
# Scaling Kubernetes clusters without losing your mind or money

Yael Grossman

Sr. Compute Solutions Architect, AWS

# Agenda

Why Karpenter?

How Karpenter works

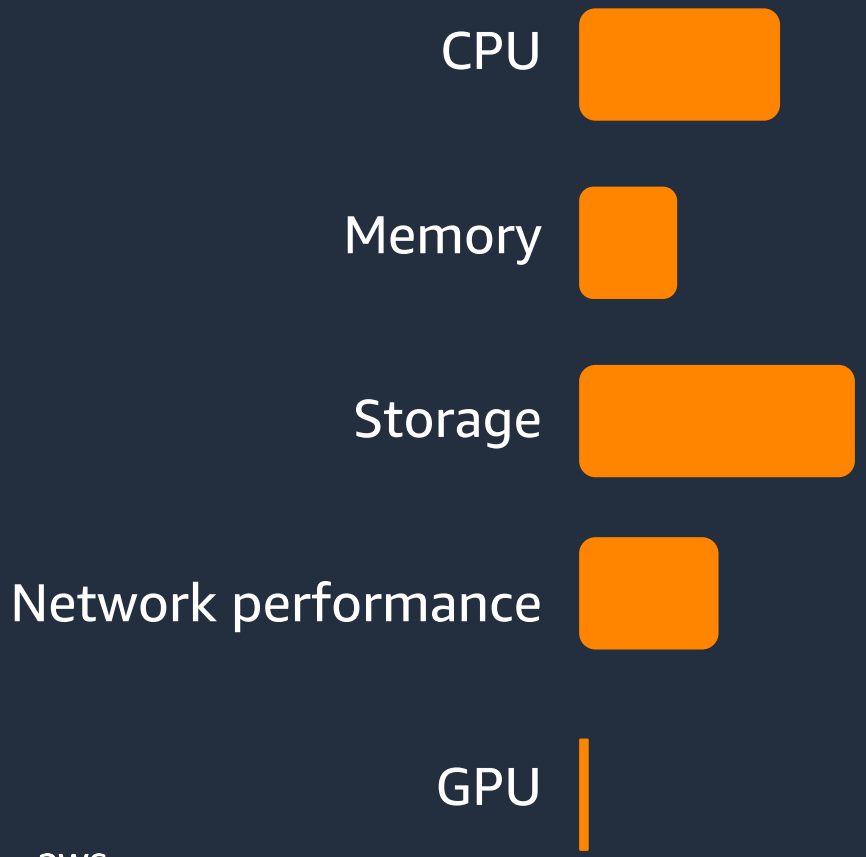
Karpenter and Flexible Compute

What’s Next

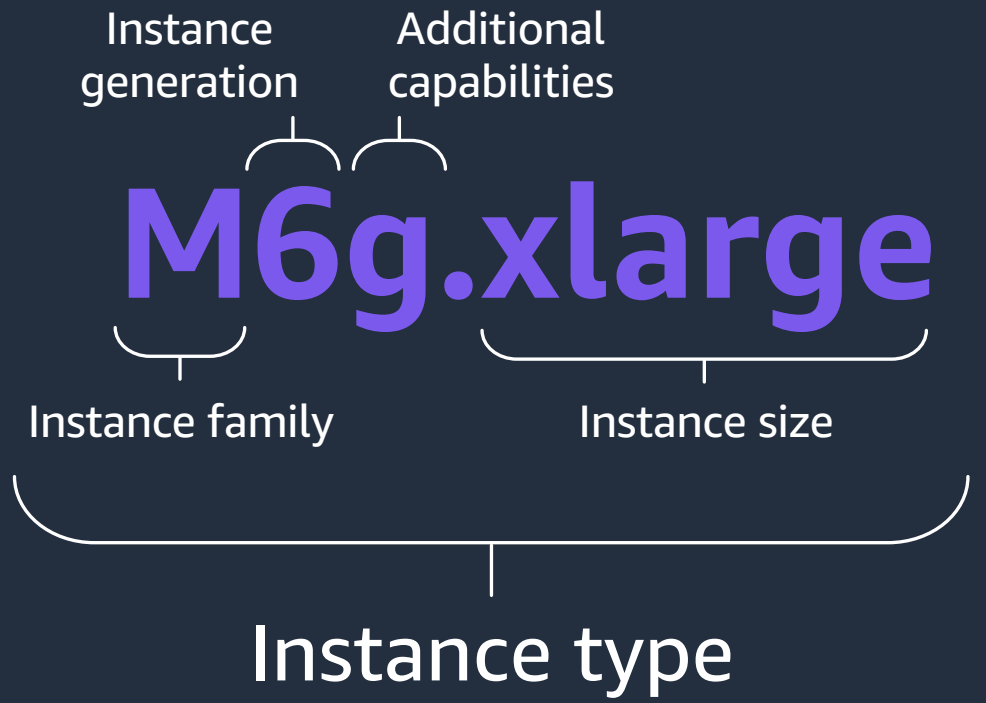
# Efficiency Requirements

| | |
|---|---|
| **Scale** | Scale up or down dynamically to minimize waste |
| **Density** | Maximize the utilization of compute resources within a scalable unit |
| **Flexibility** | Trade availability for cost or adjust compute resources to achieve higher utilization |

