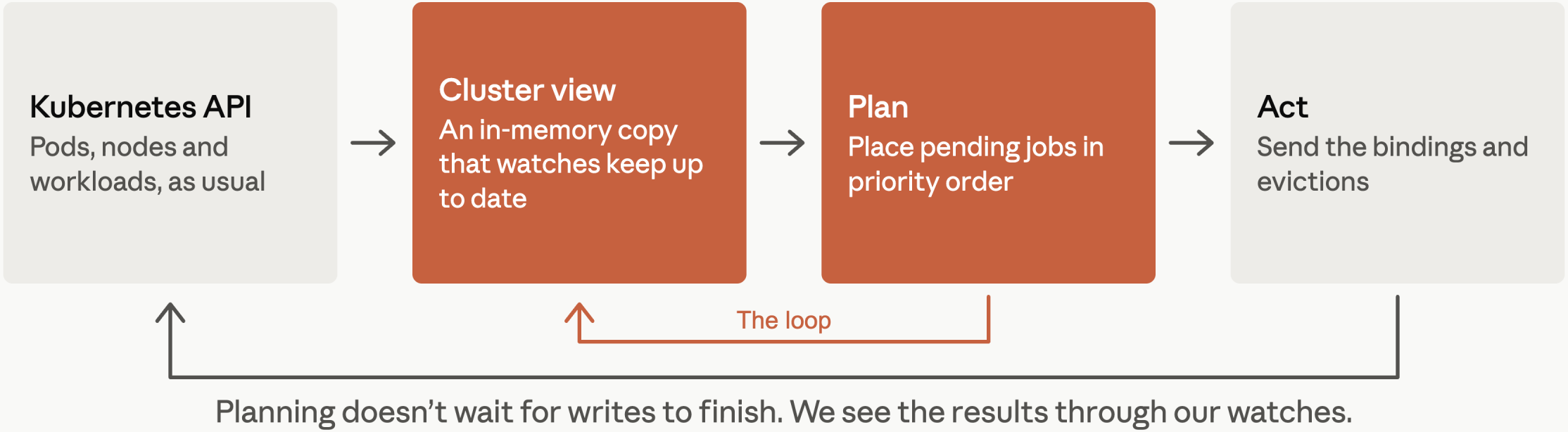


We watch the Kubernetes API, plan in memory, and send bindings separately.

The design

# The control loop

The loop only covers the cluster view and planning. Writes to the Kubernetes API happen outside it.




Planning doesn't wait for writes to finish. We see the results through our watches.

The design

# Design principles

## The model

1. Replace only the scheduler
2. Decide for the whole job
3. Score preemption with binpacking

## How it runs

4. One binary, three roles
5. Schedulers talk via nominated nodes

The design

# Replace only the scheduler

Workloads and controllers

API server

**Our scheduler**

Kubelet and device plugins

```
kind: StatefulSet
spec:
  template:
    spec:
      schedulerName: our-scheduler
```

A workload picks our scheduler by name. The API server, the kubelet and everything else work as usual.

**Upside**

Jobs are still normal workloads, so existing controllers and tools still work.

**Downside**

It can conflict with the default scheduler in the cluster.

The design

# Decide for the whole job

### One pod at a time



Two six-chip jobs each hold four chips. Neither can run.

### One job at a time



One job runs. The other waits and holds nothing.

We schedule jobs, not pods. The queue is ordered by job, and a gang gets placed whole or not at all.

**Upside**

A job holds no chips until all its pods fit, so gangs can't deadlock.

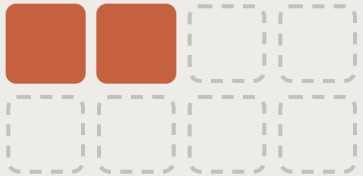
**Downside**

Disruptions like evictions hit the whole gang.

The design

# Score preemption with binpacking

**Free chips**



Breaks up an empty shape.

**Chosen**



Evicts low-priority work, but the empty shape stays intact.

Free chips aren't always the best place for a job. A small job may evict low-priority work to keep a large shape free.

**Upside**

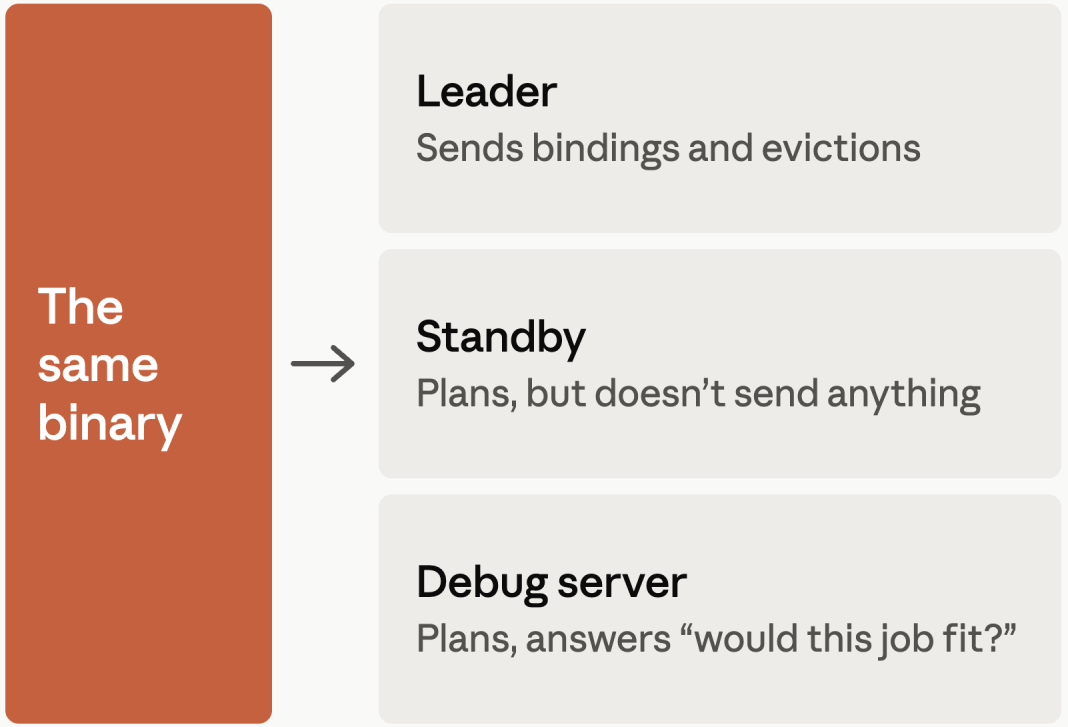
Large jobs can still find whole shapes.

