

Scaling Smarter Not Harder: How Extending Cluster Autoscaler Saves Millions

KubeCon + CloudNativeCon, London 2025



DATADOG



Rahul Rangith

Software Engineer, Datadog



Ben Hinthorne

Software Engineer, Datadog

Agenda

01 Autoscaling at Datadog

02 Cluster Autoscaler and Expanders

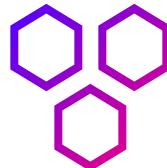
03 Identifying Optimal Instance Types

04 Scaling Optimal Instance Types

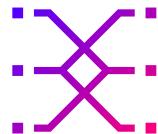
Datadog Infrastructure



Kubernetes from scratch in a multi cloud environment



Dozens of clusters
10,000s of nodes
100,000s of pods



Serving trillions of data points per hour for
over 30k customers

Compute - Autoscaling



Manage node infrastructure for product teams



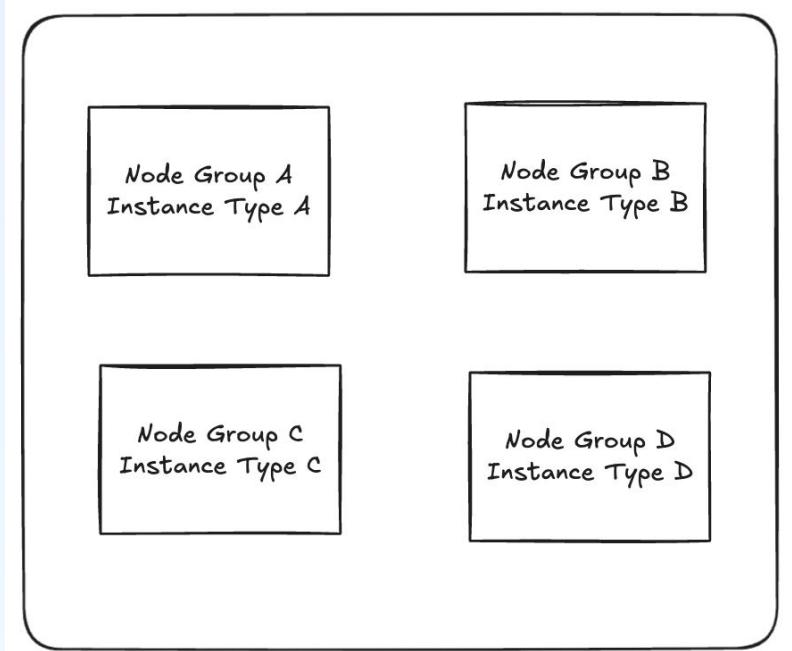
Scheduling and scaling efficiency



Bin packing and cost optimizations

Node Group Sets

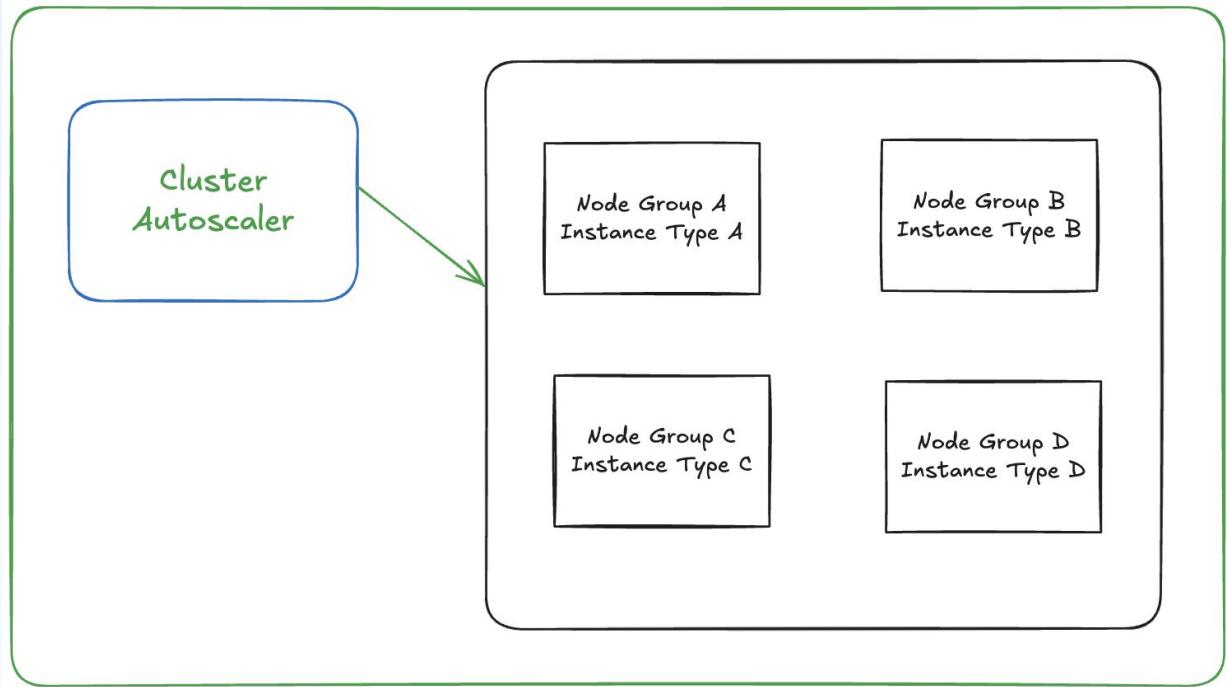
- A **Node Group** is a cloud provider agnostic representation for an ASG, MIG, etc.
- A **Node Group Set** is a set of node groups under the same scheduling domain.
 - Easier to onboard and manage many users
 - Easier to provide multiple instance type options



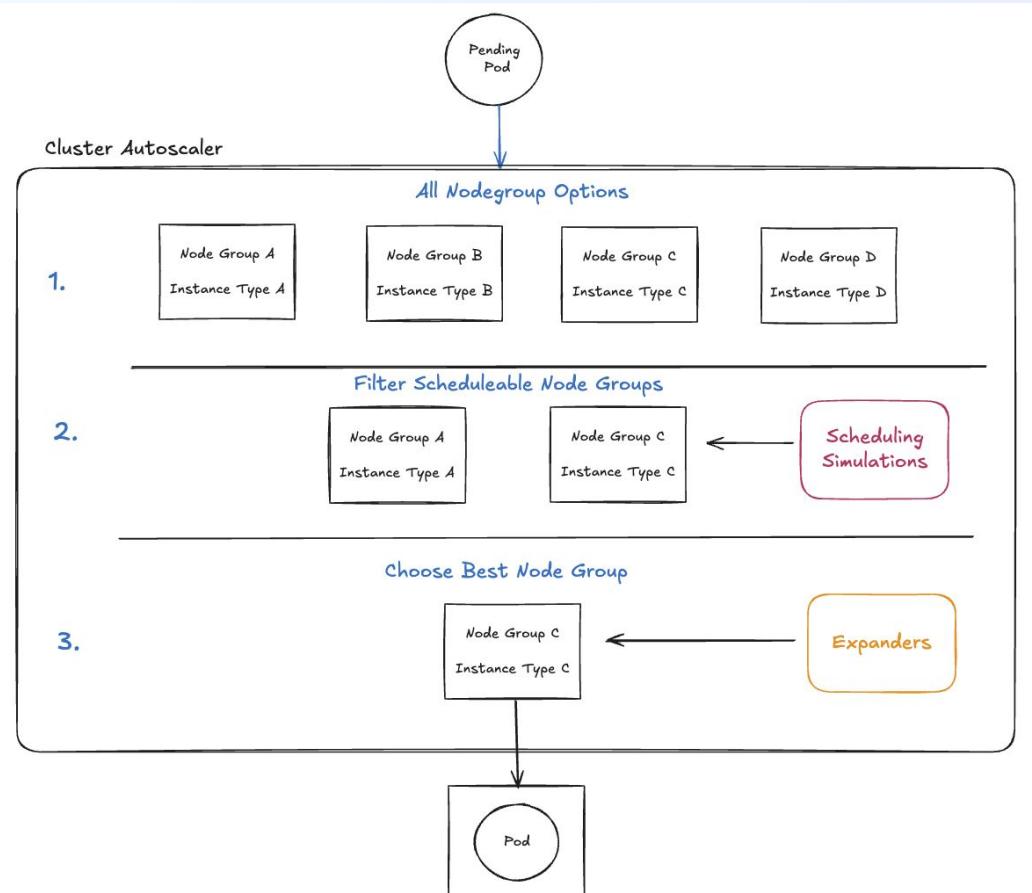
Cluster Autoscaler

We use **Cluster Autoscaler** for node autoscaling opposed to alternatives, primarily because of:

- Multi cloud support
- Happy with the operational experience



Cluster Autoscaler: Instance Type Selection



All Nodegroup Options

Filter Scheduleable Node Groups

Expanders pick the best option

Cluster Autoscaler: Built in Expanders

Random: Select a random node group

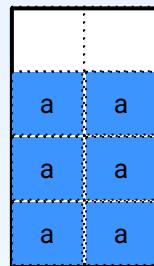
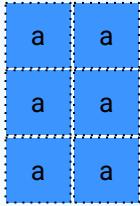
Least Waste: Select the node group that will waste the fewest resources

Price: Select the cheapest node group option

Priority: Select node groups based on user assigned priorities

Case Study: Least Waste Expander

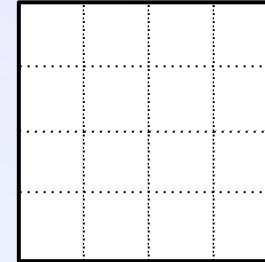
Pod Requesting 6 Cores



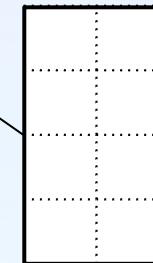
Least Waste

Node Group Set

16 Core Machine

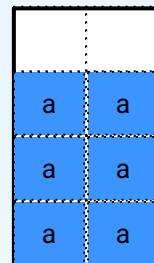
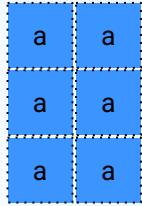


