
KubeCon
CloudNativeCon
Europe 2021
Virtual
Forward Together »

Groupless Autoscaling with Karpenter

KubeCon CloudNativeCon Europe 2021

Virtual

Who are we?

Ellis Tarn
Software Engineer (AWS/EKS)

Prateek Gogia
Software Engineer (AWS/EKS)

aws

Overview

- Traditional Autoscaling (horizontal)
  - Pod Autoscaling
  - Node Autoscaling
  - E2E Mechanics and Edge Cases

- Groupless Autoscaling
  - Launching Capacity
  - Scheduling
  - Bin Packing
  - Defragmentation
  - Cloud Provider

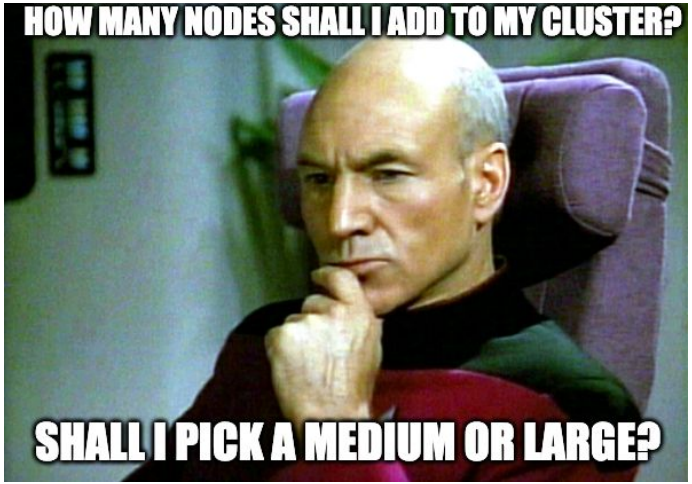
- Demo
- Q&A

So where do pods come from, anyways?

- Creation of Jobs and Deployments
- CronJobs
- Scaling existing Deployments
  - HPA
  - KEDA
  - Knative

… and what about nodes?

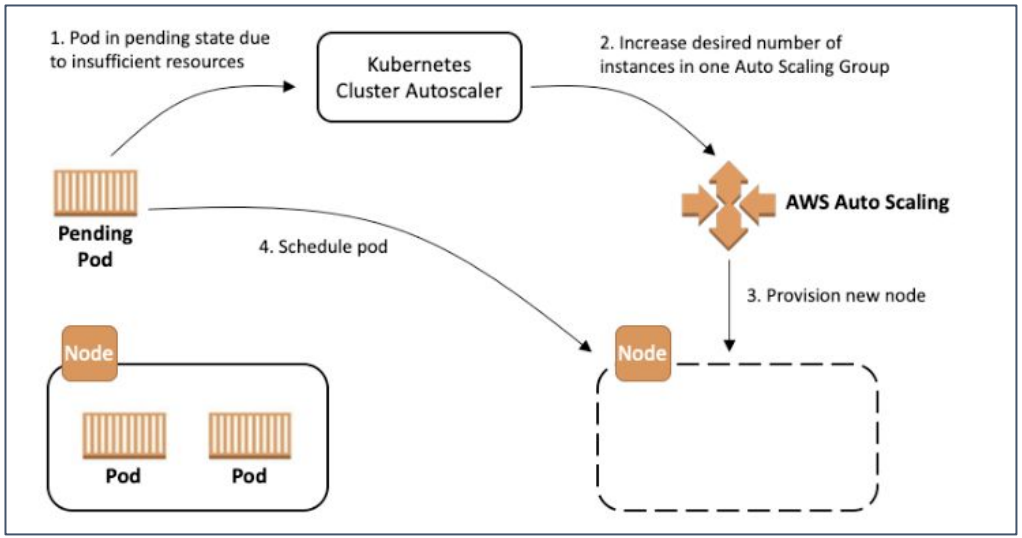
- Manual provisioning
- Cluster Autoscaler
  - Zalando (cluster autoscaler fork)
- Other attempts
  - Escalator
  - Cerebral



Cluster autoscaler Benefits?