**Downside**

A job can get evicted while free chips sit idle.

The design

# One binary, three roles




All instances read from the API server and plan, but only the leader sends mutating API calls.

**Upside**

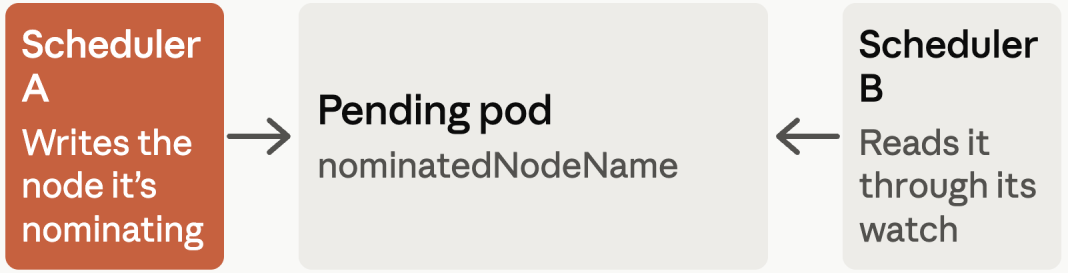
The standby is always warm.

**Challenge**

Leader handover is still hard.

The design

# Schedulers talk via nominated nodes




We run multiple schedulers. One writes its intent to the NNN field on the pod, and the others leave that room free.

**Upside**

No new API or shared state, and each scheduler can restart or change on its own.

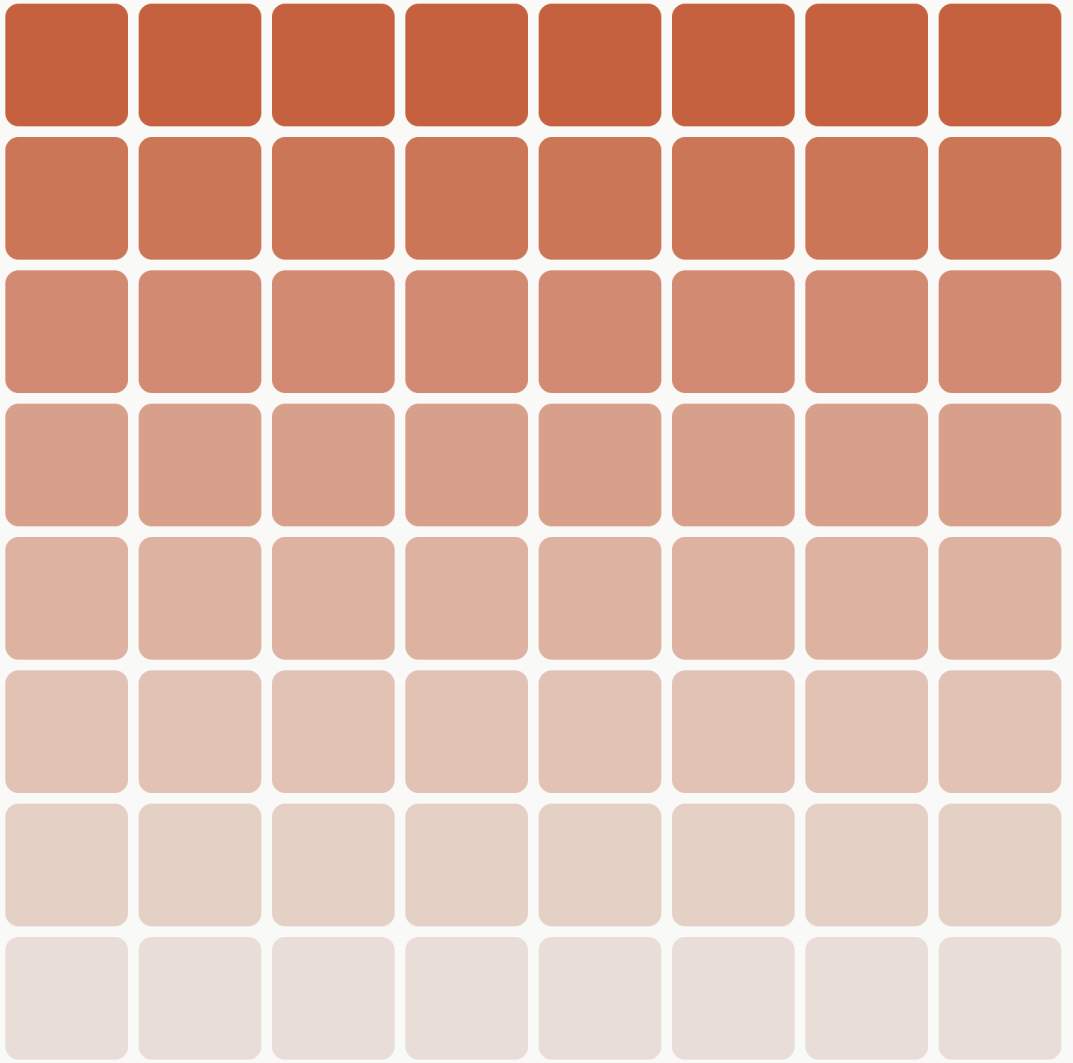
**Downside**

It's unclear when a hold has to expire.
Autoscalers won't add a node for a nominated pod.

Making it fast

# Scalability

How we made each scheduling pass faster.



Making it fast

# The cost of a pass

The work in a pass

**pending jobs × nodes**

Our clusters are very large, and we replan on every pass. That only works if a pass is fast.

| | |
|---|---|
| **Memory** | Keep only the fields we read |
| **Snapshots** | Snapshot without copying |
| **Parallelism** | Serial decisions, parallel scoring |
| **Caching** | One scan for same-shape jobs |
| **Replay** | Replay unchanged plans |
| **API writes** | Protect the API server |

Making it fast

# Keep only the fields we read

We decode pods and nodes into small types with only the fields the scheduler uses.

**What the API sends**



**What we keep**



Fields we read  managedFields  Everything else

Not to scale.