8 Core Machine



Case Study: Least Waste Expander

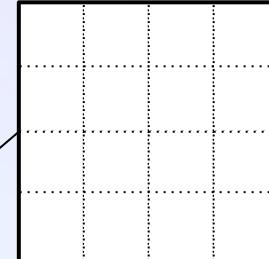
Pod Requesting 6 Cores



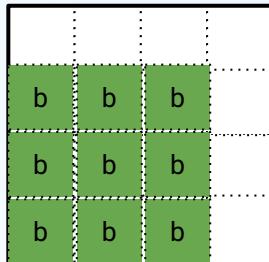
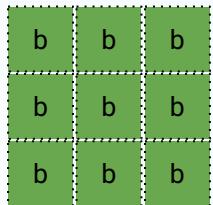
Least Waste

Node Group Set

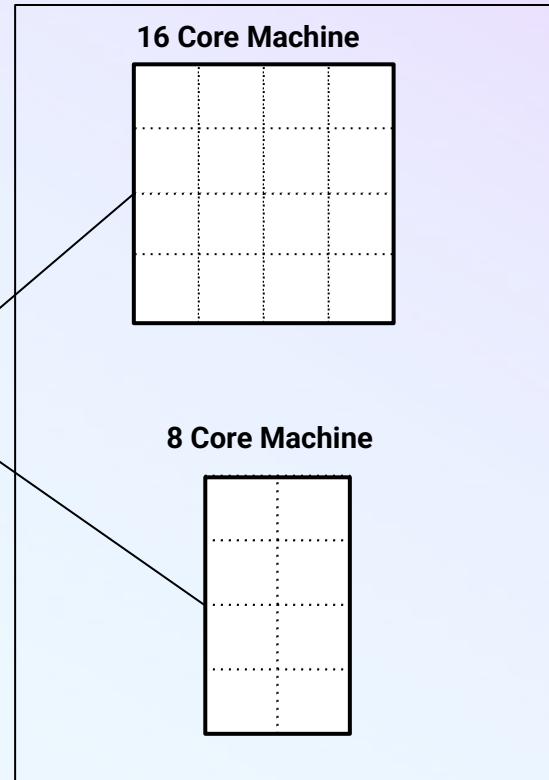
16 Core Machine



Pod Requesting 9 Cores

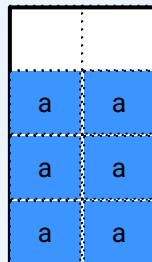
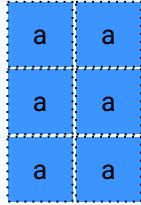


8 Core Machine



Case Study: Least Waste Expander

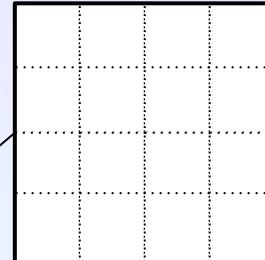
Pod Requesting 6 Cores



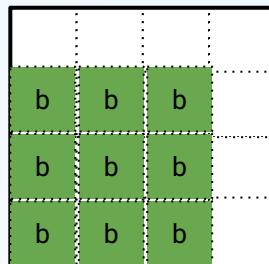
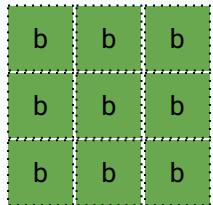
Least Waste

Node Group Set

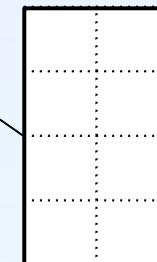
16 Core Machine



Pod Requesting 9 Cores

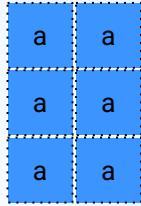


8 Core Machine

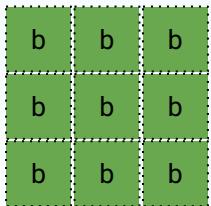


Case Study: Least Waste Expander

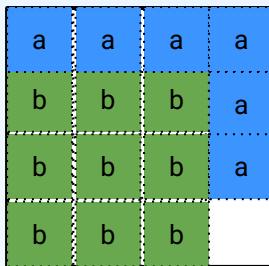
Pod Requesting 6 Cores



Pod Requesting 9 Cores

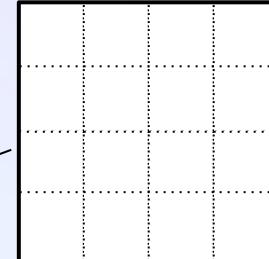


1 wasted core!!



Node Group Set

16 Core Machine



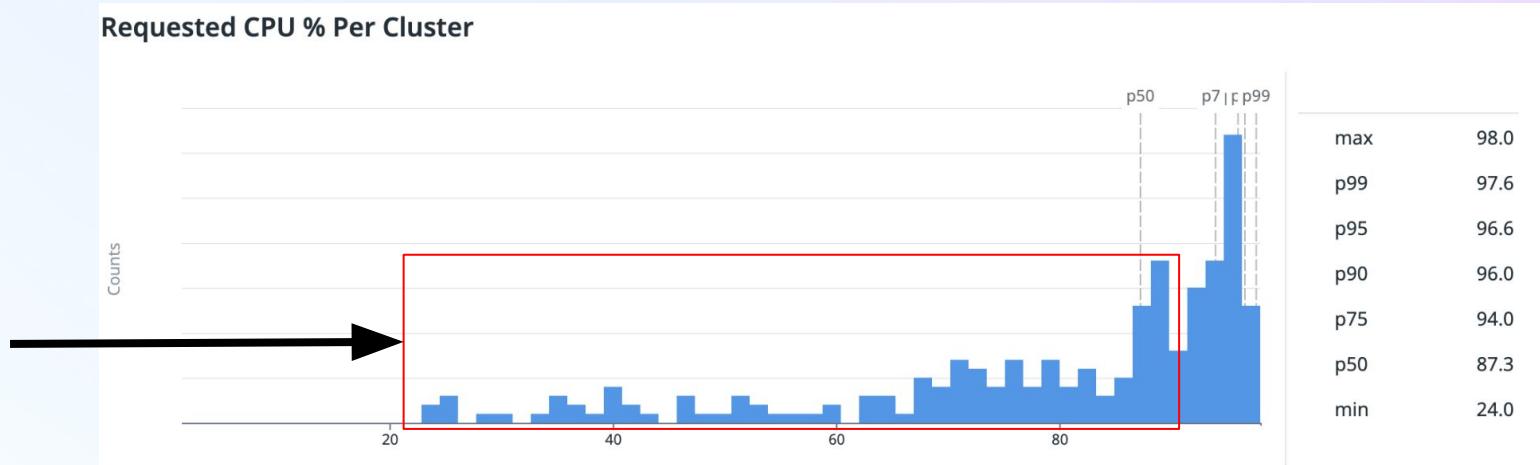
8 Core Machine



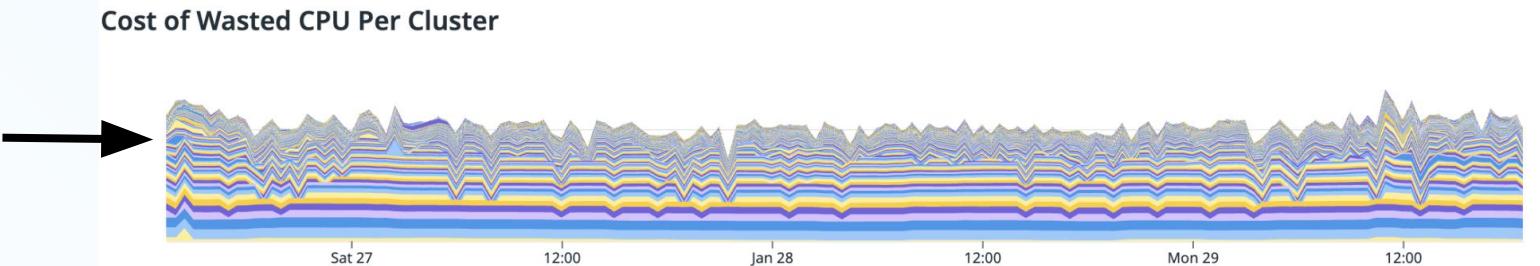
Is this theoretical optimization applicable?