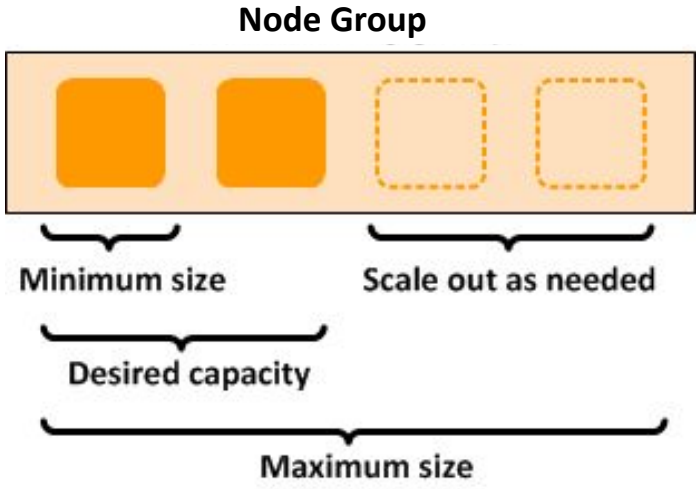
- Automatically grow the cluster to meet increased demands
- Reduce infrastructure costs through efficient resource utilization and node termination
- Vendor neutral with support for all major cloud providers
- Widely adopted and battle tested approach
- Works great for cluster sizes ~1000 nodes

How does cluster autoscaler work?



Node Group

- Well known capacity templates
- Mirroring the data center “Rack” model
- Building upon existing layers in the cloud



Drawbacks and Caveats?

- One node group per AZ * Instance Type * Labels
- Timeout-based error handling
- Tested up to 1000 nodes and 30 pods/node
- Operates globally in the cluster
- Complex to operate with 78 cli flags
- Doesn’t work with custom scheduler

Traditional Node Autoscaling

Other approaches

- **Cerebral (deprecated)**
  - An autoscaler with pluggable metrics backends and scaling engines
  - Cluster autoscaler - often triggering events too late to be rendered useful

- **Escalator**
  - The need for this autoscaler is derived from our own experiences with very large batch workloads being scheduled and the default autoscaler not scaling up the cluster fast enough.
  - Designed to work on selected auto-scaling groups to allow the default Kubernetes Autoscaler to continue to scale service based workloads

Other approaches (cont.)

- **Zalando changes to cluster autoscaler**
  - More robust template node generation
  - Support for AWS autoscaling groups with multiple instance types
  - More reliable backoff logic
  - Improved handling of template nodes

Traditional Node Autoscaling

- Can we simplify and do better?
- Can we perform faster on larger clusters?
- Can we avoid painful failure modes?

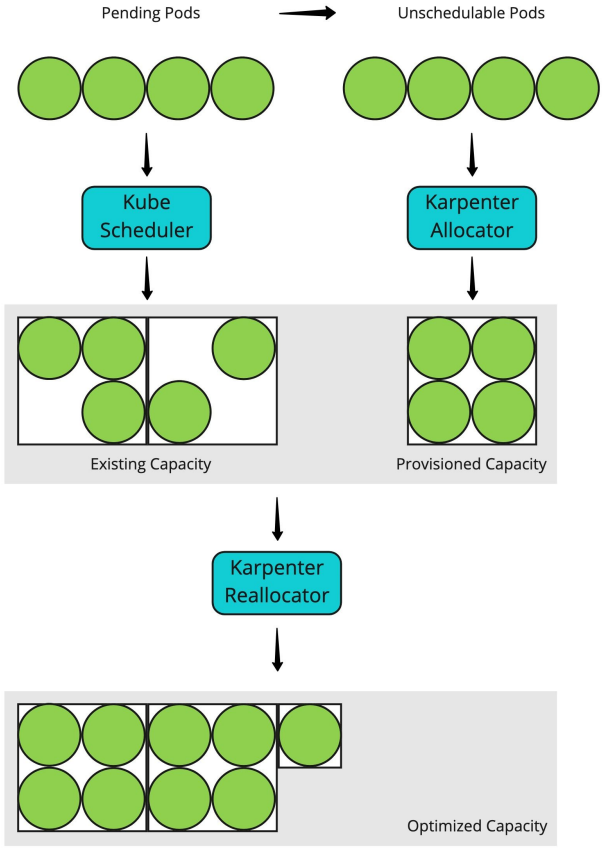
Groupless Node Autoscaling

What if we remove the concept of Node Groups?