**Upside**

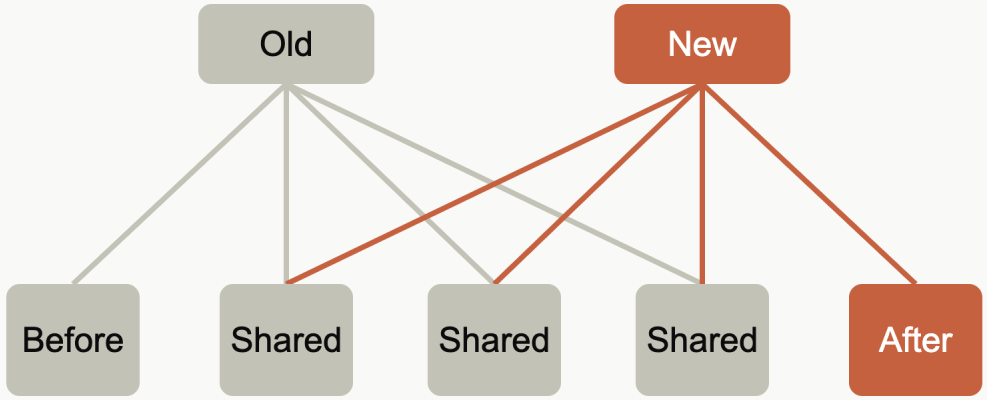
Memory consumption is much smaller than the total object size.

**Downside**

To read a new field you have to add it to the small type first. If you forget, it's just empty.

Making it fast

# Snapshot without copying




When one item changes, we copy only that item and the path to it. Old and new snapshots share the rest, so a pass can keep reading the old one.

A pass needs a view of the cluster that stays still while it runs. We use a persistent map: each snapshot reuses entries from the last one.

**Upside**

A snapshot costs almost nothing, and watches can keep updating during a pass.

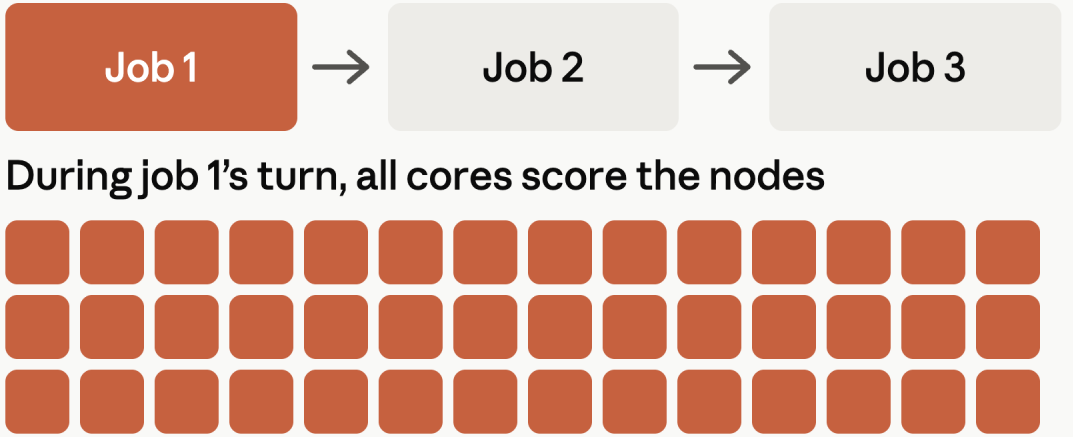
**Downside**

Each pass still builds its own working data, and freeing it takes time.

Making it fast

# Serial decisions, parallel scoring

One job at a time




Jobs go one at a time, in priority order, and each one sees the placements before it. During a job's turn, all cores score nodes.

**Upside**

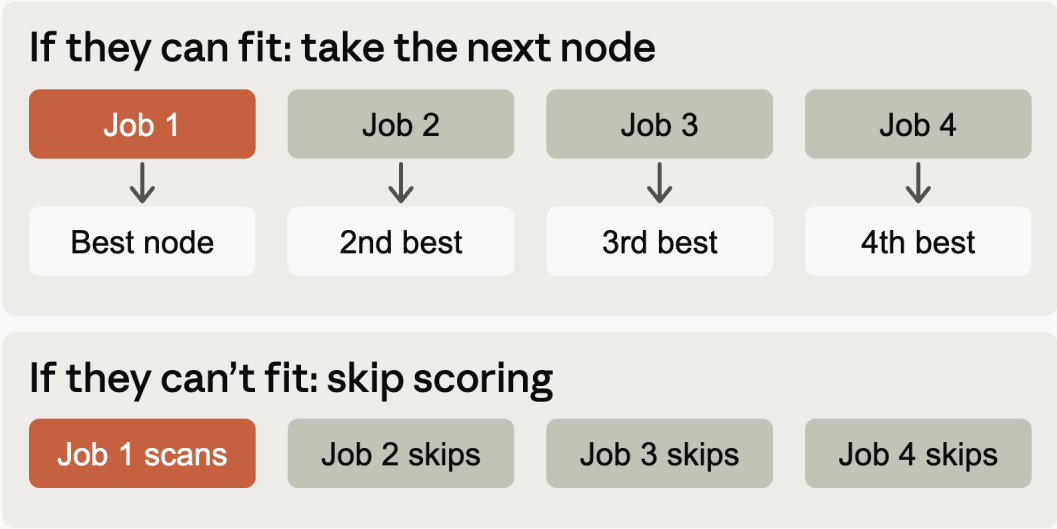
Node scoring is the bottleneck, so more cores speed up the whole pass.

**Downside**

We tried caching scores by node shape, but cache conflicts and lock overhead outweighed the win.

Making it fast

# One scan for same-shape jobs




Many pending jobs have the same spec. The first one scans the cluster, and the rest reuse what it found.

**Upside**

A long queue of same-spec jobs costs about the same as one job.

**Challenge**

Pod anti-affinity is hard to handle.

Making it fast

# Replay unchanged plans

Each pass, for a waiting job

| | |
|---|---|
| **Job and target nodes match** | Replay the plan |
| Anything differs | Score again |
| The plan is too old | Score again, maybe move |

A job can wait for preemptions for hours. We fingerprint the job and its nominated nodes, and reuse the plan as long as they match.

**Upside**

Waiting jobs don't cost a full scan every pass, and their plans don't jump around.

**Downside**

The NNN could become suboptimal over time.

Making it fast

# Protect the API server

**Pod status**
We write it once

**Pod events**
We don't write them

**Nominated node**
Write only on a real change, and wait for the plan to settle

We only write the changes that actually matter.

**Upside**

Waiting jobs add almost no writes, no matter how long they wait.