In Practice: Least Waste Expander

Clear opportunity
for bin packing
improvements



Translates to
significant cost
impact



Instance Selection: Beyond Least Waste



Bin Packing



Instance
Performance



Capacity and
Homogeneity

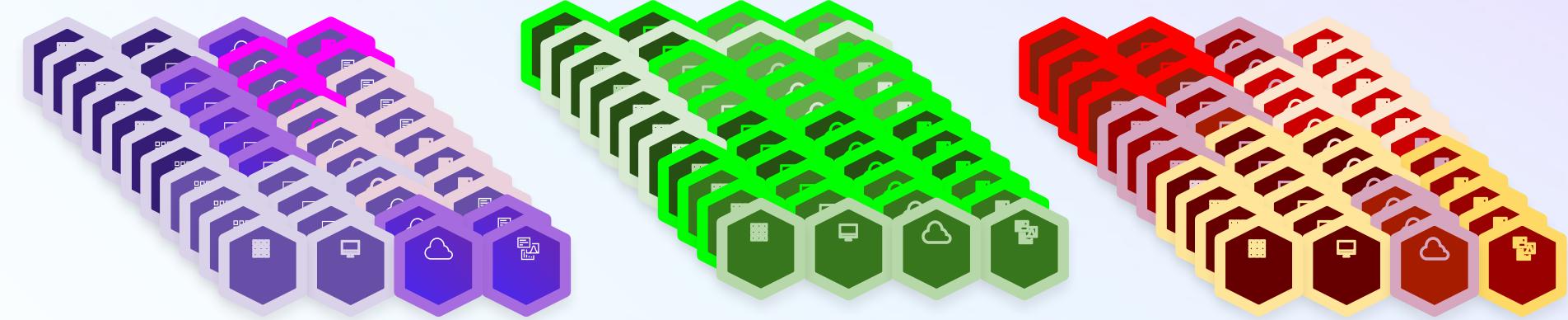


Cluster Specific
Preferences

Instance Selection: At Scale



Instance Selection: At Scale

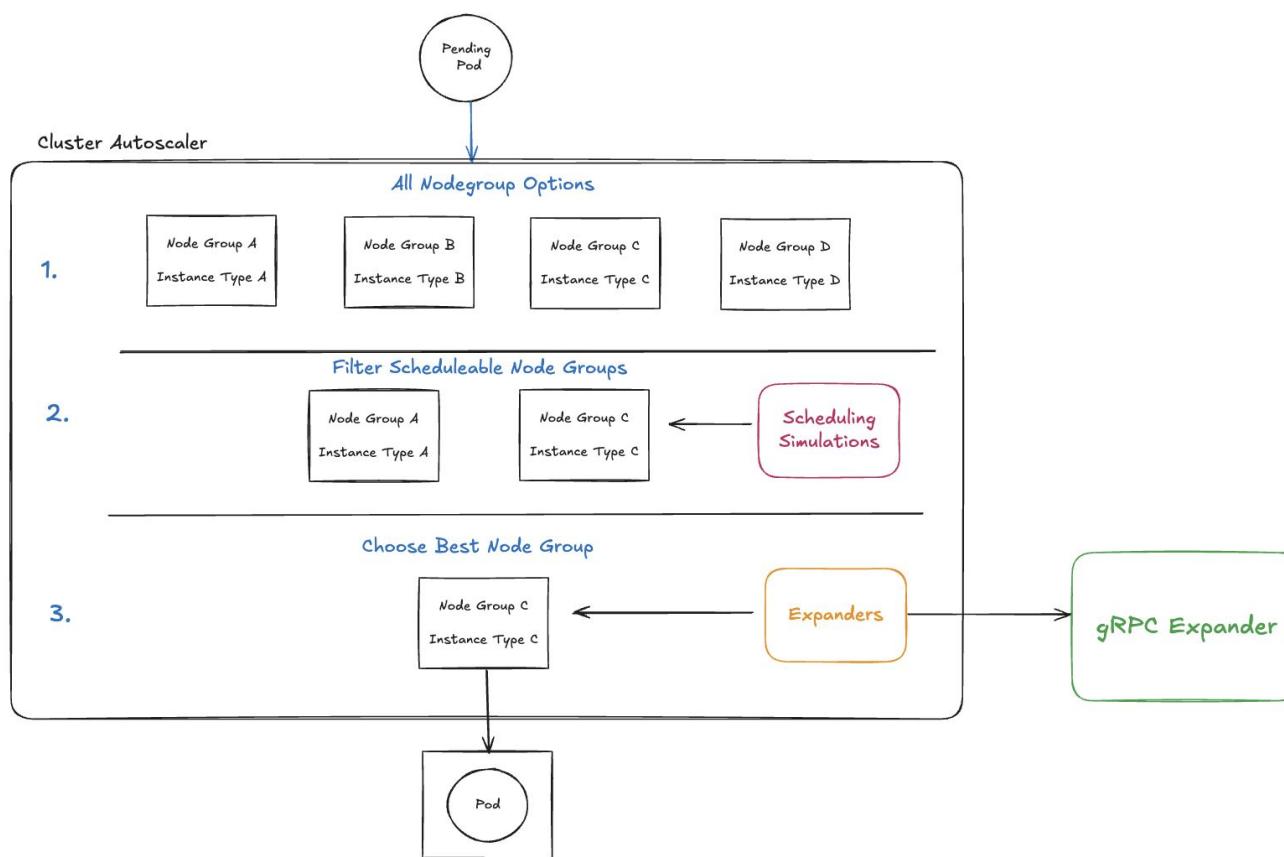


There is no “one expander fits all” solution.

There is no “one expander fits all” solution.

Unless...

Cluster Autoscaler: gRPC Expander



Cluster Autoscaler supports a **gRPC Expander!**

Allows users to build a service to make custom node group decisions!

gRPC Expander: Upstream Example

```
// BestOptions method filters out the best options of all options passed from the gRPC Client in CA, according to the defined strategy.
func (ServerImpl *ExpanderServerImpl) BestOptions(ctx context.Context, req *protos.BestOptionsRequest) (*protos.BestOptionsResponse, error) {
    opts := req.GetOptions()
    log.Printf("Received BestOption Request with %v options", len(opts))

    // This strategy simply chooses the Option with the longest NodeGroupID name, but can be replaced with any arbitrary logic
    longest := 0
    var choice *protos.Option
    for _, opt := range opts {
        log.Println(opt.NodeGroupId)
        if len(opt.NodeGroupId) > longest {
            choice = opt
        }
    }

    log.Print("returned bestOptions with option: ", choice.NodeGroupId)

    // Return just one option for now
    return &protos.BestOptionsResponse{
        Options: []*protos.Option{choice},
    }, nil
}
```

https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/expander/grpcplugin/example/fake_grpc_server.go

What we know so far

1. We have an opportunity to improve our fleet's efficiency via instance type selection.
2. The **gRPC Expander** gives us ultimate flexibility in instance type selection.

Our Goals

1. **Identify** the best instance types
2. **Scale** the best instance types

Instance Type Selection: Simplified Criteria



Cost



Performance



Reliability

Cost: Back to the Bin packing Problem

a	a
a	a
a	a

b	b	b
b	b	b
b	b	b

c
c
c

d	d
d	d

e	e	e	e	
e	e	e	e	
e	e	e	e	e

f	f
f	f

g	g	
g	g	
g	g	g

• • •

?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?

How can we
binpack 1000s of
unique workloads?

Cost: Scheduling Simulations

Instance Type Advisor

Set of Pods

- Selected via a NodeSelector, e.g. "get all pods running on nodes that have label foo=bar"

Instance Catalog

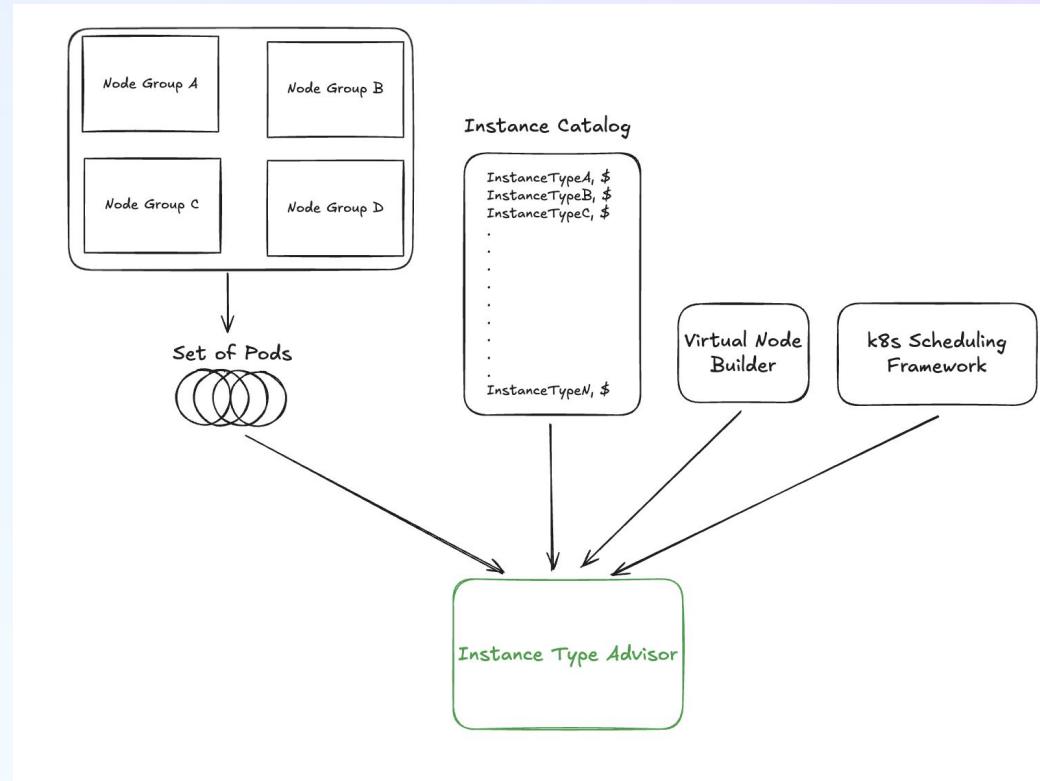
- Catalog of all possible instance types and their specs, including cost per hour, cpu/mem capacity, etc.