- Provision capacity directly
- Choose instance types from pod resource requests
- Generate node properties from pod scheduling constraints
- Track nodes using native Kubernetes labels

- Reduces configuration burden
- Reduces Cloud Provider API Load
- Reduces simulation complexity

Groupless Node Autoscaling



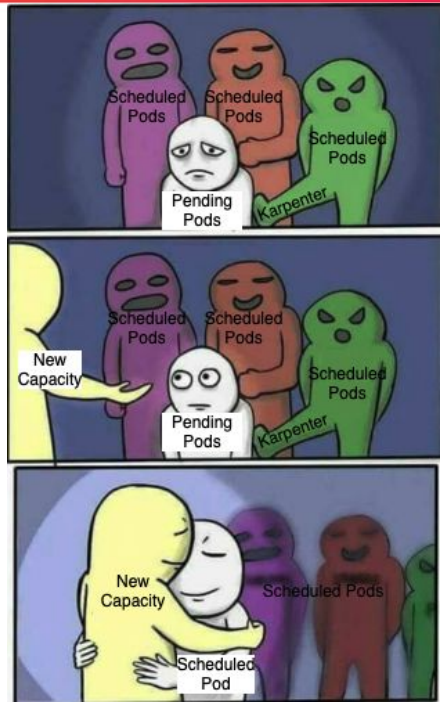
Allocator and Reallocator

- Fast acting, latency sensitive controller
- Slow acting, cost sensitive controller

Launching Capacity Directly

- Using the EC2.*CreateFleet()* API
- Compute a list of viable instance types
- Specify the list of viable availability zones
- EC2 Fleet picks the cheapest instances given the constraints

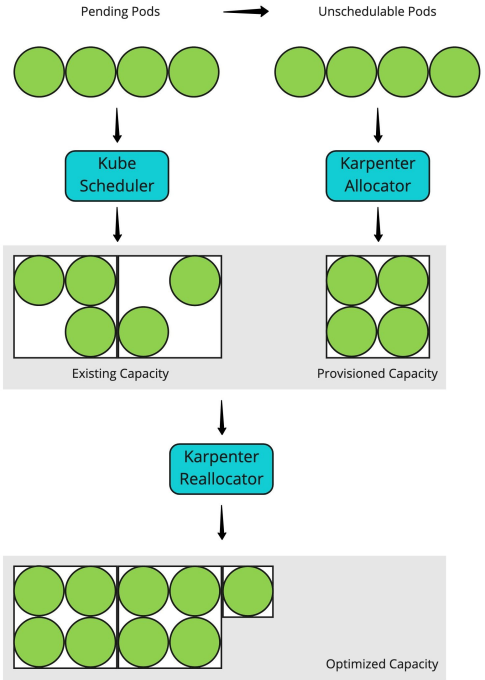
- Increases node flexibility
- Reduces node provisioning latency



Groupless Node Autoscaling

Scheduling

- Working in tandem with kube-scheduler
- Provisioning decisions == scheduling decisions
- Binding to provisioned nodes
- Cross version Kubernetes compatibility



aws

Scheduling API Conformance

- Resource Requests (CPU, Memory, GPU, HPC)
- Node Selectors & Node Affinity
- Topology Spread Constraints & Pod Affinity

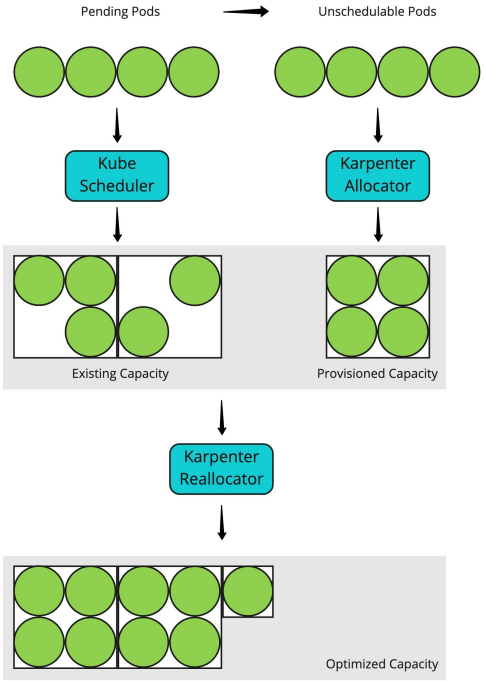
Well Known Labels

- topology.kubernetes.io/zone = us-west-2a
- node.kubernetes.io/instance-type = m5.large
- node.k8s.aws/capacity-type = spot
- kubernetes.io/arch = arm64
- kubernetes.io/os = linux

Groupless Node Autoscaling

Bin Packing (Online Bin Packing)