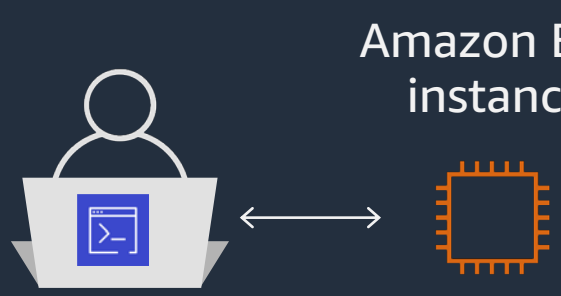
# Containers resource requirements



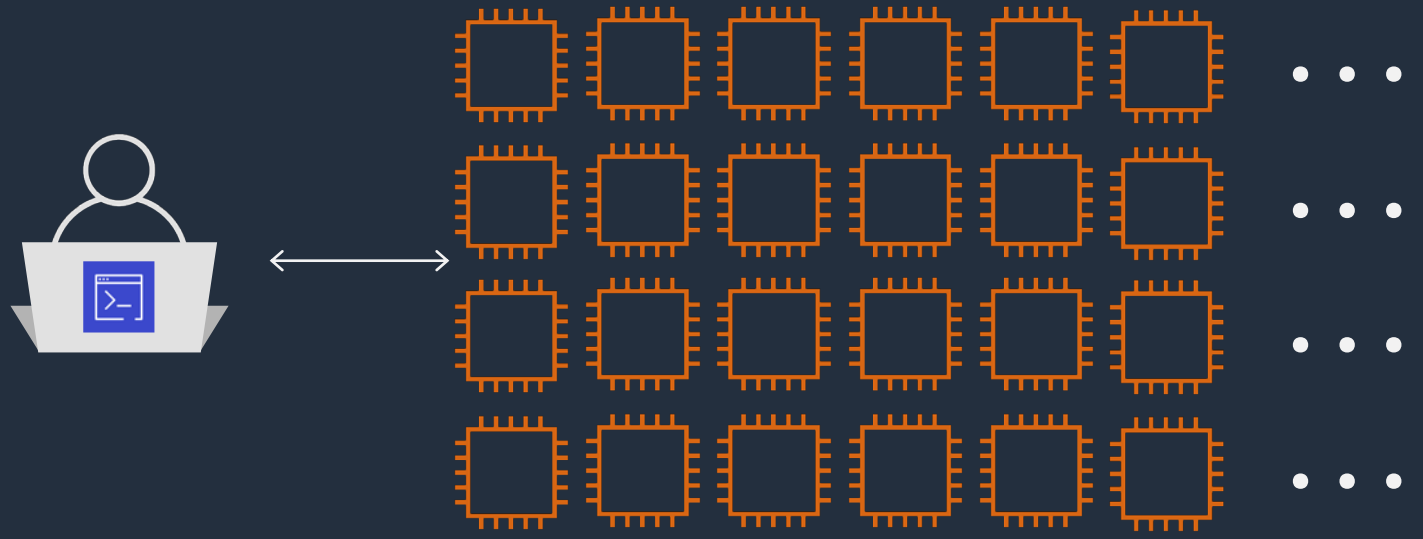

# Amazon EC2 instance characteristics



# So, how do we scale?

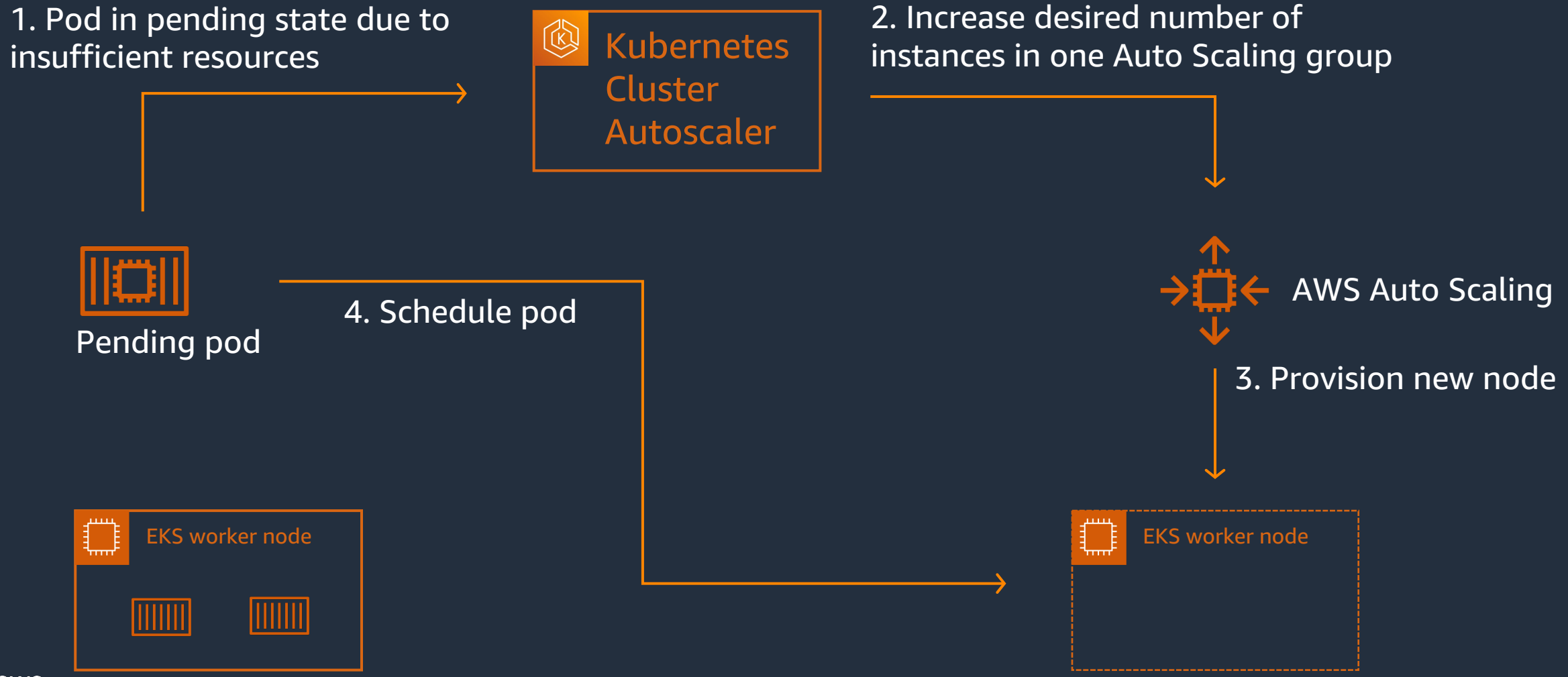



This



To this

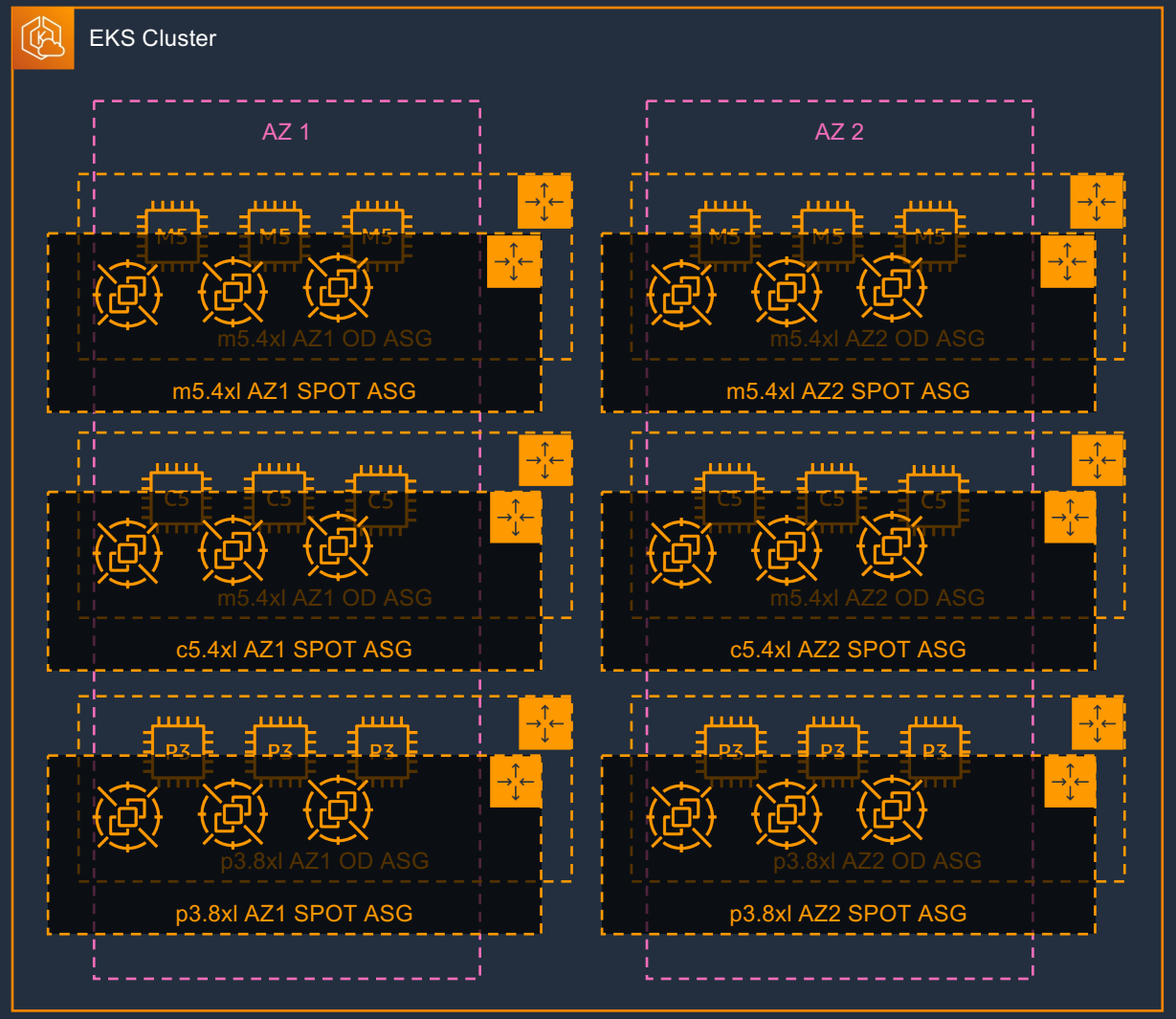
# Recap: Cluster Autoscaler scale-up



# Cluster Autoscaling is Challenging to Configure

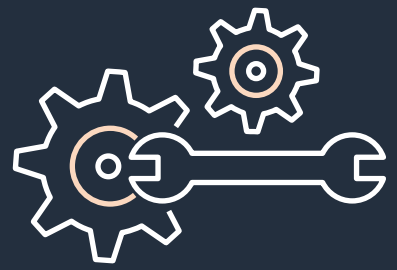
Nearly half of AWS Kubernetes customers tell us that configuring cluster autoscaling is challenging.

- Multi-AZ availability
- Instance type flexibility
- Spot capacity




# Karpenter - Cost-efficient compute for Kubernetes

# Karpenter – Cost-efficient compute for Kubernetes



Karpenter is an **intelligent** and **high-performance** Kubernetes compute provisioning and management solution

Karpenter lets you take **full advantage of AWS** with its deep integration between Kubernetes and Amazon EC2