Virtual Node Builder

- Interface to build virtual nodes with desired configuration

Scheduling Framework

- Upstream k8s scheduling framework, or other scheduler plugins possible



Cost: Scheduling Simulations

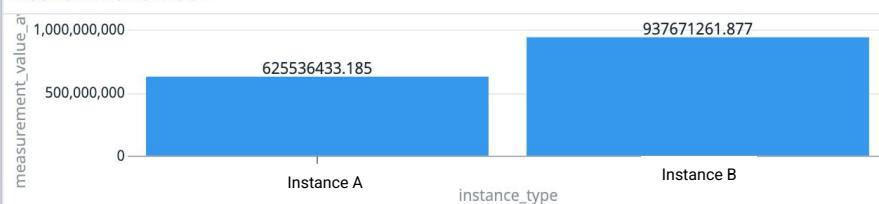
Instance Type	Hourly Cost	Number of Nodes	Requested CPU %	Requested Memory %
m6a.8xlarge (current)	5555.55	152	62	76
r6a.4xlarge (best)	4444.44	160	95	62
c6a.8xlarge	6655.55	180	54	85

The **Instance Type Advisor** results allows us to compare the cost and bin packing efficiency for each instance type for our specific set of pods.

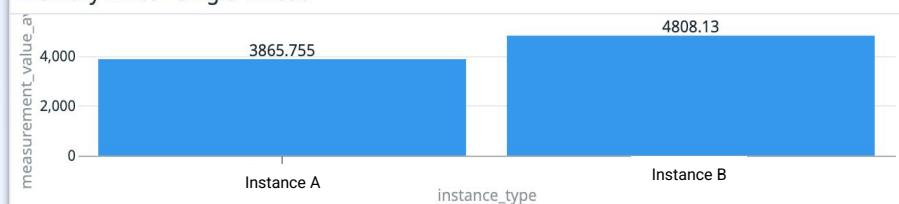
We run automated scheduling simulations in every cluster, and store the results in a CRD in the cluster itself.

Performance: Benchmarks

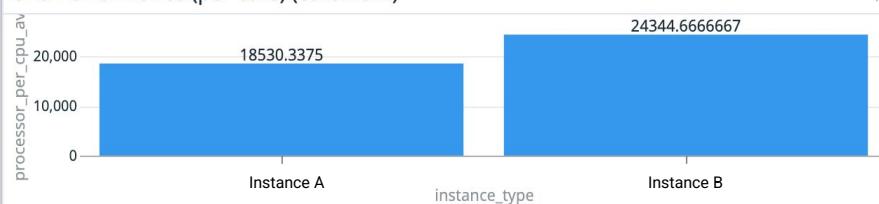
Network Bandwidth



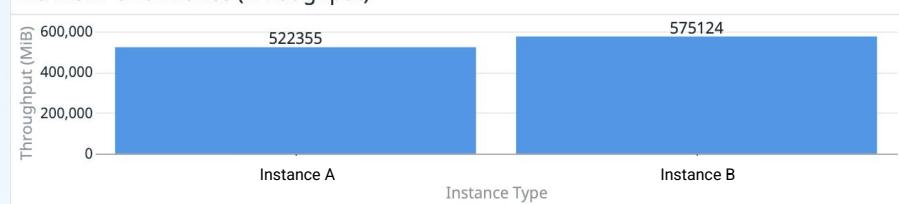
Memory Write - Single Thread



CPU Performance (per core) (coremark)



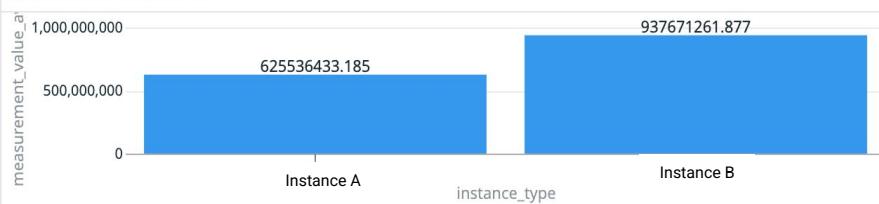
Fio Disk Performance (Throughput)



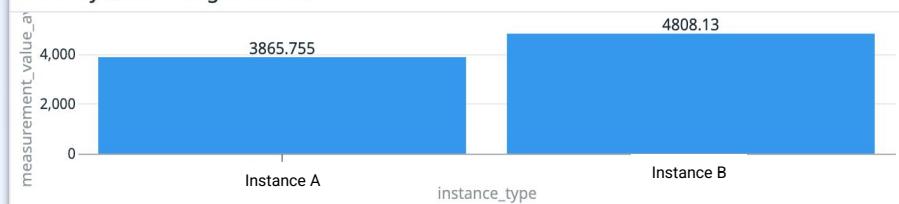
Performance Benchmarks for network, cpu, memory, and storage for us to weigh cost vs. performance for every instance type.

Performance: Benchmarks

Network Bandwidth



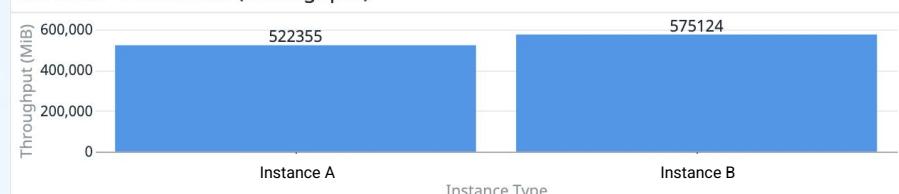
Memory Write - Single Thread



CPU Performance (per core) (coremark)



Fio Disk Performance (Throughput)



Performance Benchmarks for network, cpu, memory, and storage for us to weigh cost vs. performance for every instance type.

"Instance B is 20% more expensive than Instance A, but 30% more performant".

Reliability: Instance Attributes

We have the flexibility to express various instance type reliability preferences via **CEL Go selectors**.

This allows us to specify preferences such as:

- Capacity per environment
- Instance size ranges
- Performance preferences

```
message Environment {  
    string cluster = 1;  
    string provider = 2;  
    string region = 3;  
    string datacenter = 4;  
    string environment = 5;  
    string lineage = 6;  
    string flavor = 7;  
    int64 maxnodes = 8;  
    bool edge = 9;  
    bool experimental = 10;  
}
```

```
message InstanceType{  
    string name = 1;  
    string provider = 2;  
    string region = 3;  
    string arch = 4;  
    string family = 5;  
    int64 cpus = 6;  
    double memory = 7;  
    int64 generation = 8;  
    double cost = 9;  
    int64 bandwidth = 10;  
}
```

Reliability: Instance Attributes

- `instance.family in ["m6g", "r6g", "c6g"] && instance.cpus >= 8 && instance.cpus <= 32`
- `instance.family in ["m5n", "r5n", "c5n"] && env.edge`
- `Instance.generation==8 && env.cluster in ["cluster-foo", "cluster-bar"]`
- `instance.family == "fancy-new-type" && env.experimental`

Instance Type Selection: Simplified Criteria



Instance Type Advisor



Performance
Benchmarks



Instance Attribute
Selectors

Our Goals

1. **Identify** the best instance types
2. **Scale** the best instance types



Autoscaling Ecosystem

Instance Analysis Tools

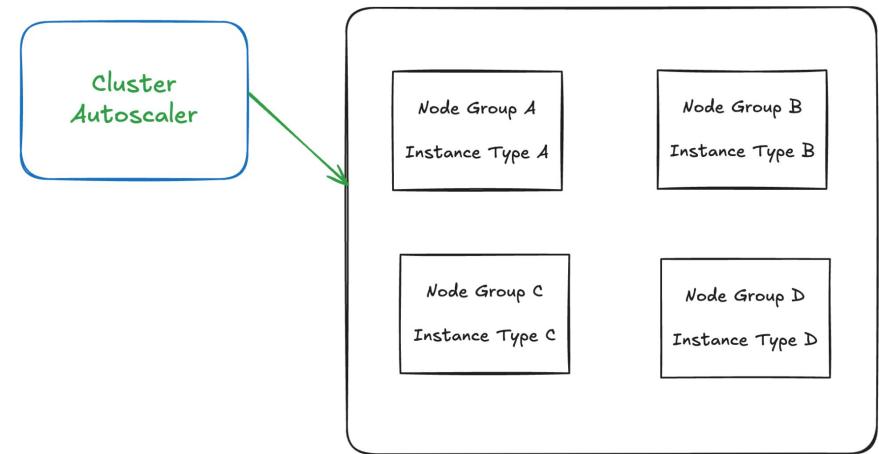
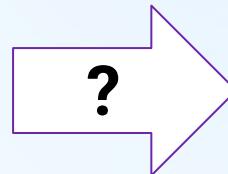
Instance Type Advisor

Capacity

Performance Benchmarks

Human Preferences

etc.



How does the
Cluster Autoscaler
know what the best
instance type is?

Autoscaling Ecosystem

Instance Analysis Tools

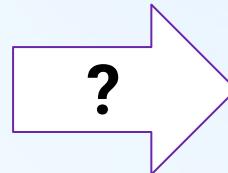
Instance Type Advisor

Capacity

Performance Benchmarks

Human Preferences

etc.



How does the gRPC Expander know what the best instance type is?