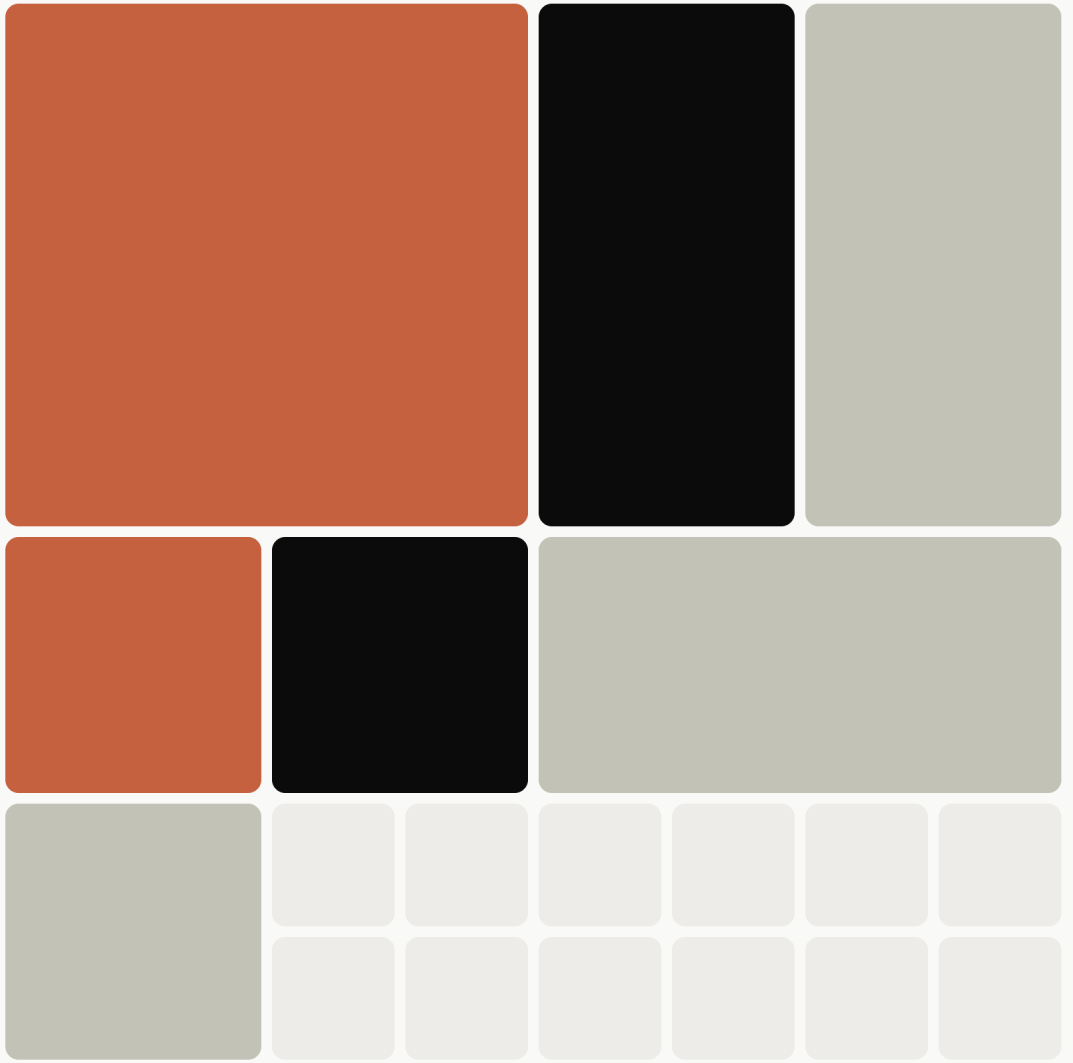
**Downside**

A pending pod's status message can be out of date, and its nominated node can lag behind the plan.

Using every chip

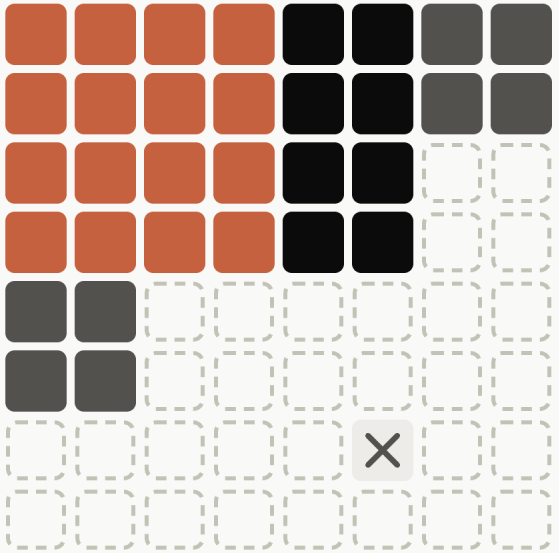
# Slicing and large jobs

How we share hardware between jobs, and how we place large ones.



Using every chip

# TPU soft slicing



**One 16x16 slice**

- 8x8
- 4x8
- 4x4

Each square is one VM of 4 chips. Dashed squares are free. X marks a failed VM.

One failed node leaves a hole in a slice, and a network domain only has a few slices. So we keep using the slice, and place smaller jobs in aligned tiles around the hole.

**Upside**

A failure costs us a small tile instead of a whole slice, and we can change job sizes without reprovisioning.