# Karpenter

GROUPLESS PROVISIONING AND AUTO SCALING

**What if we remove the concept of node groups?**

- Improve the efficiency and cost of running workloads
- Simplification of configuration
- Kubernetes native
- Flexible compute built-in

- Provision capacity directly with “instant” EC2 Fleets
- Choose instance types from pod resource requests
- Provision nodes using K8s scheduling constraints
- Track nodes using native Kubernetes labels
- Fully supported by AWS and ready for production

# How Karpenter works


**Consolidates instance orchestration responsibilities within a single system**

# Provisioner CRD

- **Provisioner** – custom resource to provision nodes with a set of attributes (taints, labels, requirements, TTL)
- Single provisioner can manage compute for multiple teams and workloads
- Can also have multiple provisioners for isolating compute for different needs

```yaml
apiVersion: karpenter.sh/v1alpha5
kind: Provisioner
metadata:
  name: default
spec:
  consolidation:
    enabled: true
  requirements:
    # Include general purpose instance families
    - key: karpenter.k8s.aws/instance-family
      operator: In
      values: [c5, m5, r5]
    # Exclude small instance sizes
    - key: karpenter.k8s.aws/instance-size
      operator: NotIn
      values: [nano, micro, small, large]
    - key: karpenter.sh/capacity-type
      operator: In
      values: ["on-demand", "spot"]
    - key: kubernetes.io/arch
      operator: In
      values: ["amd64", "arm64"]
  providerRef:
    name: default
```

# Compute per workload scheduling requirements

# Compute per workload scheduling requirements

## Workloads may be required to run

In certain AZs

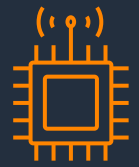
On certain types of processors or hardware (AWS Graviton, GPUs)

On Spot or on-demand capacity

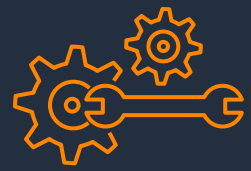
## Standard K8s pod scheduling mechanisms

Node selectors

Node affinity



Taints and tolerations



Topology spread

**Pod scheduling constraints must fall within a provisioner's constraints**

# Strategies for defining provisioners

## Single

A single provisioner can manage compute for multiple workloads

Example use cases:

- Single provisioner for a mix of Graviton and x86
- Single provisioner for Spot and On-Demand

## Multiple

Isolating compute for different purposes

Example use cases:

- Expensive hardware
- Security isolation
- Team separation
- Different AMI

## Prioritized

Define order across your provisioners

Example use cases:

- Prioritize SPa and RI ahead of other types
- Ratio split – Spot/OD, x86/Graviton

# Compute flexibility

## Instance types, Purchase options, CPU architecture

- No list → picks from all instance types in EC2 universe, excluding metal
- Attribute-based requirements → sizes, families, generations, CPU architectures