Cluster Autoscaler

gRPC Expander

Node Group A
Instance Type A

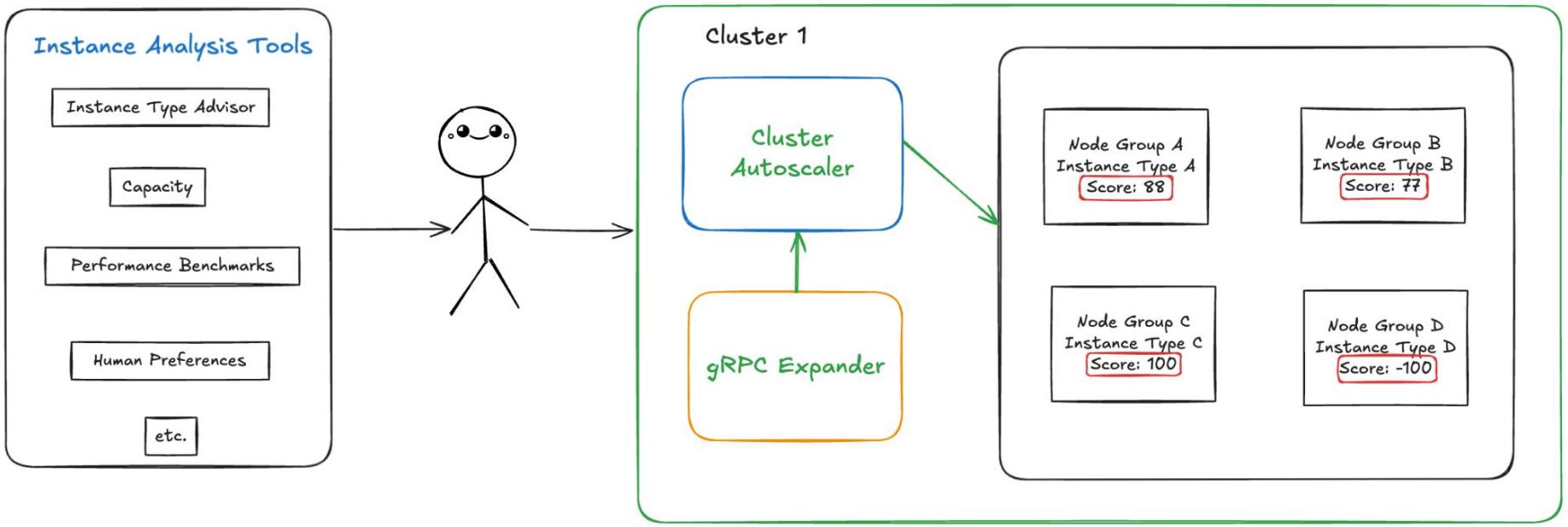
Node Group B
Instance Type B

Node Group C
Instance Type C

Node Group D
Instance Type D

Autoscaling Ecosystem

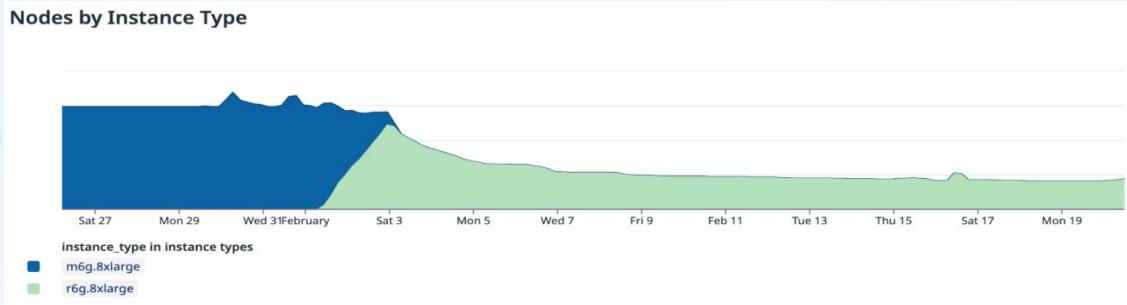
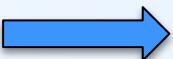
A human can translate the preferences into something the gRPC Expander can read



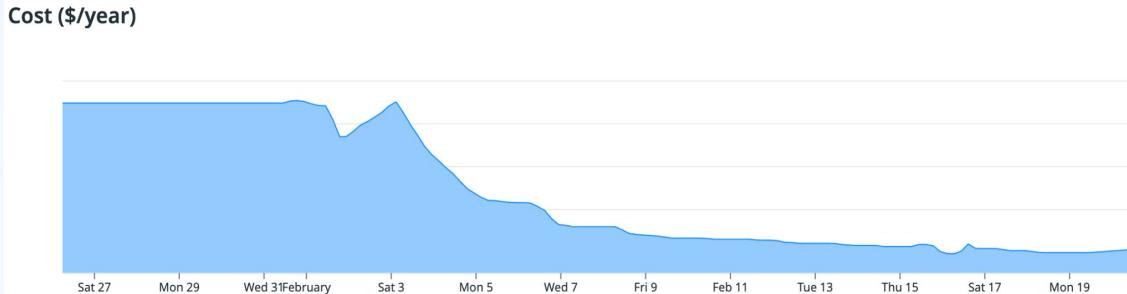
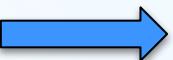
Results

Apply: transition to suggested type

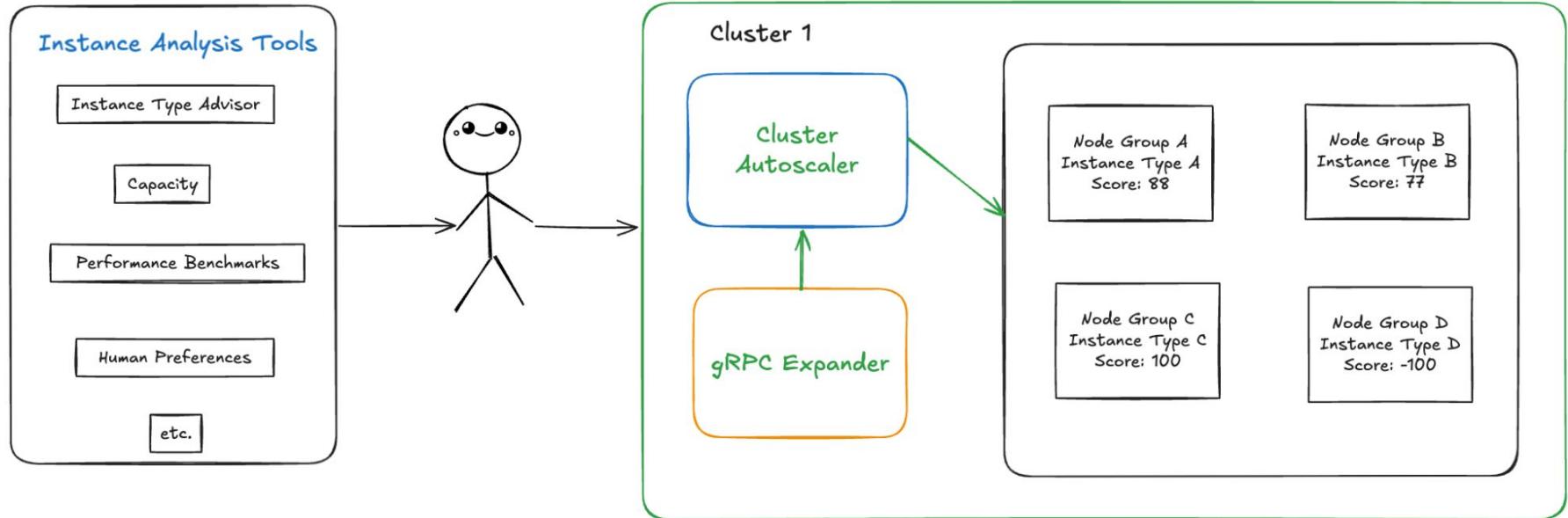
m6g.8xlarge -> r6g.8xlarge



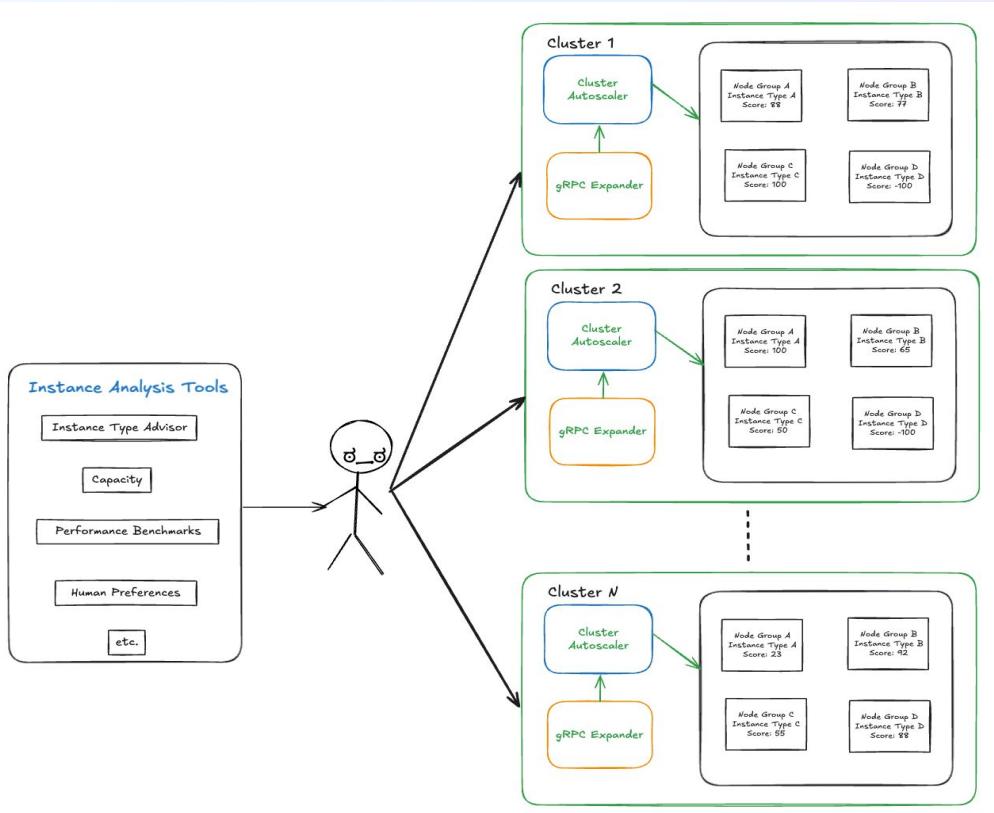
Save: we reduced costs by over 60% in a single cluster



Autoscaling Ecosystem



Autoscaling Ecosystem



This does not scale
with many clusters

Scores can frequently
change

Autoscaling Ecosystem

Instance Analysis Tools

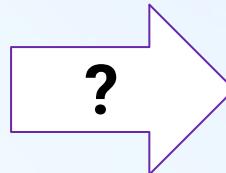
Instance Type Advisor

Capacity

Performance Benchmarks

Human Preferences

etc.