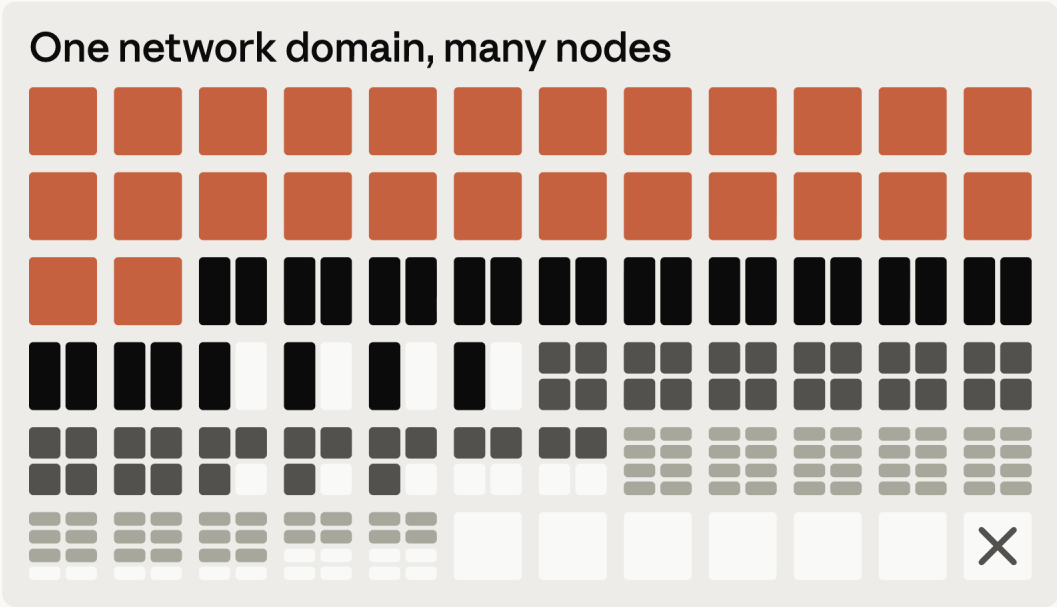
**Downside**

Smaller tiles have no wraparound links, and free tiles can end up scattered.

Using every chip

# One pod size per Trainium node

When a node fails it's gone whole, so there's no hole to work around. A network domain has many nodes, so limiting each to one pod size doesn't waste much.




Each small square is a 4x4 node. Pale pods are free.

**Upside**

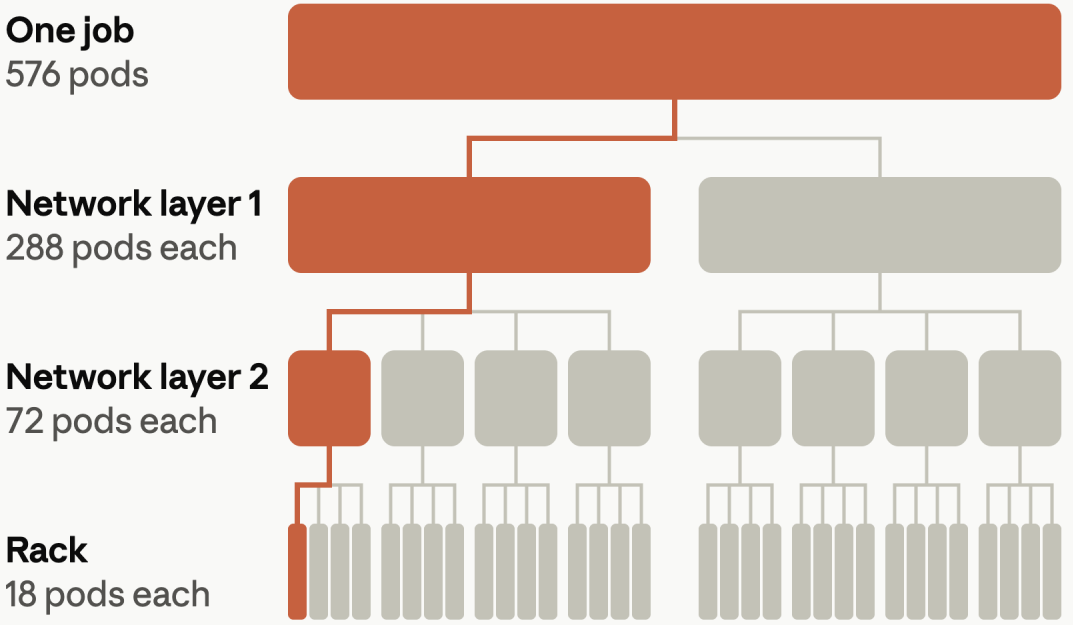
Picking chips and preempting are simple, because every pod on a node is the same size.

**Downside**

Capacity can get stranded: free chips split across small pods can't fit a larger one.

Using every chip

# Hierarchical placement




Made-up numbers. The orange path follows one rack's 18 pods.

A large job gets split down the network's layers, so the pods that talk the most end up close together.

**Upside**

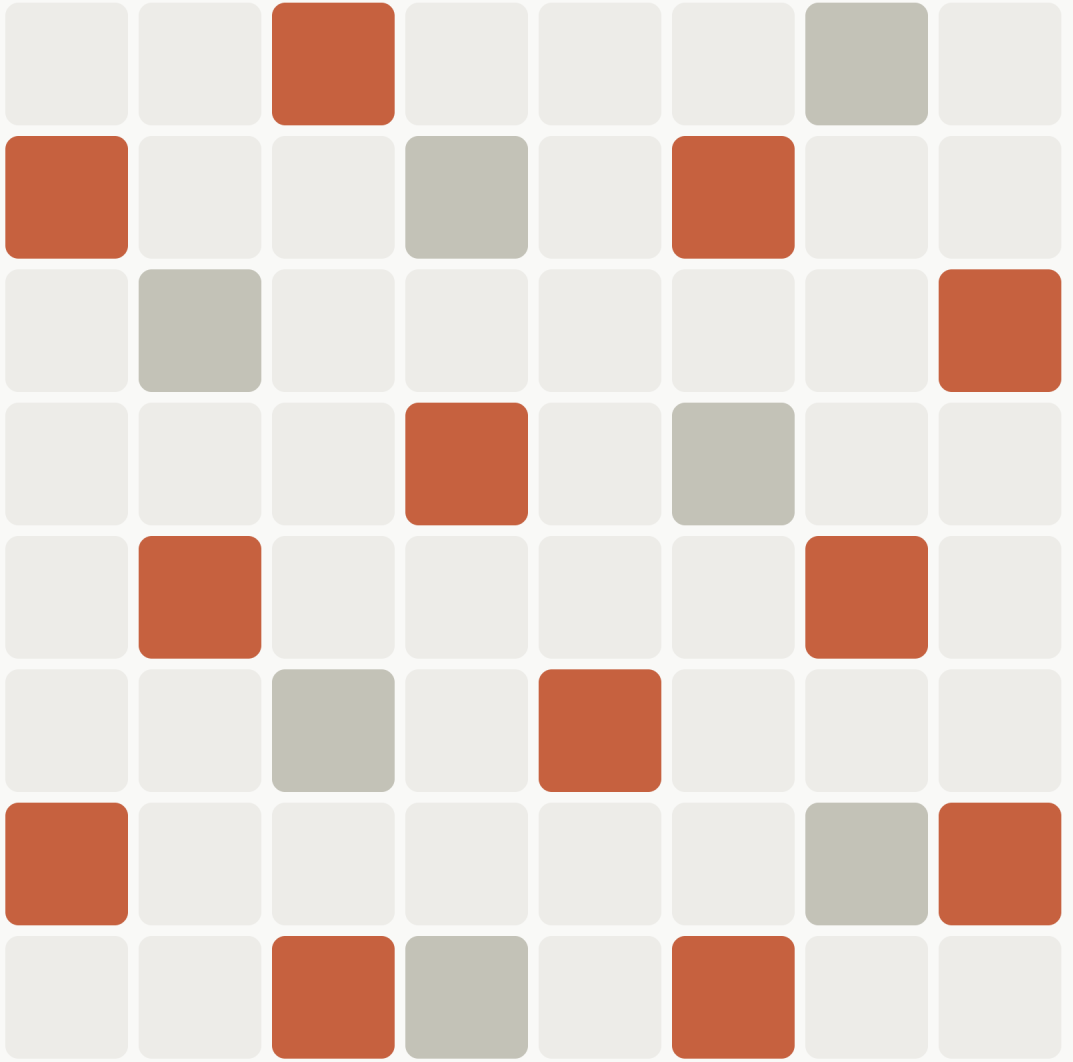
Most traffic stays low in the tree, and a job can grow without moving running pods.