## Availability Zones

- Provision in any AZ
- Provision in specified AZs

```yaml
apiVersion: karpenter.sh/v1alpha5
kind: Provisioner
metadata:
  name: default
spec:
  requirements:
    - key: karpenter.k8s.aws/instance-family
      operator: In
      values: [c5, m5, r5]
    - key: topology.kubernetes.io/zone
      operator: In
      values: ["us-west-2a", "us-west-2b"]
    - key: karpenter.sh/capacity-type
      operator: In
      values: ["on-demand", "spot"]
    - key: kubernetes.io/arch
      operator: In
      values: ["amd64", "arm64"]
  providerRef:
    name: default
```

# Spot interruption handling with Karpenter

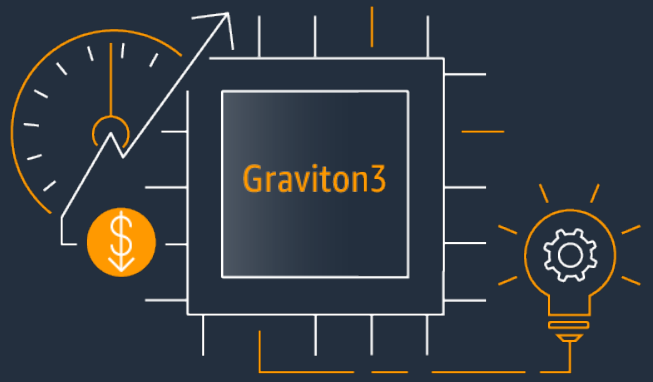
The work you are doing to make your applications fault-tolerant also enabled Spot

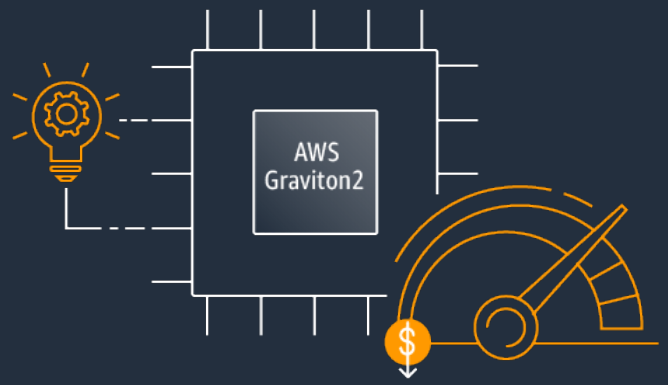
**Spot notification**

- 2-minute Spot Instance interruption notice via instance metadata or Event Bridge event

- Flexibility is key to successful adoption. Karpenter seamlessly supports flexibility across different instance types, sizes and Availability Zones
- Provisioners can be configured for a mix of On-Demand and Spot. Spot is prioritized if flexible to both capacity types.
- Use price-capacity-optimized allocation strategy for Spot Instances
- Built-in Spot instance lifecycle management

# CPU Architecture flexibility – Graviton based instances




Why run containers on Graviton with Karpenter?

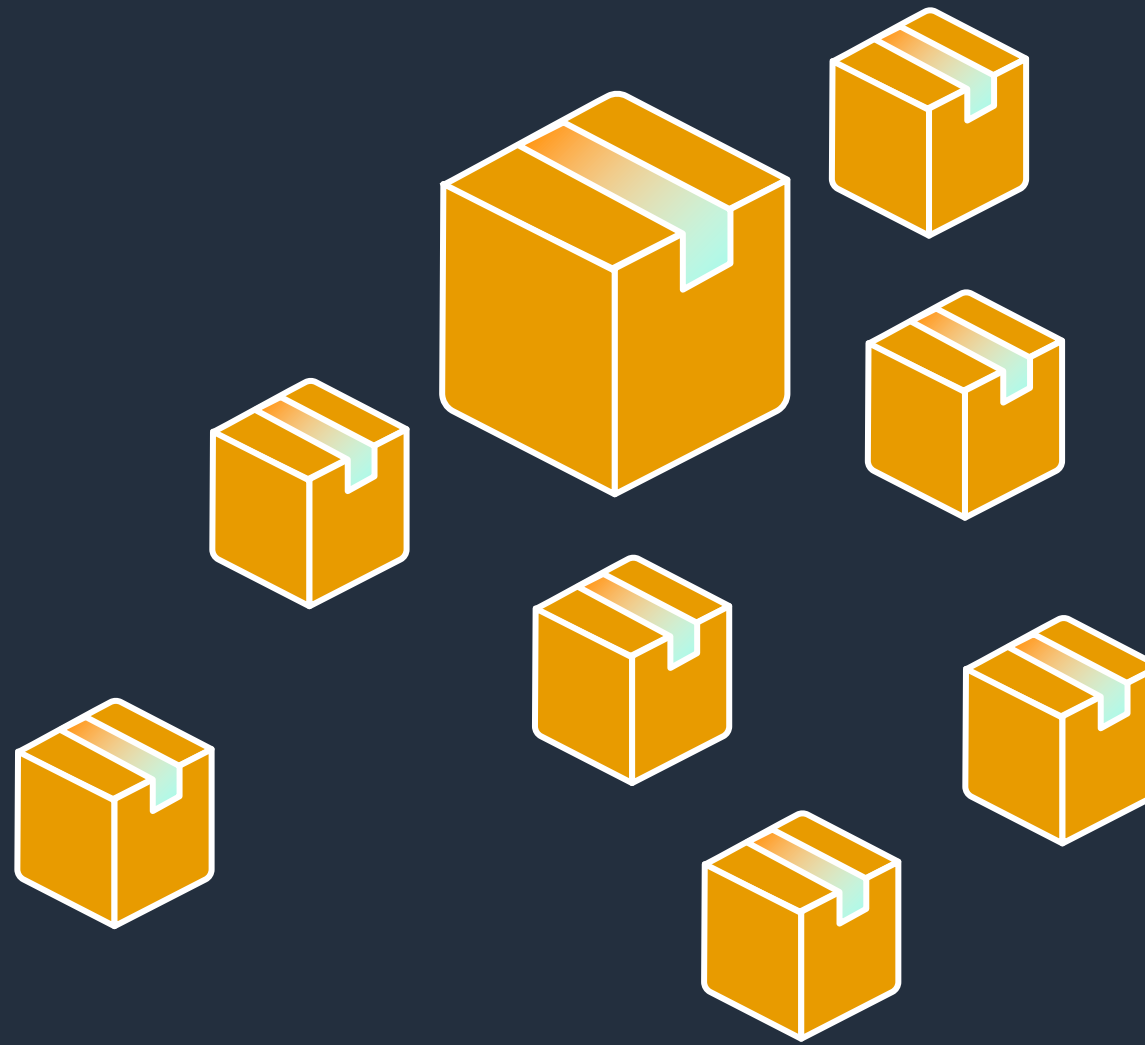
- Amazon EKS and Amazon ECR are multi-architecture friendly
- Managed EKS Addons are supported on Graviton nodes
- Labels are applied automatically to identify worker nodes ARCH and OS
  - Kubernetes.io/arch -> amd64, arm64
  - Kubernetes.io/os -> Linux, Windows
- Provisioners can be configured for a mix of Graviton and x86
  - Container runtimes automatically pull the correct image

# Key Takeaways

- Karpenter is compatible with native k8s scheduling
- Karpenter offers compute flexibility and cost optimization
  - Schedule pods to EC2 Spot Instances to optimize cost
  - Mix x86 and Graviton instances for different workloads
- Use provisioners to ensure you are scaling using best practices
  - Default provisioner with diverse instance types and availability zones
  - Additional provisioners for specific compute constraints
  - Control scheduling of application pods with node selectors, topologySpreadConstraints, taints and tolerations

# Karpenter what’s next

Learn more at: github.com/aws/karpenter



# Thank you!

Yael Grossman

yaelgr@amazon.com
linkedin.com/in/yael-grossman
aws-experience.com/emea/tel-aviv/meet-aws-expert?e=yael-grossman