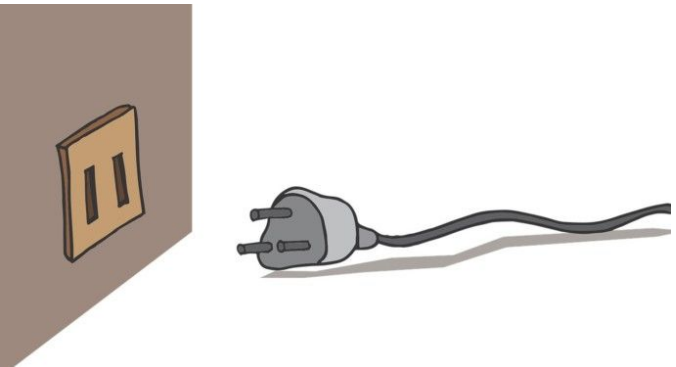
- First Fit Descending
- Prioritize CPU, Memory, or Euclidean?
- Room for innovation with new algorithms



aws

Termination

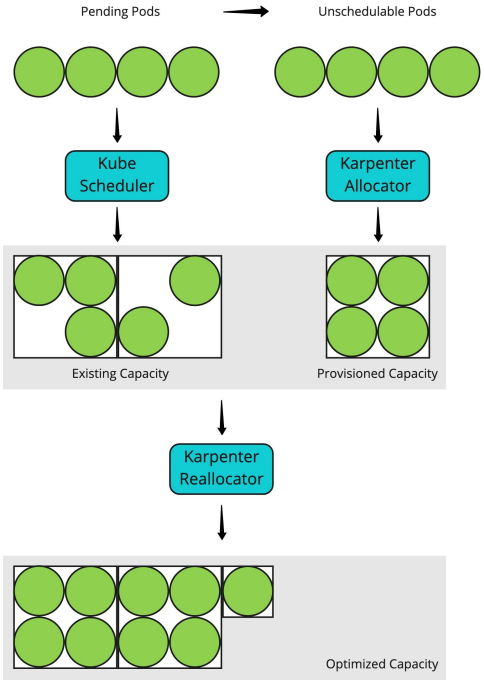
- Remove underutilized nodes
- Shutdown grace period
- Spot rebalance events
- Spot termination events
- Cordon and drain
- EC2 unhealthy events
- Node TTL



Groupless Node Autoscaling

Defragmentation (Offline Bin Packing)

- Global optimization
- Re-pack inefficiently scheduled pods
- Replace instances in accordance with spot markets
- Complexities around pod immutability
- Implementation TBD



Optimizing Provisioning Latency

- EC2 Instance Launch
- VM Boot Duration
- Kubelet Startup
- CNI Node Readiness
- CNI Pod Readiness
- Kube API Server QPS
- Kube Scheduler Client QPS
- Kube Controller Manager Client QPS

Groupless Node Autoscaling

Provisioner CRD

- Strongly typed configuration
- Provisioner defaults w/ pod overrides
- Workload isolation support
- Scalable sharding

```yaml
apiVersion: provisioning.karpenter.sh/v1alpha1
kind: Provisioner
metadata:
 name: default
spec:
 cluster:
   name: "${CLUSTER_NAME}"
   caBundle: "${CLUSTER_CA_BUNDLE}"
   endpoint: "${CLUSTER_ENDPOINT}"
 taints:
   - key: example.com/special-taint
     effect: NoSchedule
 labels:
   node.k8s.aws/capacity-type: "spot"
```

50 different flags for configuration

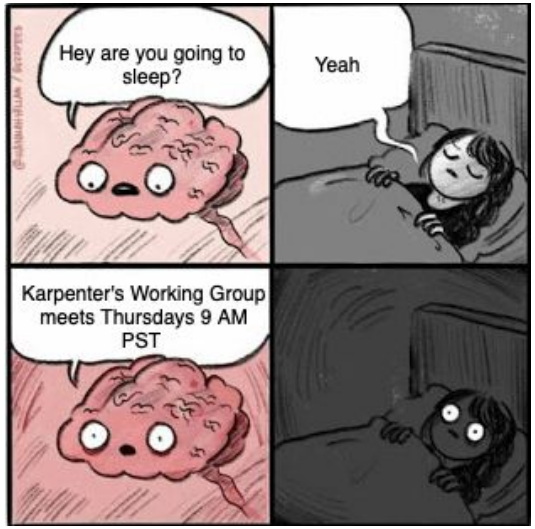
One .yaml file

aws

Groupless Node Autoscaling

github.com/awslabs/karpenter

- Open Source
- Vendor Neutral
- Incubating in awslabs
- Check out our roadmap
- Join our working group



aws

Demo