We need a scalable and dynamic way to know the best instance type

Cluster
Autoscaler

gRPC Expander

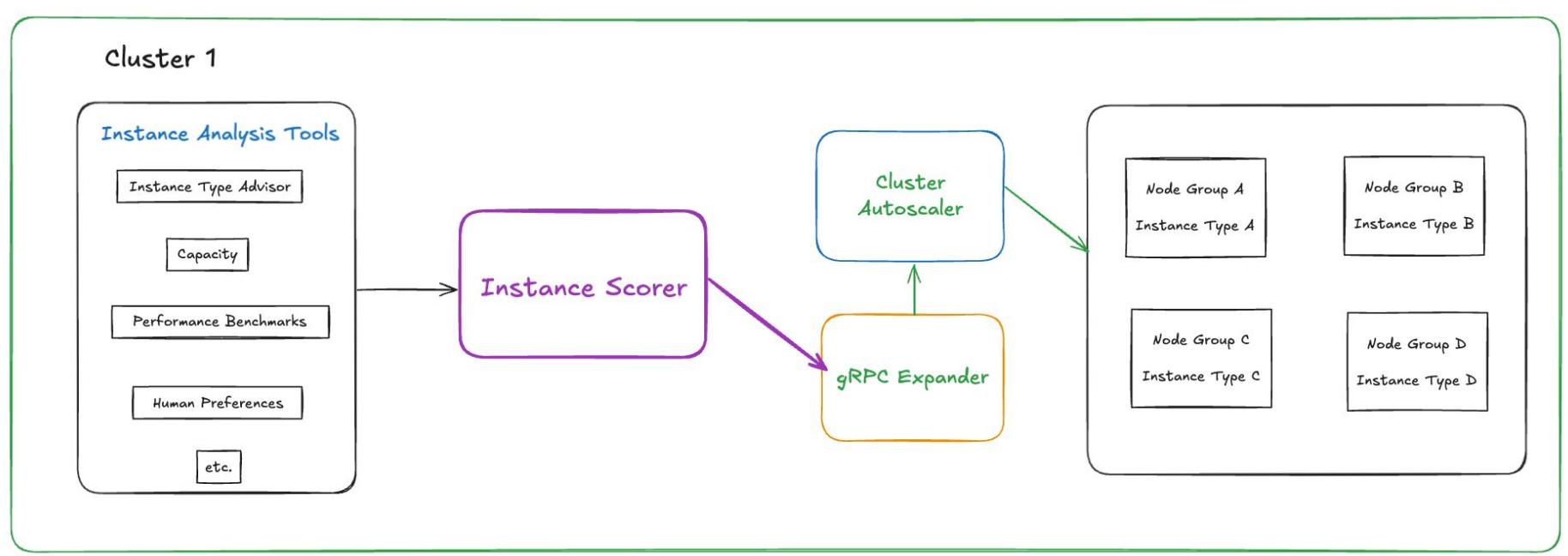
Node Group A
Instance Type A

Node Group B
Instance Type B

Node Group C
Instance Type C

Node Group D
Instance Type D

Dynamic Scaling Ecosystem



Results

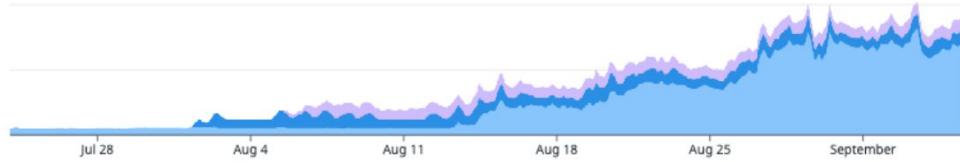
- Estimated savings of \$4M/year
- Migration done automatically across dozens of clusters

Potential Cost Savings

= Current Cost - Optimal Cost

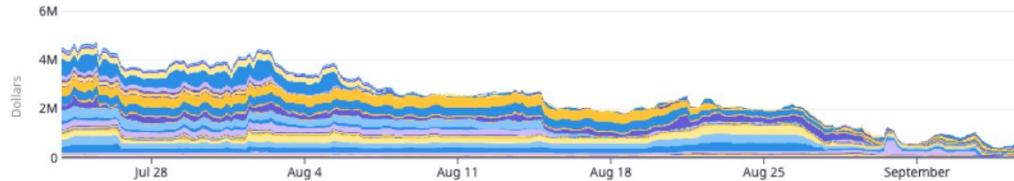
Instances Migrated

1mo Jul 23, 10:07 pm – Sep 5, 8:13 pm



Yearly Potential Cost Savings

1mo Jul 23, 10:07 pm – Sep 5, 8:13 pm



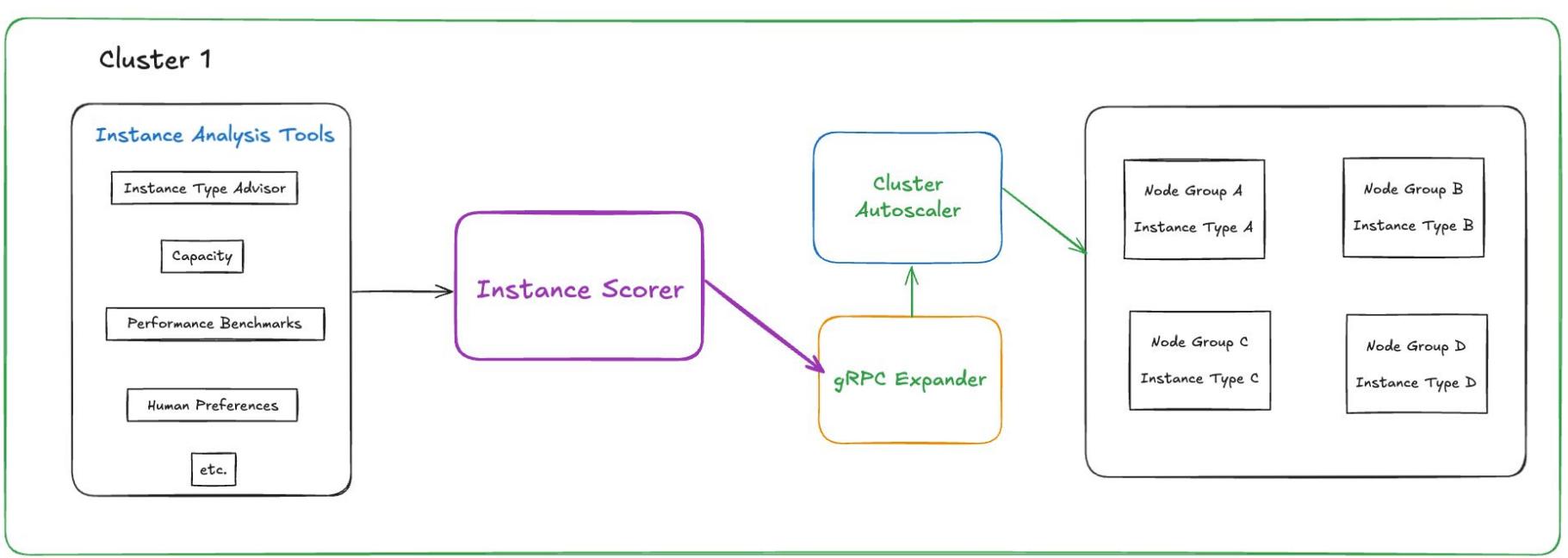
Instance Types

- Cost
- CPU
- Memory
- Storage
- Performance
- Capacity
- etc.

A dense network of overlapping text labels representing various AWS instance types, including:

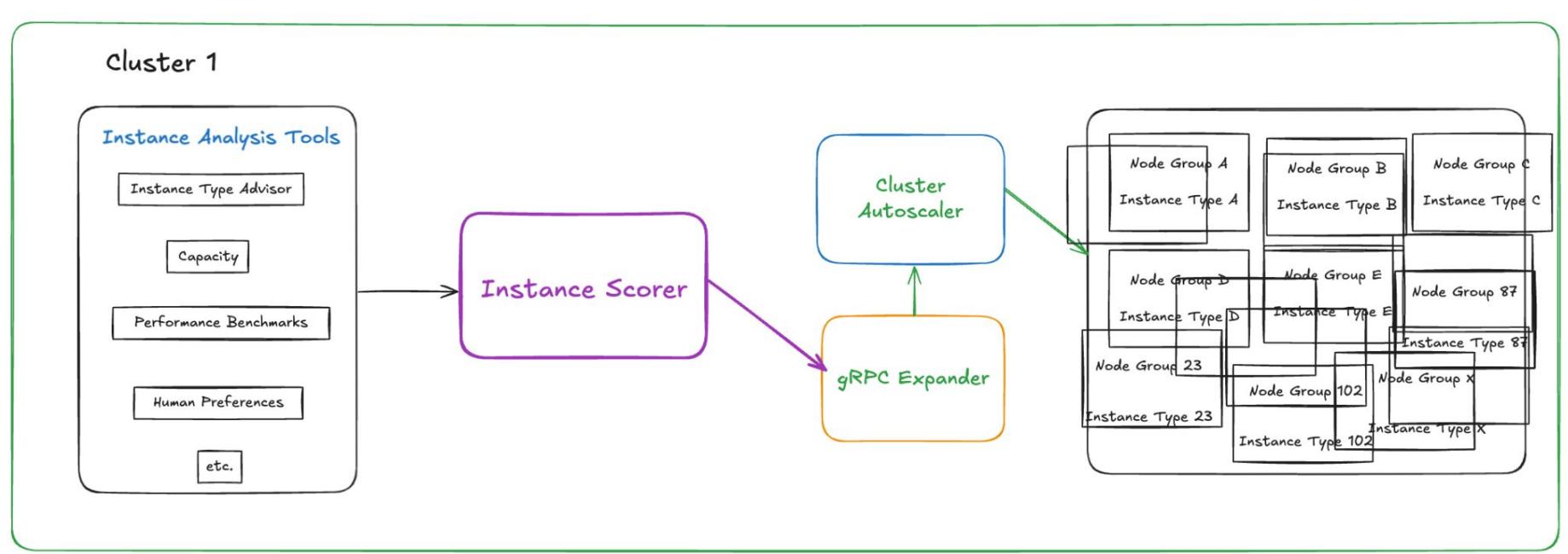
- i3.xlarge
- r6g.large
- r6i.xlarge
- d3.xlarge
- r5a.large
- trn1.32xlarge
- t3a.small
- c7g.large
- c6i.large
- r5a.12xlarge
- i3en.2xlarge
- t2.large
- c5.large
- g5.4xlarge
- r7g.12xlarge
- r6g.xlarge
- p3.8xlarge
- t2.micro
- r6i.2xlarge
- c7g.12xlarge
- r6i.4xlarge
- m5.xlarge
- r7g.xlarge
- c5n.18xlarge
- m5.24xlarge
- m5n.xlarge
- t3.xlarge
- g5.2xlarge
- c6g.2xlarge
- m7g.metal
- inf1.6xlarge
- c7g.2xlarge
- x2dn.16xlarge
- p4de.2xlarge
- c5.12xlarge
- m5.12xlarge
- c6i.16xlarge
- c6g.large
- d3.2xlarge
- m3.4xlarge
- m7g.16xlarge
- m5a.16xlarge
- c7g.4xlarge
- t2.xlarge
- r6i.metal
- g6g.12xlarge
- is4gen.xlarge
- x2iedn.16xlarge
- i3en.24xlarge
- r6g.2xlarge
- r5.8xlarge
- t3.medium
- is4gen.2xlarge
- i3.2xlarge
- r6i.16xlarge
- c5.24xlarge
- t3a.2xlarge
- r6g.2xlarge
- c6g.4xlarge
- im4gn.2xlarge
- m5.24xlarge
- i3en.24xlarge
- r6g.2xlarge
- c6i.12xlarge
- m6g.2xlarge
- inf1.16xlarge
- c6i.32xlarge
- e6i.2xlarge
- im4gn.large
- m5n.16xlarge
- m5n.8xlarge
- m5n.2xlarge
- r6g.8xlarge
- c6i.large
- m6g.2xlarge
- inf1.16xlarge
- i3.16xlarge
- r6i.16xlarge
- c5.24xlarge
- m5.24xlarge
- i3en.24xlarge
- r6g.2xlarge
- c6i.nano
- m6g.2xlarge
- inf1.16xlarge
- i3.16xlarge
- r6i.16xlarge
- c5.24xlarge
- i3.2xlarge
- r6g.2xlarge
- trn1.2xlarge
- c5a.12xlarge
- g4dn.4xlarge
- i3.16xlarge
- r6i.16xlarge
- c6i.32xlarge
- e6i.2xlarge
- im4gn.large
- m5n.16xlarge
- m5n.8xlarge
- m5n.2xlarge
- r6g.8xlarge
- c6i.nano
- m6g.medium
- inf1.16xlarge
- i3.16xlarge
- r6i.16xlarge
- c5.24xlarge
- i3.2xlarge
- r6g.2xlarge
- r5a.24xlarge
- c5a.xlarge
- r5.8xlarge
- x2dn.8xlarge
- is4gen.12xlarge
- g4dn.16xlarge
- i3.16xlarge
- r6i.16xlarge
- c6i.32xlarge
- e6i.2xlarge
- im4gn.large
- m5n.16xlarge
- m5n.8xlarge
- m5n.2xlarge
- r6g.8xlarge
- inf1.2xlarge
- is4gen.medium
- r5.8xlarge
- t2.small
- t2.medium
- r7g.5.5xlarge
- g4dn.8xlarge
- i3a.large
- m5n.12xlarge
- m5n.24xlarge
- m5.16xlarge
- r6g.4xlarge
- inf1.24xlarge
- p3.16xlarge
- r7g.8xlarge
- x2iedn.32xlarge
- r7g.8xlarge
- m5a.24xlarge
- m5.9xlarge
- m7g.medium
- t2.nano
- d2.4xlarge
- is4gen.8xlarge
- c7g.medium
- x2dn.24xlarge
- r6g.4xlarge
- inf1.24xlarge
- p3.16xlarge
- r6i.16xlarge
- m6g.16xlarge
- r5a.xlarge
- r7g.2xlarge
- m5a.12xlarge
- m6g.xlarge
- c5.4xlarge
- m5.48xlarge
- m5a.xlarge
- c6g.8xlarge
- c5a.16xlarge
- im4gn.4xlarge
- m5a.4xlarge
- m6g.4xlarge
- r7g.medium
- i3.16xlarge
- c6i.8xlarge
- trn1.32xlarge
- m5a.8xlarge
- m5.16xlarge
- c7g.16xlarge
- i3en.3xlarge
- m6g.large
- r5.16xlarge
- m5a.8xlarge
- m5.16xlarge
- c5n.9xlarge
- t3.2xlarge
- i3.4xlarge
- d3.8xlarge
- e6i.12xlarge
- c7g.8xlarge
- c5n.9xlarge
- t3.2xlarge
- r5.16xlarge
- r6i.12xlarge
- c5a.2xlarge
- r5.24xlarge
- r7g.large
- r5.2xlarge
- t2.2xlarge
- is4gen.large
- inf1.xlarge
- c6g.metal
- c5a.large
- i3a.xlarge
- r5.xlarge
- r5.2xlarge

Instance Types

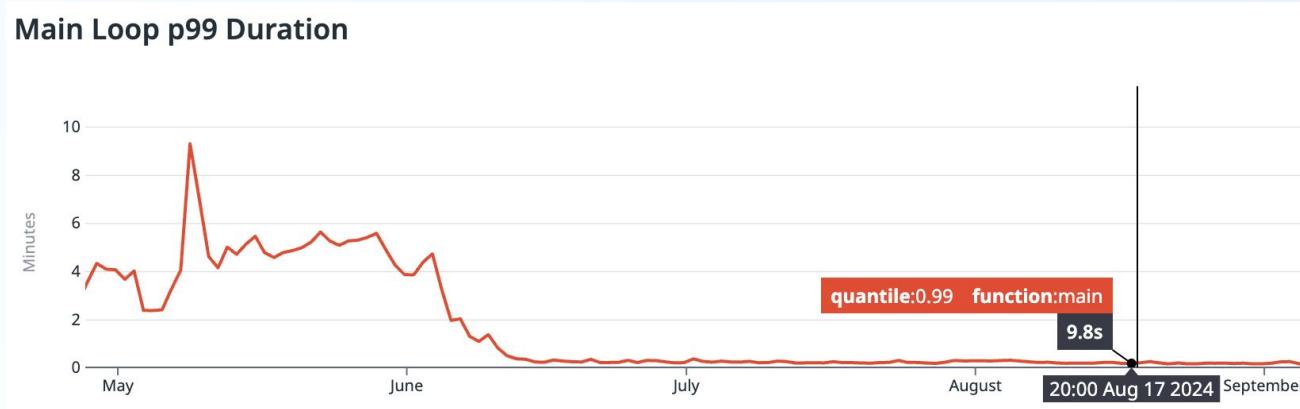
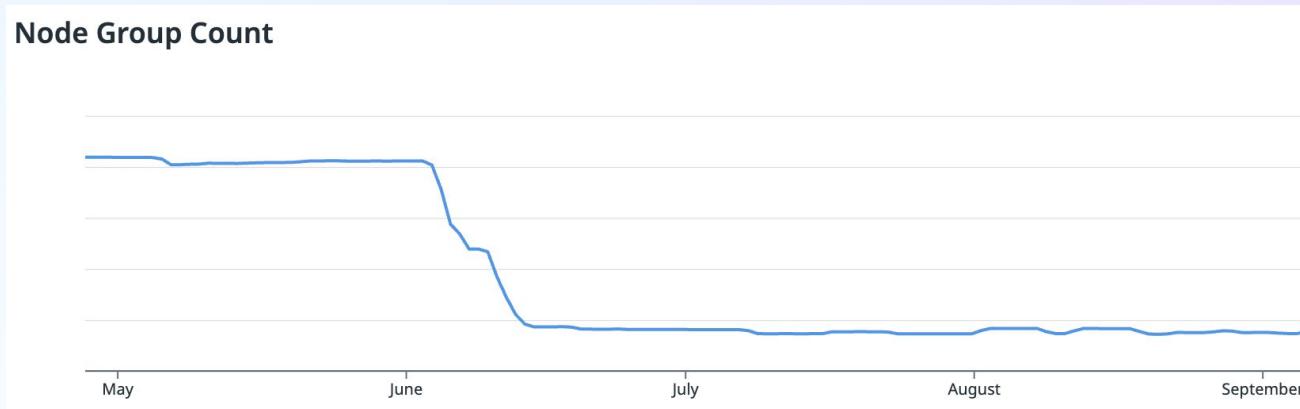


How do make sure the best node group exists?