**Downside**

This is best effort. On a full or unhealthy cluster, a branch can still split.

Using every chip

# Fragmentation

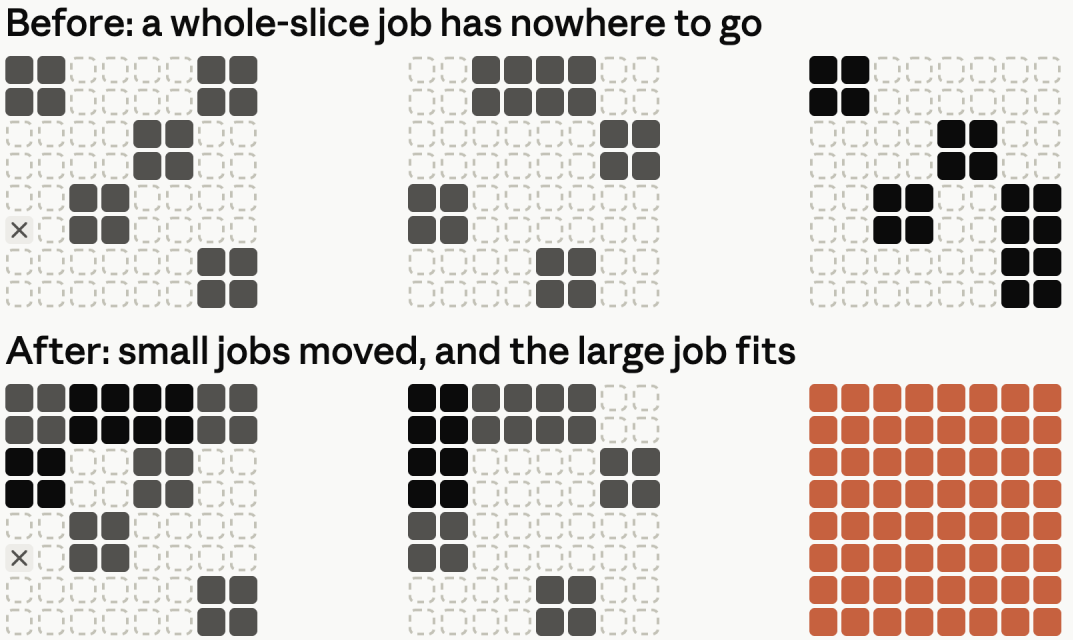


Free TPU chips get scattered over time, which blocks large jobs.

Using every chip

# Keep defragmenting so large jobs fit

Job churn and broken nodes keep scattering free chips across TPU slices, so we keep moving small jobs together to free up whole slices.




Three TPU slices. Black jobs move. X means failed, and orange is the large job.

**Upside**

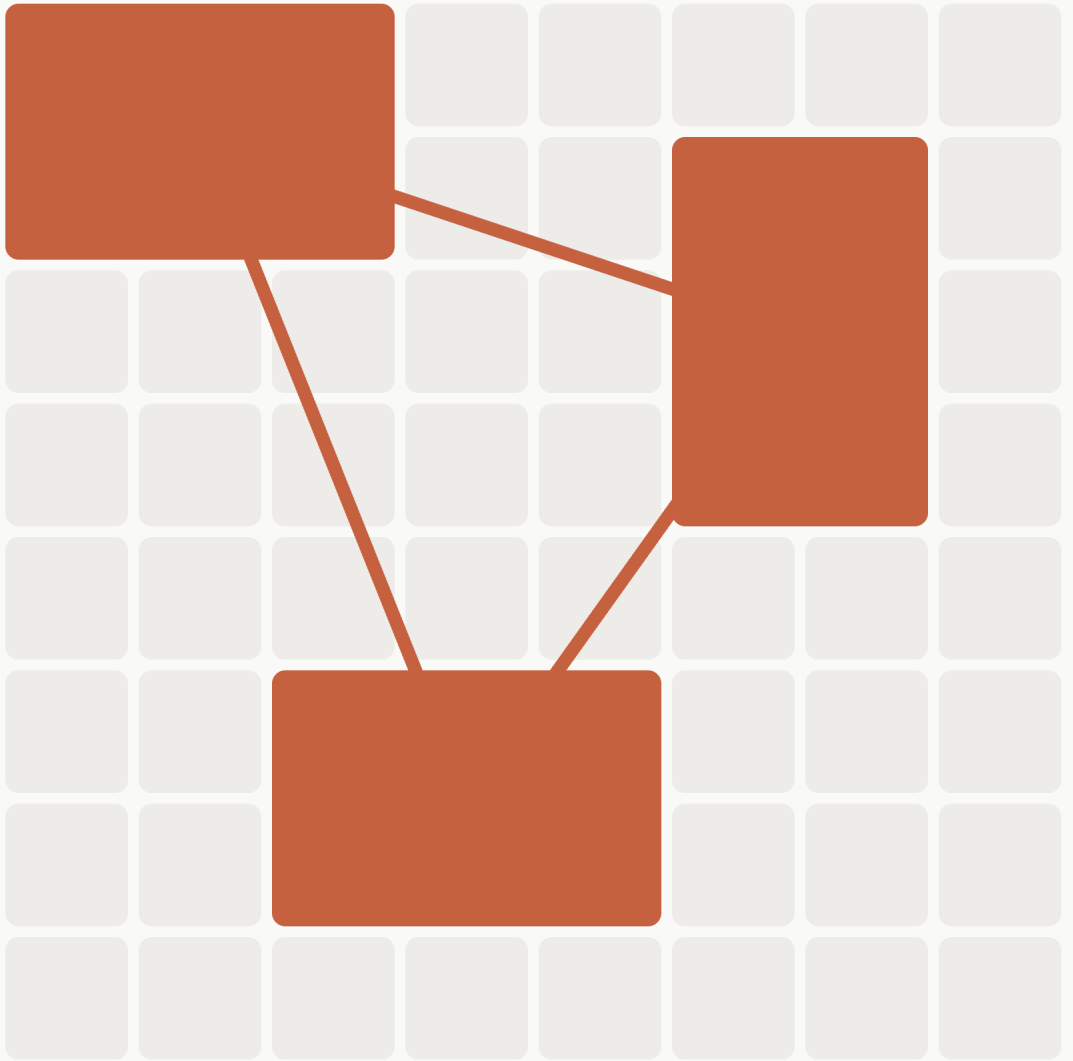
Large jobs don't have to wait for small ones to finish.

**Downside**

Moving a job costs it some work, so we limit how often and how much we move.

Placement rules

# JobGroup

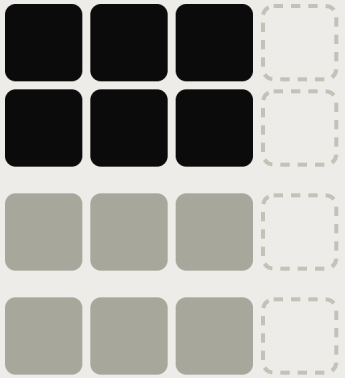


Some jobs only work together, so we place them as a group.

Placement rules

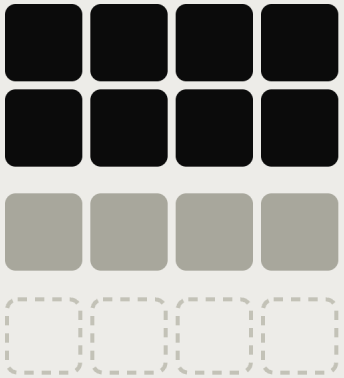
# Group jobs, not pods

**Pod group**
Counts pods



12 pods can be running with no job complete.

**Job group**
Counts whole jobs



The same 12 pods: a trainer and one sampler.

Black: a trainer. Gray: two samplers. Dashed: not placed.

Our scheduler works in jobs, so a JobGroup is just a set of jobs, placed all or none. A pod group would add a second unit.

**Upside**

Each job is a minimum scheduling unit.

**Downside**

It's our own CRD.

Upstream

# What's new upstream

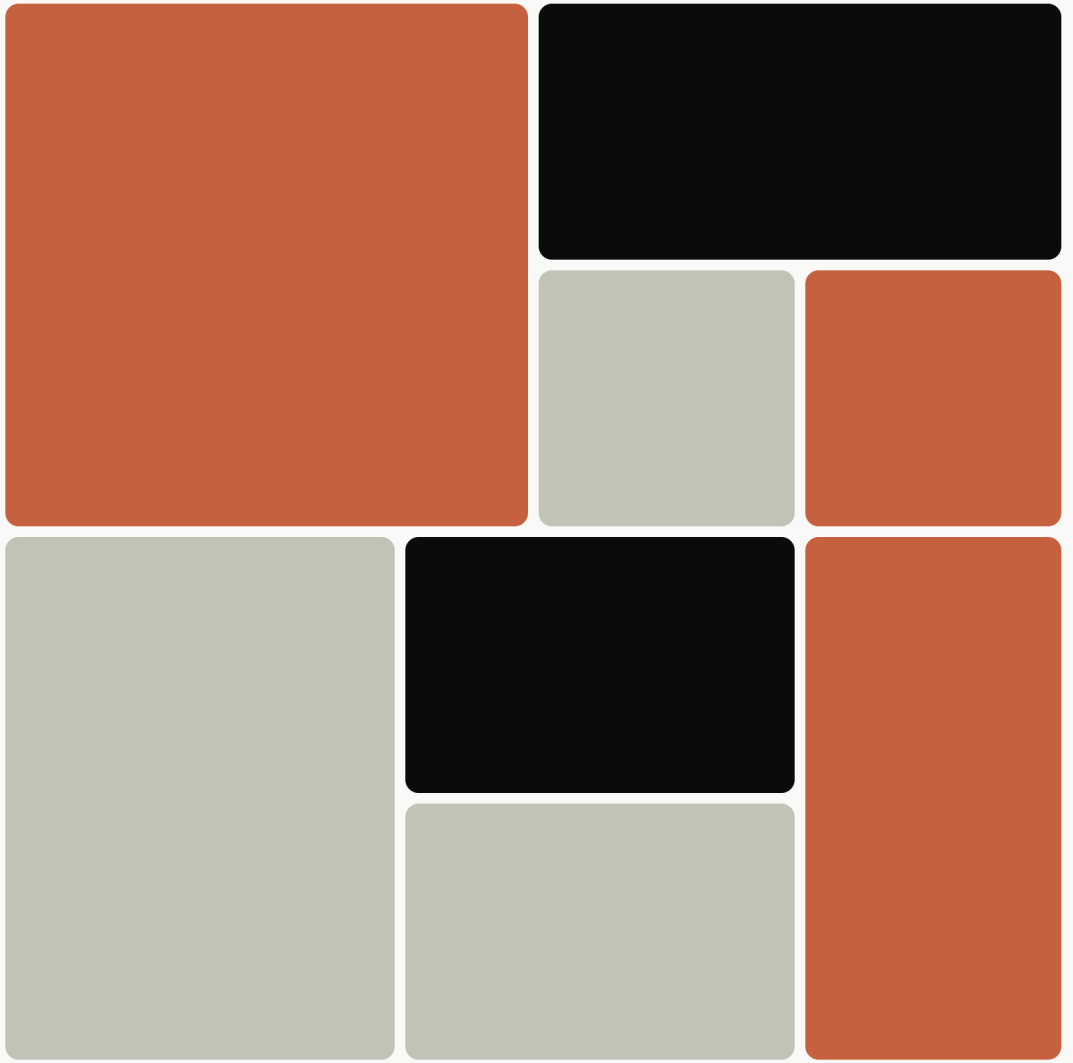
| Upstream | Stage | Closest in this talk |
|---|---|---|
| Gang scheduling and the PodGroup API | Beta | Decide for the whole job |
| Workload-aware preemption | Beta | Score preemption with binpacking |
| Multi-level topology-aware scheduling | Alpha | Hierarchical placement |
| CompositePodGroup | Alpha | JobGroup |
| DRA claims for PodGroups | Beta | We wish DRA were simpler |

We started before Kubernetes had any of this. As of v1.37 a lot of it is there, though it's off by default. Kueue integration is planned too.

Source: Kubernetes blog, "Kubernetes v1.37: Advancing Workload-Aware Scheduling"

Wishlist

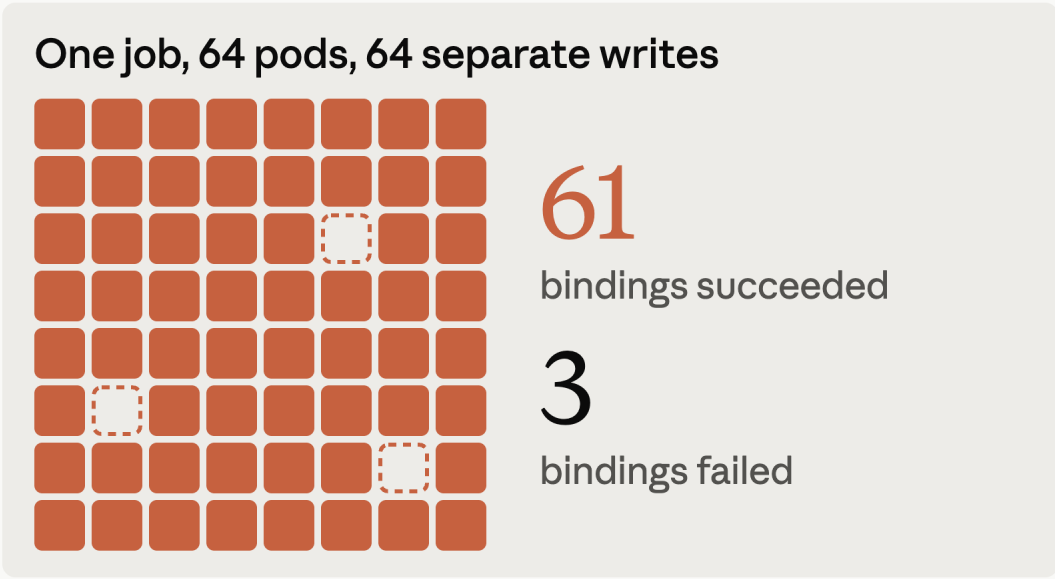
# Our Kubernetes wishlist



What we'd like to see in upstream Kubernetes.

Wishlist

# We wish the API had transactions




For example: the dashed pods aren't bound, so the other 61 hold their chips and the job can't run.

Binding is one write per pod, and nothing ties the writes together. Some can fail while the rest succeed.

**What we do today**

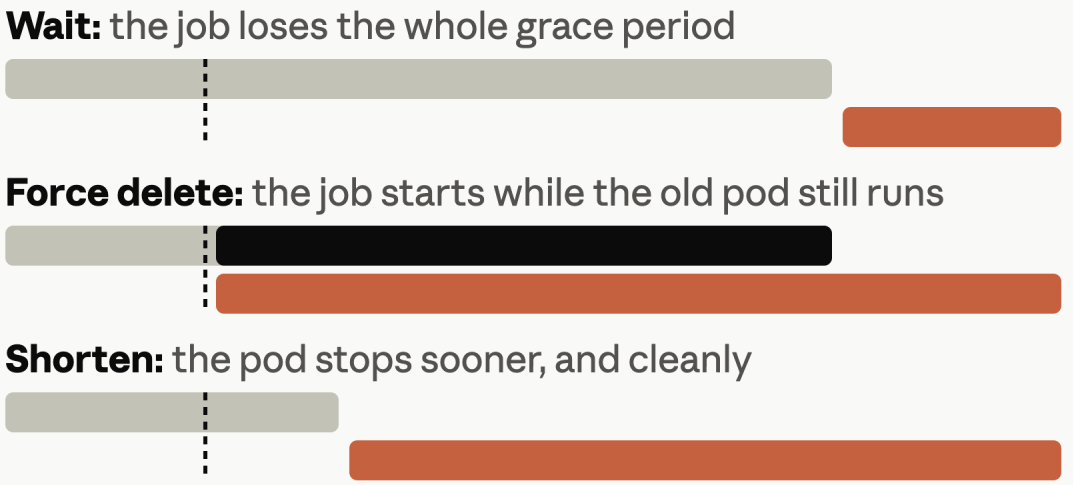
Retry each write, and replan any failed bind from live state.

**What we'd want**

Bind all 64 pods in one write, or none. We could drop a lot of repair logic.

Wishlist

# We wish we could shorten a drain




Time runs left to right. Gray: old pod. Orange: waiting job. Black: both at once. Dashed: the job needs the room.

A second delete can move deletionTimestamp earlier, but the kubelet sets its shutdown clock once and never updates it.

**What we do today**

Wait out the grace period, or plan the job somewhere else.

**What we'd want**

A shorter grace period that the kubelet respects, so the pod stops before its chips get reused.

Wishlist

# We wish DRA were simpler

| | Typical resource | DRA |
|---|---|---|
| The node offers | A number | A list of devices with attributes |
| The pod asks for | A count | A claim, with filters written in CEL |
| API objects | Pod and Node | Plus four DRA object kinds |
| The scheduler | Checks that it fits | Filters and picks devices |
| It writes | The binding | The allocation, then the binding |

A typical resource is a number on a node, and the scheduler checks that it fits. With Dynamic Resource Allocation, every scheduler has to become a device allocator.

**Upside**

You can describe devices in detail, share them and select them precisely.

**What we'd want**

A small profile for the common case that any scheduler can fully implement.

Wrapping up

# Thanks! Questions?

One bonus

## We can change anything we need

For example, we added many per-job features and gate them behind labels. We also make up new resources and new placement rules.

Decheng Dai, Anthropic