PLATFORM ENGINEERING · FINOPS · EKS

Karpenter vs Cast.ai on EKS: *I Ran Both in Production and Here's What I Found*

A technical evaluation of two node autoscalers across provisioning speed, driver protection, operational burden, and real cost — with a live 90-day benchmark.

Chinmaya Kumar Mishra Principal Platform Engineer · EKS Architect 18 min read · March 2026

> *I ran both tools simultaneously against the same workload in a production AWS EKS environment, dug through both teams' official documentation, and came out the other side with a clear — and perhaps counterintuitive — opinion.*

Why This Comparison Needed to Exist

Karpenter and Cast.ai look like they do the same job. Both watch for pods that can't be scheduled, spin up EC2 nodes to fit them, and clean up nodes when they're no longer needed.

The reason for this comparison was curiosity about Cast.ai's cost optimisation claims — and whether its broader feature set (live pricing intelligence, VPA rightsizing, continuous bin-packing) justified switching away from a tool that was already working well.

The comparison took longer than expected, and not just because of the technical depth involved. There is no single document that puts these two tools side by side the way an engineer actually needs. The official documentation for each tool is good within its own world, but crossing between them required extensive cross-referencing and hands-on testing.

These tools are solving the same problem from completely opposite architectural positions. Those positions have measurable consequences for how fast your jobs start, how much your compute bill is, and whether your Spark driver survives a consolidation cycle.

§

The Fundamental Difference Nobody Explains Clearly

The one question that unlocks everything else: **who makes the provisioning decision, and where?**

Karpenter lives entirely inside your cluster. When a pod goes Pending, its controller — a pod running in the `karpenter` namespace — reacts within milliseconds. It already has NodePool constraints loaded in memory, cached pricing data, and fires an `ec2:CreateFleet` call to AWS using credentials held via IRSA. No external call, no waiting for a backend, no sync delay. The entire path from "pod is stuck" to "EC2 API called" happens inside your cluster in roughly two seconds.

Cast.ai's intelligence lives on Cast.ai's servers. A lightweight agent called `castai-agent` runs inside your cluster and ships all cluster state changes to Cast.ai's cloud platform every 15 seconds. The actual decision — which instance type, which AZ, what price point — is made on Cast.ai's backend using ML models trained across their entire customer fleet and live AWS pricing APIs. Once the decision is made, Cast.ai's platform calls `ec2:RunInstances` directly in your AWS account via a cross-account IAM role.

That 15-second sync interval is the trade-off Cast.ai makes to access better intelligence. A pending pod might wait up to 15 seconds before Cast.ai even starts thinking about provisioning. In exchange, every provisioning decision has access to live pricing data and a bin-packing engine that a local rule-based system simply cannot replicate.

	Karpenter	Cast.ai
Decision Location	Inside your cluster	Cast.ai's cloud platform

Intelligence	Rule-based NodePool constraints	ML models + live AWS pricing
Pricing Data	Cached, refreshed periodically	Queried live at decision time
Internet Required?	No — fully in-cluster	Yes — outbound HTTPS (PrivateLink available)
Multi-cluster Learning	No — per cluster only	Yes — learns across all customers
Deployment Model	Open source, self-managed	Commercial SaaS

§

What's Actually Running in Your Cluster

KARPENTER'S FIVE MOVING PARTS

Component	What It Does
Controller Pod	Sits in the `karpenter` namespace watching the Kubernetes API. The brain of the operation.
NodePool	Where you declare your constraints — instance families, AZs, Spot vs On-Demand, labels.
EC2NodeClass	The AWS-specific half — AMI selection, subnets, security groups, user data.
NodeClaim	A Kubernetes object representing "I need a node with these properties." Created automatically.
IRSA	How Karpenter authenticates with AWS — short-lived credentials tied to a Kubernetes service account.

CAST.AI'S ARCHITECTURE

Component	What It Does
`castai-agent`	Ships cluster state to Cast.ai's platform every 15 seconds. The eyes of the operation.
`castai-cluster-controller`	Receives instructions from Cast.ai's platform and executes them inside the cluster.
`castai-aws-node`	Handles node bootstrap, labelling, and cleanup. Does *not* call EC2 — that's Cast.ai's platform.

`castai-evictor`	Continuously looks for underutilised nodes and evicts pods to consolidate. Runs all the time, not on a schedule.
`castai-workload-autoscaler`	Watches actual CPU and memory usage per pod and recommends rightsizing. The VPA engine.
`castai-pod-pinner`	Binds a pending pod directly to a newly Ready node — bypassing the scheduler to prevent misplacement.
`castai-spot-handler`	Listens for AWS Spot interruption notices and handles graceful draining.
Cast.ai SaaS	The external platform that does the actual thinking: bin-packing, pricing, ML models.

§

Authentication: Where the Models Differ Most Sharply

KARPENTER: STANDARD IRSA

Karpenter authenticates the way every well-behaved EKS workload should — using IAM Roles for Service Accounts. A service account annotation tells Kubernetes which IAM role to assume, Kubernetes issues a short-lived OIDC JWT, that token is exchanged with AWS STS for temporary credentials, and Karpenter uses those credentials to call EC2 APIs. Credentials expire hourly, rotate automatically, and nothing sensitive is ever stored anywhere.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: karpenter
  namespace: karpenter
  annotations:
    eks.amazonaws.com/role-arn: arn:aws:iam::ACCOUNT_ID:role/KarpenterControl
```

The IAM policy Karpenter needs is minimal. The most sensitive permissions — `ec2:CreateFleet`, `ec2:RunInstances`, `ec2:TerminateInstances` — are scoped to resources tagged with your cluster name. Nothing leaves your AWS account.

CAST.AI: CROSS-ACCOUNT IAM WITH SAAS BACKEND

Cast.ai's SaaS platform — running in Cast.ai's own AWS account — calls `ec2:RunInstances` in **your** AWS account using a cross-account IAM role you create during onboarding. This is a standard AWS pattern, but it means you are granting an external party the ability to launch EC2 instances in your account.

For teams in regulated environments, this cross-account trust relationship needs explicit security review. Cast.ai offers AWS PrivateLink connectivity which keeps all traffic off the public internet, and the cross-account role can be scoped with conditions to restrict which AZs and instance types Cast.ai is permitted to use.

§

Provisioning Speed: Karpenter's Structural Advantage

Speed matters most for interactive workloads and batch jobs with tight SLAs. For a Spark executor that needs to start within 30 seconds, the difference between two seconds and 17 seconds is the difference between meeting your SLA and missing it.

Karpenter: when a pod goes Pending, the controller is notified within milliseconds and fires `ec2:CreateFleet`. The time from pod Pending to EC2 API call is typically 1–3 seconds. The practical latency from "pod is stuck" to "pod is running" is dominated by EC2 boot time (~90s), not Karpenter's decision time. **Karpenter adds essentially zero overhead.**

Cast.ai: the `castai-agent` ships cluster state every 15 seconds. A pod that goes Pending one second after a sync cycle waits up to 14 seconds before Cast.ai's backend even sees it. Add network round-trip time and the practical overhead before EC2 receives the request is 15–20 seconds.

Metric	Karpenter	Cast.ai
Time to EC2 API call	**~2 seconds**	15–20 seconds
Decision location	In-cluster, in-memory	SaaS backend (network round-trip)

Warm node buffer	Not applicable	Configurable, adds cost
Practical pod start (cold)	EC2 boot time (~90s)	EC2 boot time + 15–20s overhead

§

Cost Optimisation: Cast.ai's Compute Advantage and Its Limits

Cast.ai's compute cost advantage comes from three mechanisms that operate simultaneously and compound each other. In my benchmark, they produced a **20.6% lower AWS compute bill**. Whether this translates to lower *total* cost depends entirely on Cast.ai's licence fee — covered in detail below.

MECHANISM 1: LIVE PRICING AT DECISION TIME

When Cast.ai's backend decides which instance to provision, it queries AWS pricing APIs live. It knows which Spot pools are currently cheap, which are likely to be interrupted, and which On-Demand instances offer the best price-performance at that exact moment. Karpenter uses cached pricing data — in volatile Spot markets, Cast.ai reacts to a price drop that Karpenter won't see for another refresh cycle.

MECHANISM 2: CONTINUOUS BIN-PACKING VIA THE EVICTOR

The `castai-evictor` runs continuously, not on a schedule. It constantly asks: "could I fit all the pods on this node onto other existing nodes?" If yes, it evicts the pods and terminates the node. The result is that Cast.ai's fleet shrinks over time as workload density increases — Karpenter's consolidation runs on a configurable disruption budget and isn't as aggressive.

MECHANISM 3: VPA RIGHTSIZING

The `castai-workload-autoscaler` watches actual CPU and memory usage per pod and adjusts resource requests to match. A pod requesting 4 CPU but consistently using 1.2 CPU gets rightsized to ~1.5 CPU with headroom. This compounds with bin-packing: smaller resource requests mean better bin-packing, which means fewer nodes, which means lower compute cost.

> IMPORTANT
>
> All three mechanisms reduce your AWS compute bill. They do not reduce Cast.ai's licence fee, which is charged separately on billable vCPU utilisation. For workloads with low sustained CPU, the compute saving can exceed the licence. For workloads with higher sustained utilisation — including continuous streaming jobs — the licence can exceed the compute saving. The cost analysis below covers this precisely.

§

Spark and EMR on EKS: The Specific Considerations

Most content about Karpenter and Cast.ai is written for microservice workloads. Spark and EMR on EKS have different characteristics that change the calculus significantly.

THE DRIVER POD PROBLEM

In a Spark job, the driver pod *is* the job. If the driver is evicted, the entire job fails. Both tools have protection mechanisms, but they work differently.

Karpenter uses the standard Kubernetes annotation:

```
karpenter.sh/do-not-disrupt: "true"
```

This prevents Karpenter's consolidation from evicting the annotated pod. It does not protect against VPA rightsizing, because Karpenter doesn't do VPA.

Cast.ai requires two separate annotations — one for eviction protection and one for VPA protection:

```
annotations:
  # Prevents Cast.ai Evictor from consolidating the node the driver is on
  autoscaling.cast.ai/removal-disabled: "true"

  # Disables Cast.ai vertical rightsizing for the driver only
  workloads.cast.ai/configuration: |
```

```
    vertical:
      optimization: off
```

Missing either annotation on the **driver** exposes it to that specific risk. Executor pods do not need these annotations — VPA-driven eviction of an executor is handled natively by Spark's task retry mechanism and is actually desirable.

24/7 STREAMING WORKLOADS

Karpenter has an `expireAfter` TTL on NodePool that forces node replacement after a configurable period. For a streaming job, this means the node will eventually be drained and replaced — potentially disrupting the driver unless your Spark configuration handles reconnection.

Cast.ai has no built-in node TTL. A node running a protected streaming job will not be evicted by the Evictor, and Cast.ai will not force replacement on a schedule. The node runs until you tell it to stop, or until AWS terminates it. **For 24/7 streaming workloads, this is a meaningful operational difference.**

§

The Complete Head-to-Head Scorecard

Dimension	Karpenter	Cast.ai
Provisioning Speed	★★★★★ **Fastest**	★★★☆☆
Compute Savings	★★★☆☆	★★★★☆ **20–35%***
Optimisation Intelligence	Rule-based NodePool	**ML + live AWS pricing**
VPA (Vertical Autoscaling)	✗	✓ **Built-in, per pod**
Spot Interruption Response	★★★★★ **SQS + EventBridge; actively drains + provisions replacement**	★★★☆☆ Monitors; Kubernetes handles termination
Spot Interruption	★★☆☆☆ No predictive capability	★★★★★ **ML-based proactive rebalancing**

Prevention		
Driver Pod Protection	✓ Eviction only	**✓ Eviction + VPA**
24/7 Streaming Safety	✓ Until expireAfter TTL	**✓ Indefinite — no ceiling**
Maintenance Burden	High — team owns everything	**Low — mostly automated**
Operational UI	None	**Full SaaS console**
Licence Cost	**Free (open source)**	€8/billable vCPU/month

* Total cost saving is workload-dependent; see cost analysis below.

§

Cost Analysis: A Live Node Benchmark

I ran both autoscalers simultaneously against the same workload: a **24/7 location streaming job** running continuously without interruption. Same job, same data throughput, same load profile. All cost figures are based on actual AWS instance pricing for the nodes each autoscaler provisioned.

CAST.AI FLEET — 5 NODES

Instance	Pricing	$/h	$/day	vCPU	CPU util	Mem util
c6g.16xlarge	Spot	$0.490	$11.76	64	69%	95%
m6g.12xlarge	On-Demand	$1.548	$37.15	48	84%	57%
inf1.2xlarge	Spot	$0.109	$2.62	8	71%	30%
m6g.2xlarge	On-Demand	$0.258	$6.19	8	52%	21%
c6g.large	Spot	$0.026	$0.62	2	96%	67%
Total	**60% Spot**	**$2.431**	**$58.34**	**130**	**74.4% avg**	**54% avg**

KARPENTER FLEET — 6 NODES

Instance	Pricing	$/h	$/day	vCPU	CPU util	Mem util
c6g.8xlarge	Spot	$0.392	$9.41	32	51%	78%
c6g.8xlarge	Spot	$0.392	$9.41	32	63%	88%
c6g.8xlarge	On-Demand	$0.876	$21.02	32	76%	**99% ⚑**
c6g.8xlarge	On-Demand	$0.876	$21.02	32	76%	**99% ⚑**
i4g.2xlarge	Spot	$0.141	$3.38	8	87%	14%
r6gd.2xlarge	On-Demand	$0.384	$9.22	8	50%	9%
Total	**30% Spot**	**$3.061**	**$73.46**	**144**	**67.2% avg**	**64.5% avg**

> ⚑ MEMORY SATURATION RISK
>
> Two Karpenter c6g.8xlarge On-Demand nodes are running at 99% memory utilisation. This is an active OOM risk that Cast.ai's continuous rightsizing engine would detect and rebalance automatically. It exists independent of the cost comparison and is a genuine production reliability concern.

CAST.AI LICENCE PRICING

Cast.ai charges **€8 per billable vCPU per month**. The critical distinction: **billable vCPUs = average CPU actually utilised by workloads**, not total provisioned vCPUs.

For this 24/7 streaming job, CPU utilisation was measured at a consistent **5 vCPU/day** — a stable figure driven by the continuous, steady-load nature of the workload.

```
Daily licence cost   = 5 vCPU × €8 × 1.1462 (EUR/USD, Mar 2026) = $45.85/day
Monthly licence cost = 5 vCPU × €8 × 30 × 1.1462                 = $1,375/month
```

> COMMON MISREAD
>
> Multiplying €8 by provisioned vCPUs (130) gives €1,040/month — a figure **13× higher** than the actual charge. Cast.ai bills on utilisation, not provisioned capacity.

SCENARIO 1: COMPUTE COST ONLY

On raw AWS compute, Cast.ai wins at every time horizon — 20.6% lower than Karpenter.

Period	Cast.ai	Karpenter	Cast.ai saves
Per hour	$2.431	$3.061	$0.630
Per day	$58.34	$73.46	**$15.12 (20.6%)**
Per month	$1,750	$2,204	**$454**
Per year	$21,000	$26,448	**$5,448**

SCENARIO 2: TOTAL COST INCLUDING LICENCE

At 5 billable vCPU/day, the licence costs **$45.85/day — exactly 3.03× the compute saving of $15.12/day.** The licence absorbs the entire compute advantage and adds $30.73/day on top.

Line item	Cast.ai/day	Karpenter/day	Cast.ai/month	Karpenter/month
AWS compute	$58.34	$73.46	$1,750	$2,204
Cast.ai licence	$45.85	$0	$1,375	$0
Total cost	**$104.19**	**$73.46**	**$3,125**	**$2,204**
Karpenter saves	—	**$30.73/day**	—	**$921/month**

Annualised, the gap is **$11,052/year in Karpenter's favour.**

> SCOPE OF COST FIGURES
>
> These figures reflect a 24/7 continuously running streaming job consuming a steady 5 billable vCPU/day — the worst-case scenario for Cast.ai's licence economics. For batch jobs running a few hours per day, billable vCPU averages to a fraction of this figure, the licence cost drops accordingly, and Cast.ai's compute saving easily absorbs it. **Do not use these numbers to draw conclusions about batch workload economics.**

§

The Maintenance Reality: Life After Setup

Getting either tool running isn't the hard part. Living with it through EKS version upgrades, changing workload profiles, new AMI releases, and team growth is where the real operational cost reveals itself.

With Karpenter, your team owns quarterly Helm upgrades (EKS versions occasionally require minimum Karpenter versions — the v0.x to v1.x migration from `Provisioner` → `NodePool` and `AWSNodeTemplate` → `EC2NodeClass` required rewriting config and careful sequencing to avoid outages), manual AMI pin management (pin too long and you accumulate CVEs; use `@latest` and you risk unexpected node churn), NodePool tuning for every new workload type, and command-line-only debugging.

With Cast.ai, the SaaS model shifts the maintenance burden significantly. Bin-packing algorithms, pricing models, instance type catalogue, and Evictor logic are updated by Cast.ai's backend automatically. If you onboard via the Cast.ai Operator, even `castai-agent` manages itself. The main ongoing responsibility is annotation discipline — every new stateful workload type needs the two protection annotations applied. Lightweight, but it requires process.

Task	Karpenter	Cast.ai
Version upgrades	Manual — Helm + staging + CRD checks	**Mostly automatic via Operator**
AMI / OS rotation	Manual — pin, validate, roll out	**Automatic via scheduled rebalancing**
Debugging	kubectl logs + NodeClaim inspection	**Console — visual provisioning history**
Operational UI	None	**Full SaaS console**
Main incident vector	NodePool misconfiguration or failed upgrade	Missing annotation on a new stateful pod

§

Conclusion

If you're running Spark or EMR on EKS and evaluating whether to adopt Cast.ai: the answer is yes — with an important financial qualification.

Karpenter is an excellent node provisioner — fast, reliable, and

operationally transparent. But for Spark and EMR workloads, Cast.ai is the more complete platform. It neutralises the speed gap through node retention, provides broader driver protection by covering both eviction and rightsizing as independent concerns, delivers continuous cost optimisation through three compounding mechanisms Karpenter simply doesn't have, and protects 24/7 streaming workloads without any expiry ceiling.

However, Cast.ai's economics are workload-dependent. At 5 vCPU/day sustained — the measured result for a 24/7 streaming workload in this evaluation — Karpenter is **$921/month cheaper on a total cost basis**. This is not a theoretical edge case; it is a direct measurement.

USE CAST.AI FOR

Batch & Low-Utilisation Workloads

ETL pipelines, nightly aggregation, ML training runs, any Spark job that starts, processes, and terminates. Bursty workloads keep billable vCPU low, live Spot pricing picks the cheapest pool at each launch, and VPA rightsizing compounds savings across dozens of runs per week.

USE KARPENTER FOR

24/7 Streaming Workloads

Location streaming, event pipelines, real-time aggregation — where sustained CPU utilisation makes the Cast.ai licence expensive relative to the compute saving. Zero licence overhead, zero expiry ceiling risk, and $921/month cheaper for the workload measured in this evaluation.

§

Recommendation: Use Both — Cast.ai for Batch, Karpenter for Streaming

Run Cast.ai as the provisioner for batch workloads — ETL pipelines, nightly aggregation, ML training, any Spark job that starts, processes, and terminates. Run Karpenter for 24/7 streaming workloads where sustained CPU utilisation makes the Cast.ai licence uneconomical.

One critical operational detail in a dual-provisioner setup: **Cast.ai's** `castai-workload-autoscaler` **operates cluster-wide.** It watches CPU and memory usage across all pods regardless of which provisioner created

the underlying node. Cast.ai's VPA engine will observe and rightsize pods running on Karpenter nodes as readily as pods on Cast.ai nodes.

This is desirable for executor pods — let it run freely. VPA rightsizes by evicting the pod so it restarts with corrected resource requests; for executors, that's just a task retry, which Spark handles natively.

The driver pod is the critical case. Apply all three protections to every Spark driver, regardless of which provisioner placed its node:

```
annotations:
  # Prevents Karpenter from disrupting this pod during node consolidation
  # Source: https://karpenter.sh/docs/concepts/disruption/
  karpenter.sh/do-not-disrupt: "true"

  # Prevents Cast.ai Evictor from removing this pod for consolidation
  # Source: https://docs.cast.ai/docs/evictor
  autoscaling.cast.ai/removal-disabled: "true"

  # Disables Cast.ai vertical rightsizing for CPU and memory on the driver
  # Source: https://docs.cast.ai/docs/workload-autoscaler-annotations-referen
  workloads.cast.ai/configuration: |
    vertical:
      optimization: off
```

Leave executor pods unannotated so Cast.ai can rightsize them freely. This is the correct annotation strategy in a dual-provisioner cluster regardless of which provisioner placed the driver's node.

THE BOTTOM LINE

This dual-provisioner approach carries an operational caveat: two sets of configurations, two upgrade paths, two operational models to own. That overhead is real and should not be underestimated.

But the trade-off is worth making. **Batch jobs on Cast.ai** keep the licence cost low and let the automation handle rightsizing and Spot selection. **Streaming jobs on Karpenter** carry no licence overhead and keep long-running workloads stable. Every workload runs on the right tool for the job — and the overall compute bill is lower than any single-tool compromise.

References

Cast.ai Autoscaler documentation · Cast.ai Evictor documentation · Cast.ai Workload Autoscaler annotations reference · Karpenter concepts documentation

CK

Chinmaya Kumar Mishra

Principal Platform Engineer and EKS Architect with 10 years of infrastructure and cloud-native experience. Holds active CKA and AWS Solutions Architect – Associate certifications. Writes about platform engineering, FinOps, and DevSecOps.

linkedin.com/in/chinmaya-mishra-2ab99014