Instance Types x 100+



Cluster Autoscaler Performance vs # Node Groups



Platform Requirements

Our platform must be able to:

- **Scaleup the most cost-efficient instance type**

Node Group Set

Platform Requirements

Our platform must be able to:

- **Scaleup the most cost-efficient instance type**

Node Group Set

- **Best instance type**

Platform Requirements

Our platform must be able to:

- Scaleup the most cost-efficient instance type
- Schedule pods requesting up to 64 CPU and 256GB Memory

Node Group Set

- Best instance type

Platform Requirements

Our platform must be able to:

- Scaleup the most cost-efficient instance type
- Schedule pods requesting up to 64 CPU and 256GB Memory

Node Group Set

- Best instance type
- **Best big instance type**

Platform Requirements

Our platform must be able to:

- Scaleup the most cost-efficient instance type
- Schedule pods requesting up to 64 CPU and 256GB Memory
- **Provide fallback instances in case the main instance types run out of capacity**

Node Group Set

- Best instance type
- Best big instance type

Platform Requirements

Our platform must be able to:

- Scaleup the most cost-efficient instance type
- Schedule pods requesting up to 64 CPU and 256GB Memory
- **Provide fallback instances in case the main instance types run out of capacity**

Node Group Set

- Best instance type
 - **2 fallbacks**
- Best big instance type
 - **2 big fallbacks**

Platform Requirements

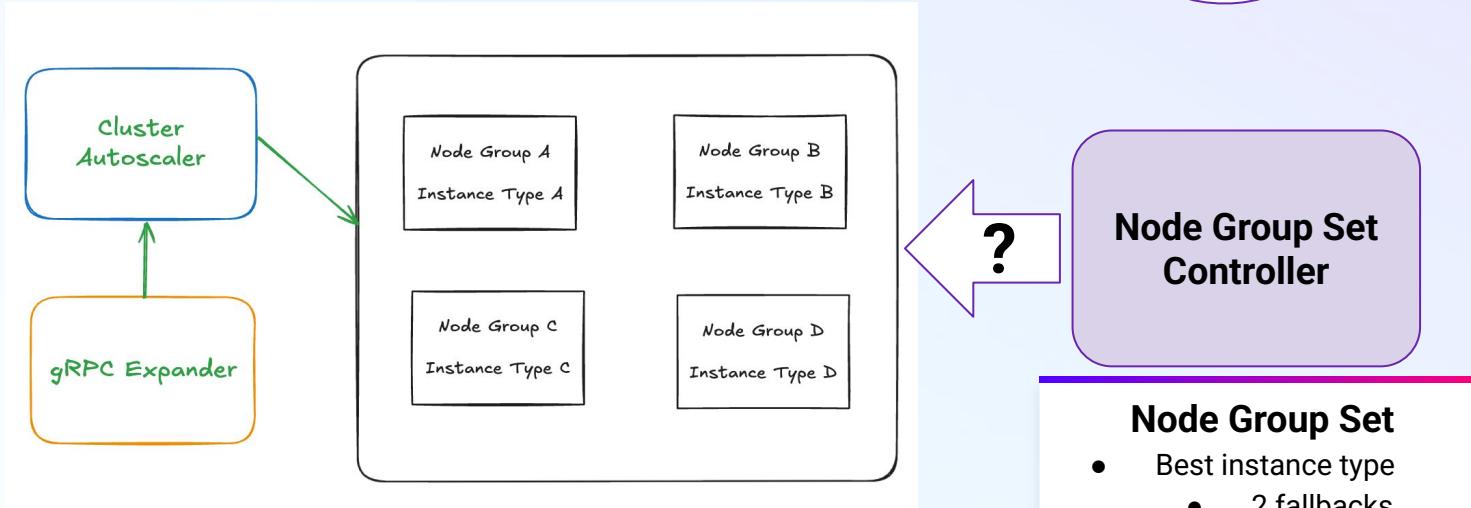
Our platform must be able to:

- Scaleup the most cost-efficient instance type
- Schedule pods requesting up to 64 CPU and 256GB Memory
- Provide fallback instances in case the main instance types run out of capacity

Node Group Set

- Best instance type
 - 2 fallbacks
- Best big instance type
 - 2 big fallbacks

Node Group Set Controller



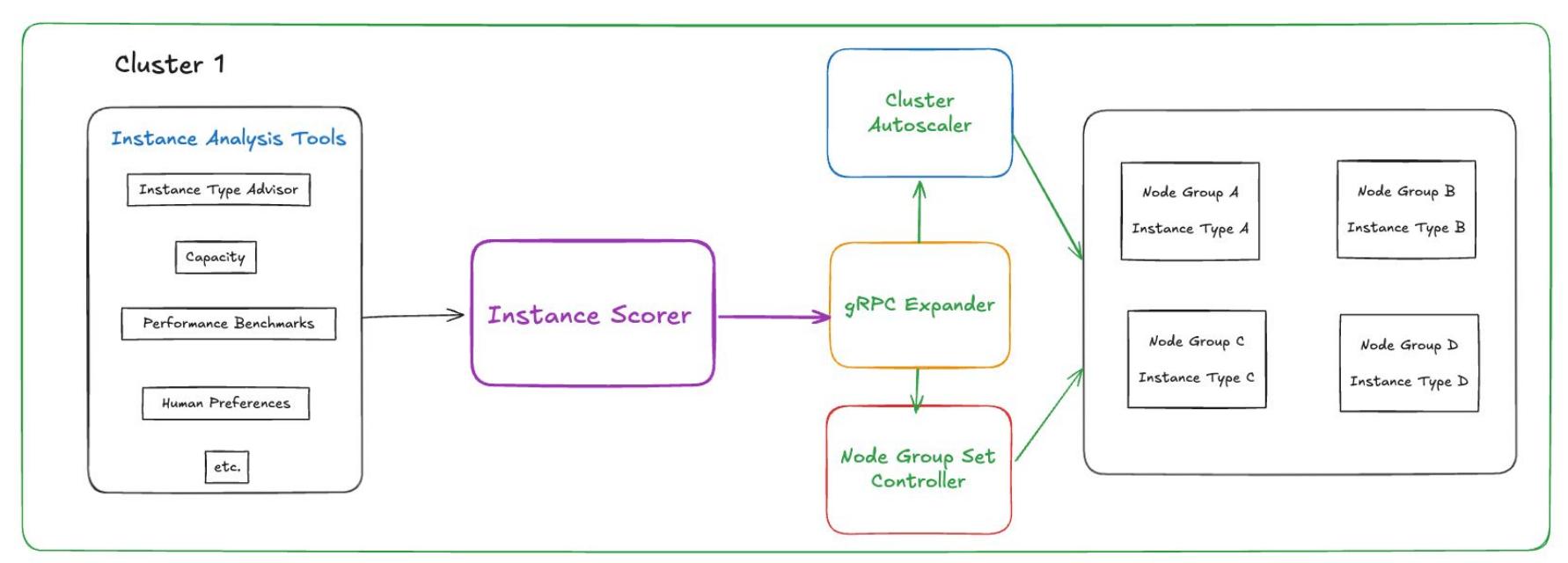
How does the Node Group Set Controller know what the best instance type is?

Node Group Set Controller

Node Group Set

- Best instance type
 - 2 fallbacks
- Best big instance type
 - 2 big fallbacks

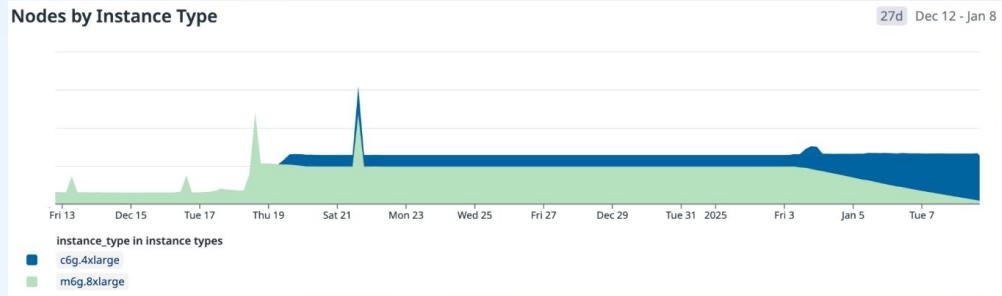
gRPC Expander Again



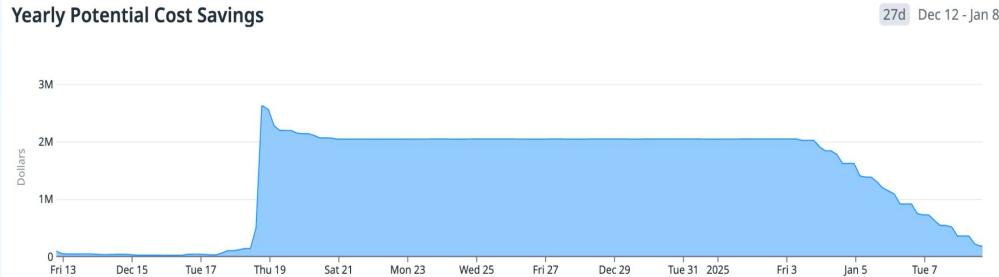
Results

New application deployed,
caused shift in optimal instance
type

m6g.8xlarge -> c6g.4xlarge



Save: observe ~\$2M/year
savings



Our Goals

1. **Identify** the best instance types
2. **Scale** the best instance types



Next Steps



Continue abstracting infrastructure details



**Providing varieties of instance types
(network enhanced, local storage, etc)**



Optimizing placement of applications

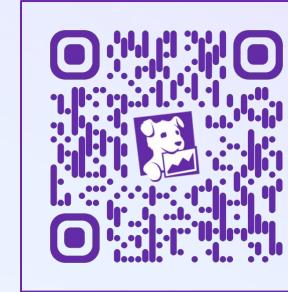
Thank you



Rahul Rangith

Software Engineer, Datadog

<https://www.linkedin.com/in/rrangith>



Ben Hinthorne

Software Engineer, Datadog



<https://www.linkedin.com/in/ben-hinthorne>

Feedback:

