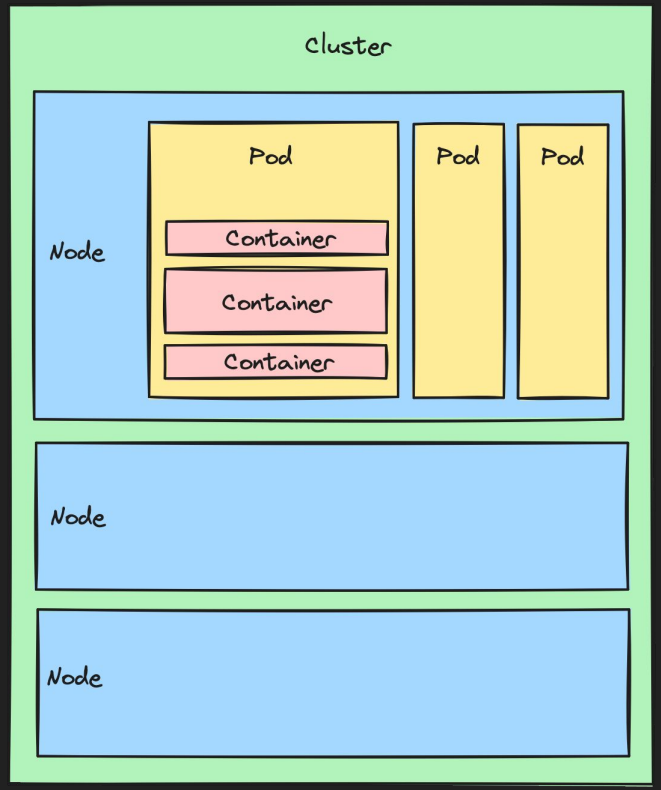
Optimizing K8s container resource requests*

* for stateless services with steady traffic

Some K8s Knowledge

- A ***container*** is a convenient way to bundle and run an application
- A ***pod*** is a logical group of containers that must run together
  - Often sets of containers must run together
  - Okay if a pod only has one container
- A ***node*** is a server that K8s runs containers on
  - Our nodes are EC2 instances with 16 CPUs and 62 Gi of memory
- A ***cluster*** is a pool of nodes used by K8s



Goals

- Ensure our services get enough resources to run well
- Minimize cost

Ensuring resources

Resource requests

- K8s ensures containers will have requested resources
- Resources above request are not ensured

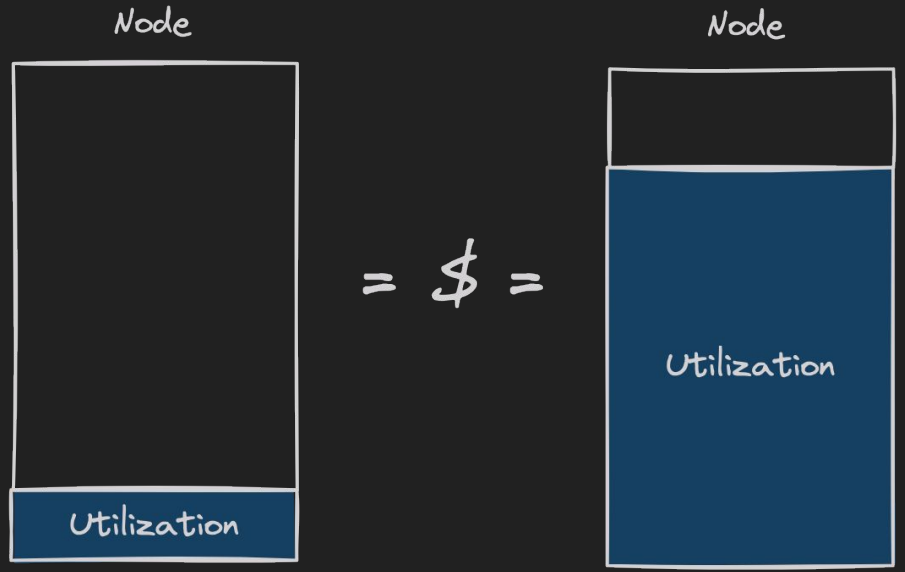
```
resources:
  requests:
    cpu: 650m
    memory: 1000Mi
```

Set resource requests to cover peak resource usage

Minimizing cost

What do we pay for?

- We pay for nodes, not containers
- We pay for each node regardless of resource utilization



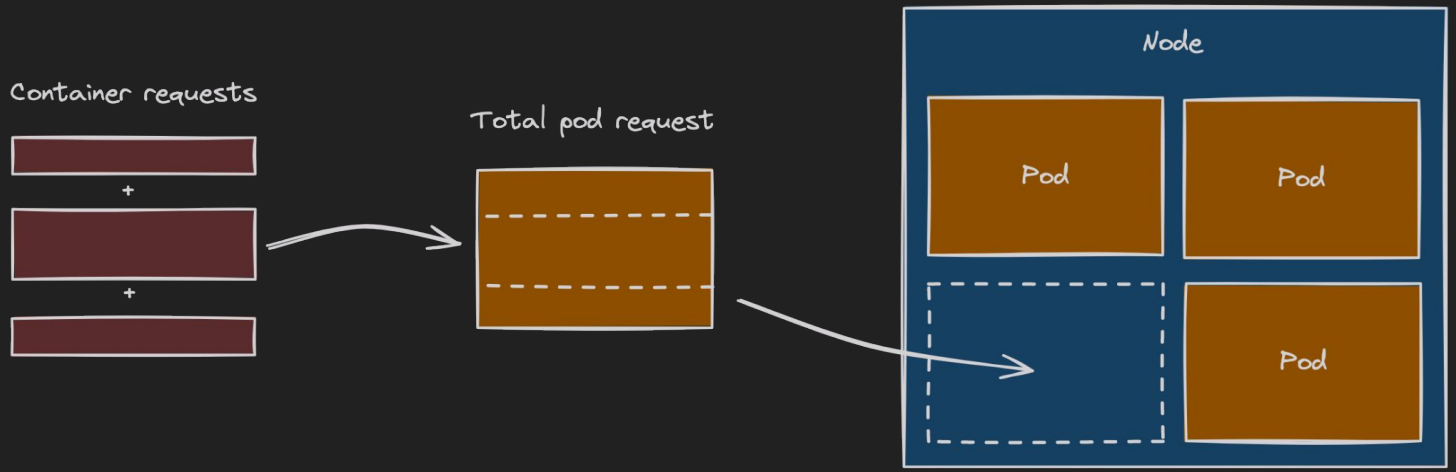
Where do nodes come from?

Node auto scaling

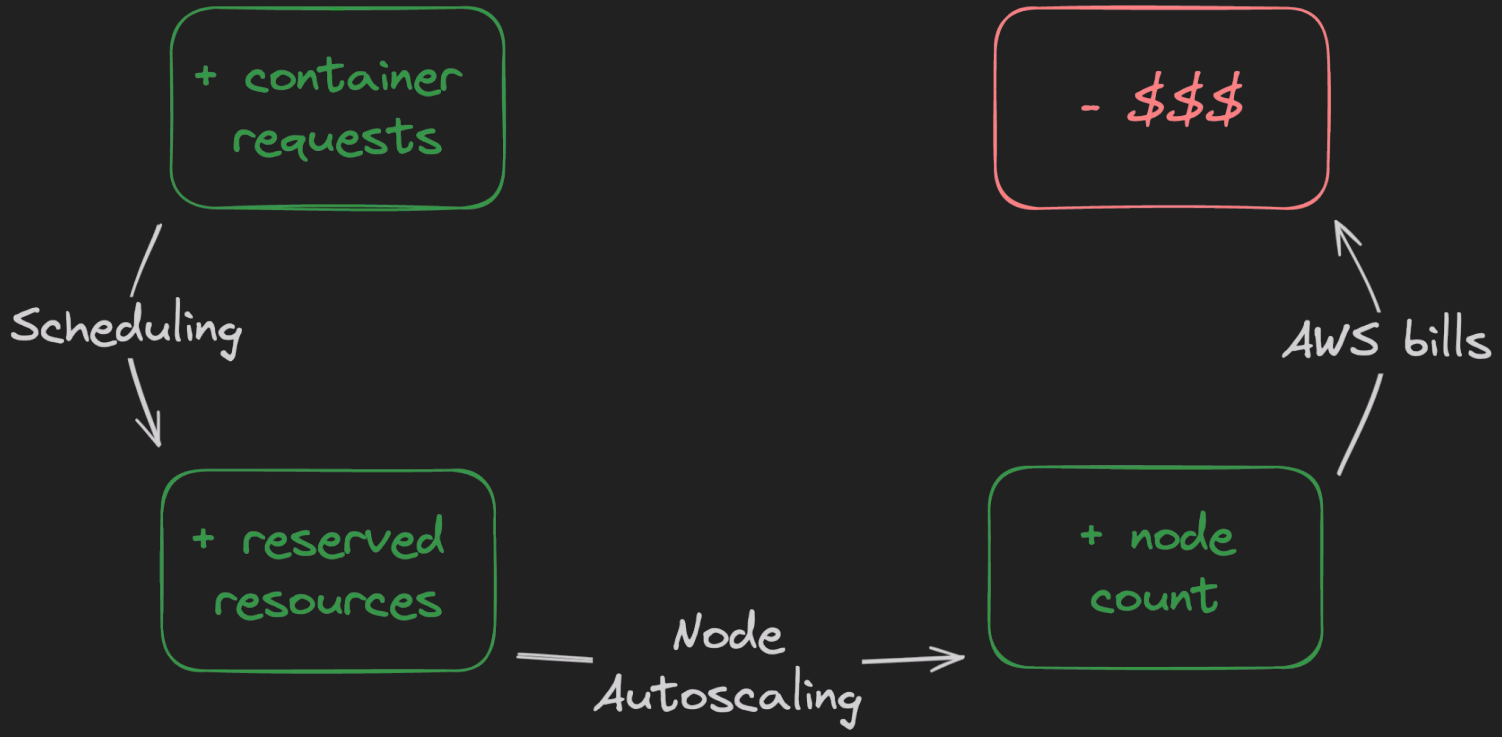
- Nodes are created when there’s no more space for pods on existing nodes
- Uses EC2 Auto Scaling Groups

Finding space on nodes

- The process of finding space to run a pod on is called ***scheduling***
- K8s sums container requests for the pod
- K8s finds a node with enough unrequested resources to fit the sum



Requests indirectly determine cost

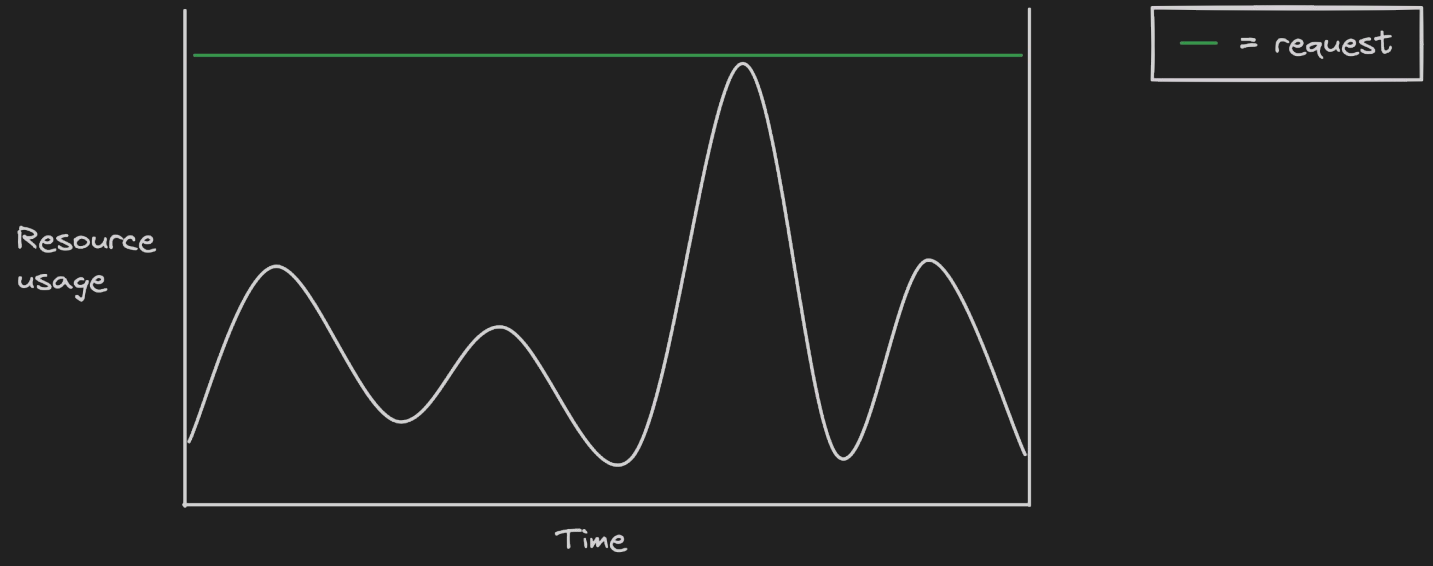


Minimize our containers’ resource requests

Finding balance

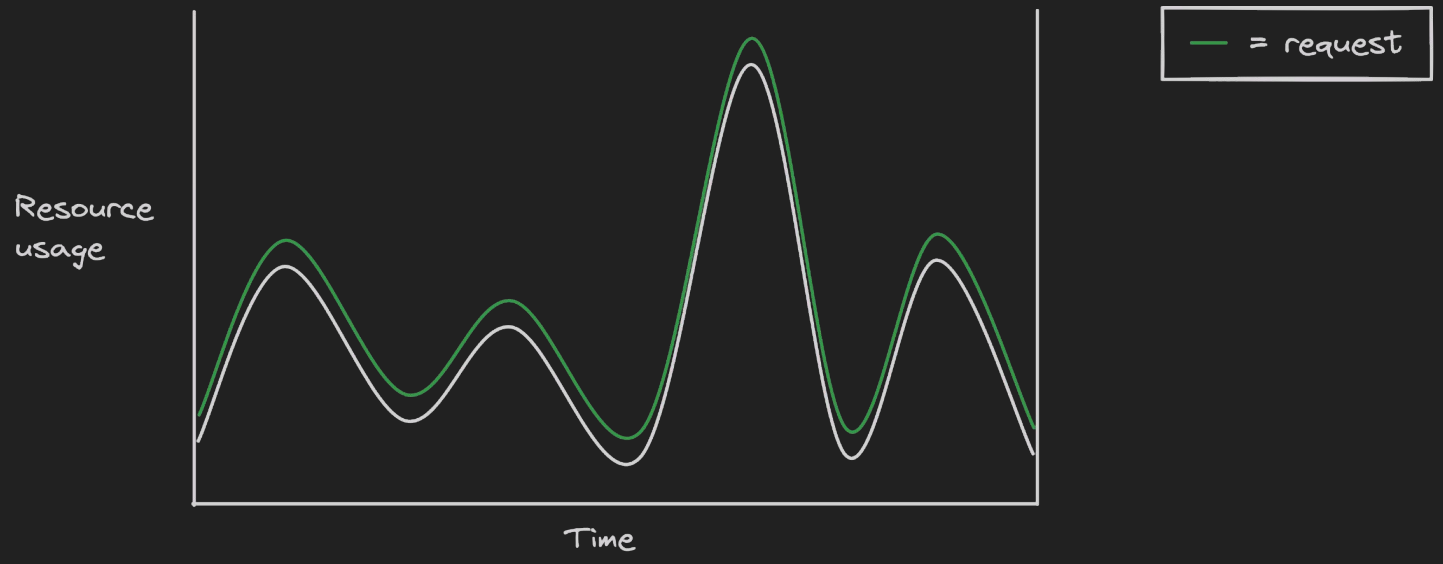
Naive approach

- Set requests to just cover peak resource usage
- Isn’t that super wasteful?
- Yes, that’s a healthy response!



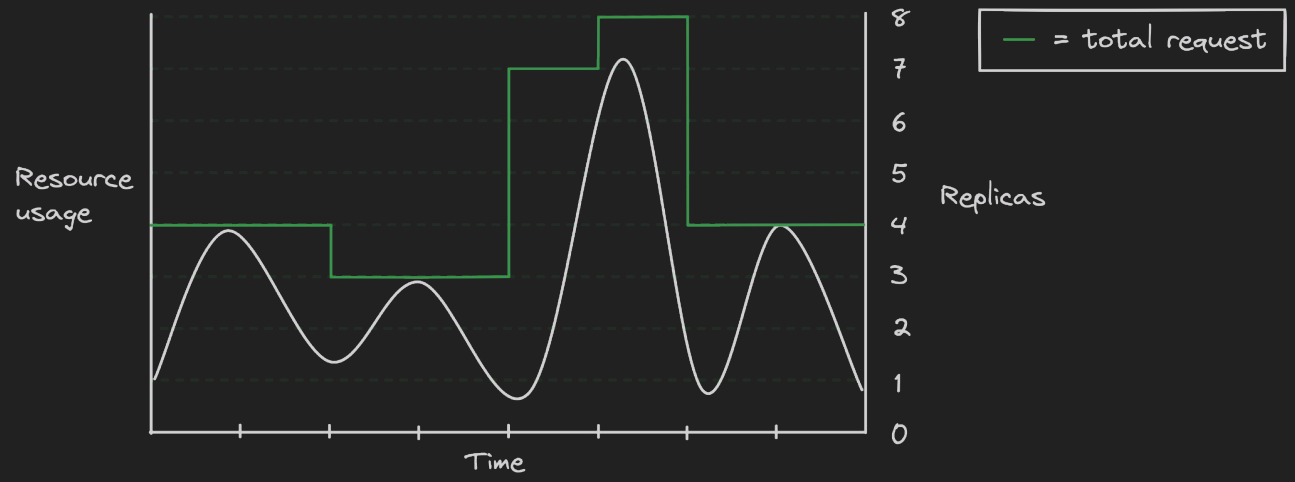
Perfect approach

- Set requests to just cover resource usage at all times
- Requests can’t be changed in-place, requires scheduling new pods
- Doesn’t work out quite right



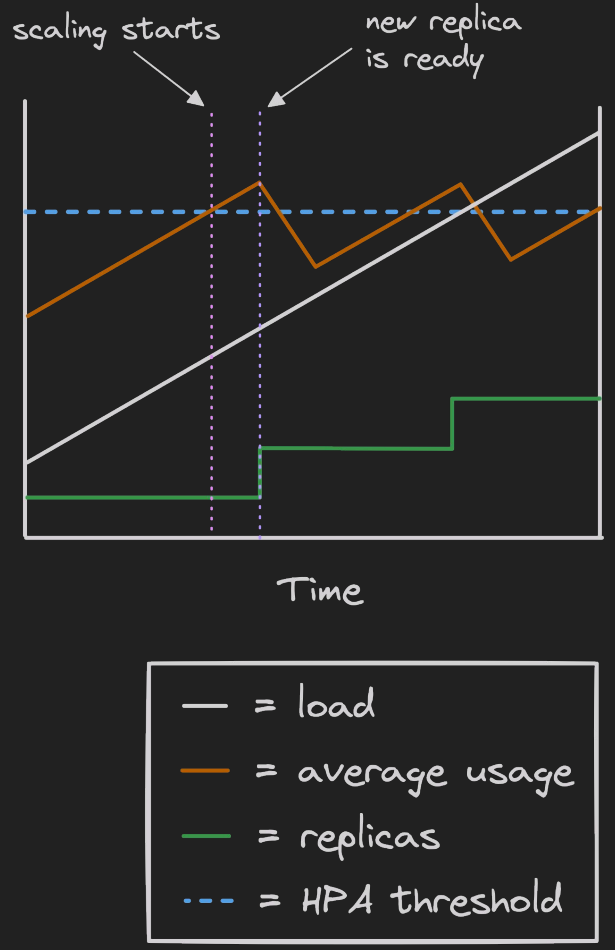
Horizontal scaling

- Dynamically sets replica count to handle load
- Static pod request but dynamic total request
- More efficient, though chunky
- Only works for stateless services



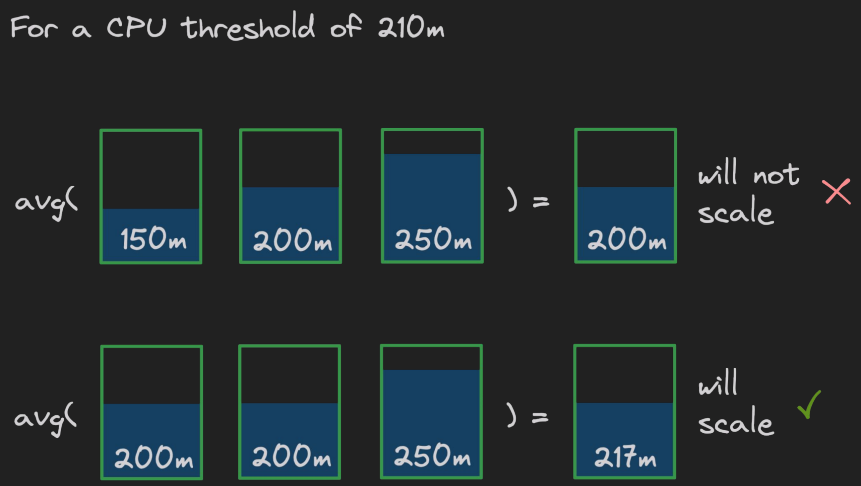
Horizontal Pod Autoscaler (HPA)

- We can't manually keep up with updating replica count to match load
- HPA handles horizontal scaling for us
- Sets replica count to keep average pod resource usage near the ***threshold***
- KEDA works the same way



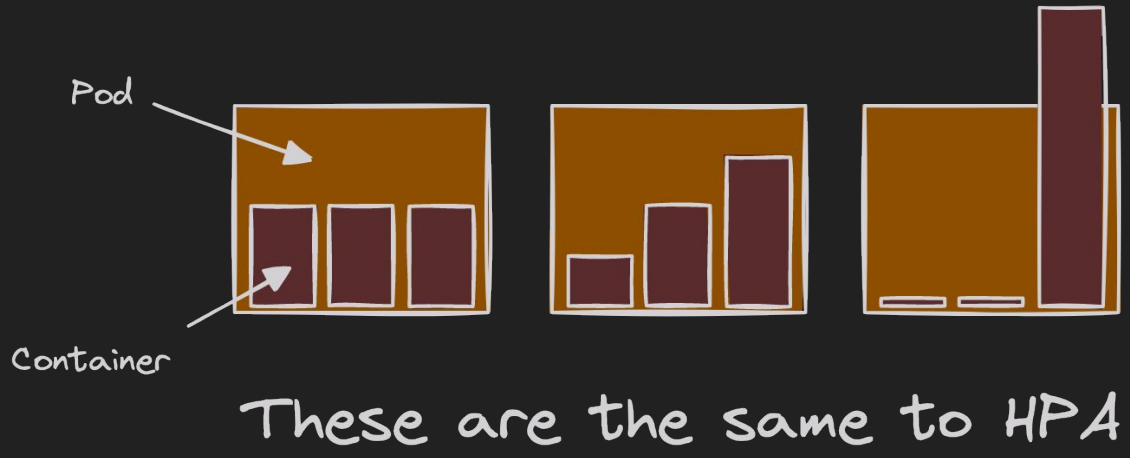
Setting a threshold

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
spec:
  metrics:
    - type: Resource
      resource:
        name: memory
        target:
          type: AverageValue
          averageUtilization: 700Mi
```



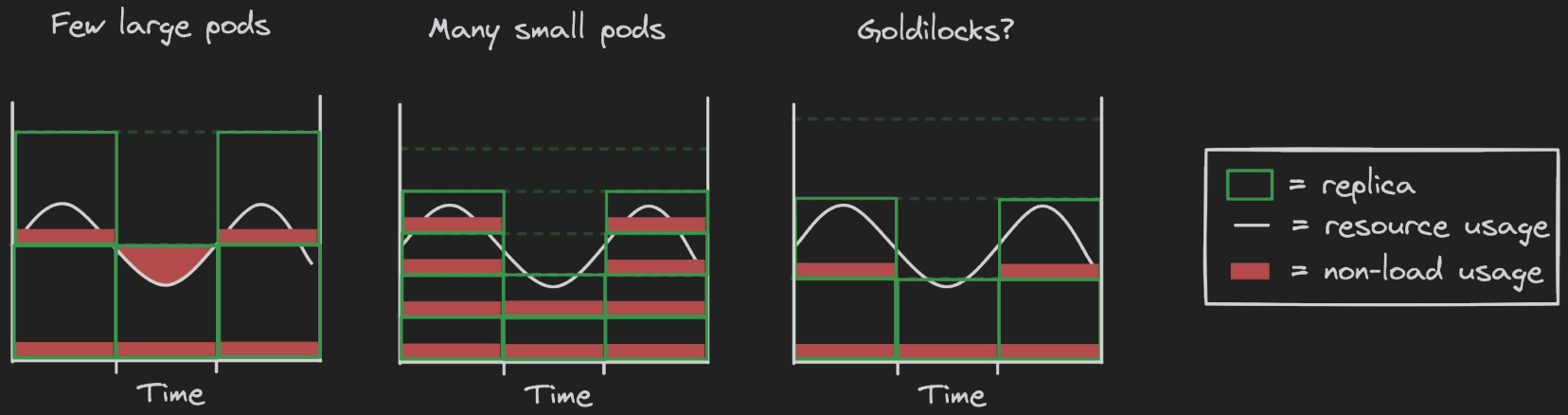
HPA observes pods, not containers

- HPA looks at averages of total pod requests and usages
- HPA does not have any notion of the containers inside pods
- We must assume usage is predictable
- Scaling on containers is coming in K8s v1.27



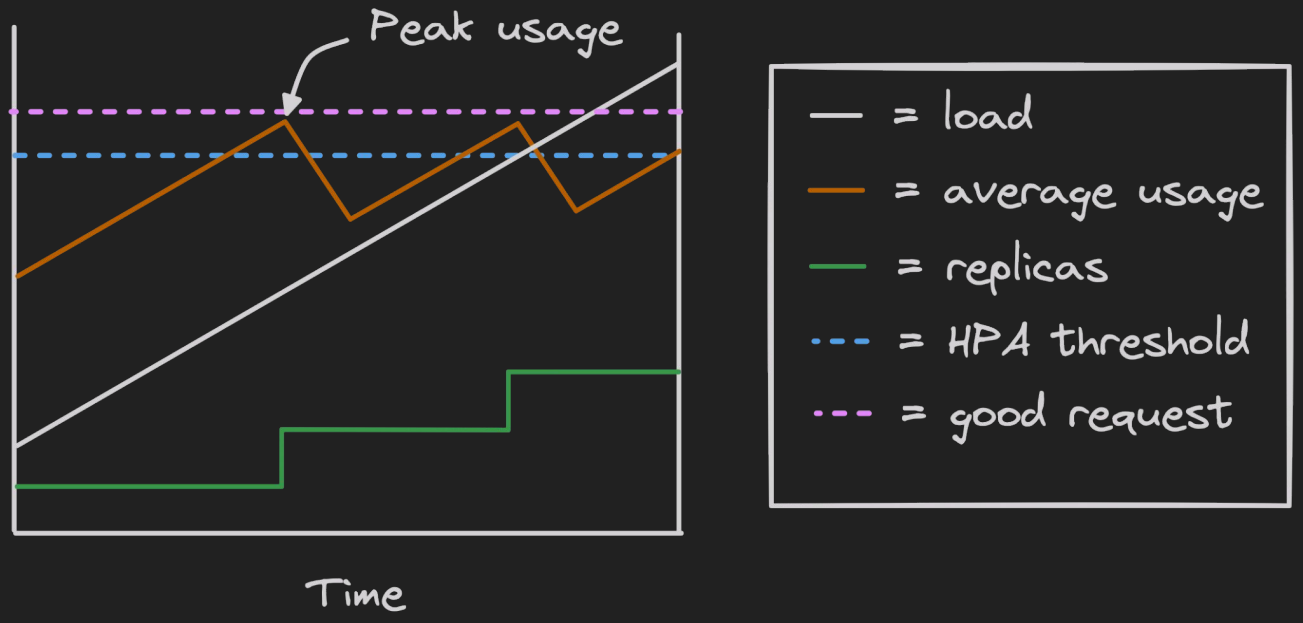
Choosing a threshold

- Using few large containers is wasteful when traffic is low
- Using many small containers compounds overhead
- Set threshold to cover minimum load as a balance
- Probably could calculate overhead, traffic, and resource efficiency



Choosing a request

- Peak usage now occurs during scaling rather than max traffic
- Request should cover peak usage during scaling



The Procedure

Finding resource usage at min load

1. Look at historical data to find min load
    - AppD is a good source
2. Create load test to replicate min load
3. Prepare for load test
    - Fix replica count to 1: we want to see resource usage for just one replica
    - Set high resource requests: we don’t want throttling/OOM killing to interfere
4. Observe resource usage

Finding peak usage during scaling

1. Look at historical data to find sharpest ramp in load
2. Create load test to replicate sharpest ramp in load
3. Prepare for load test
    - Set resource requests high: we don’t want throttling/OOM killing to interfere
    - Set HPA max replicas high: we don’t want to run into it
    - Set HPA thresholds to resource usage at min load
4. Observe peak resource usage during first scaling event

Configuration

1. Set resource requests to peak usage during scaling
2. Set HPA thresholds to usage at min load
3. Set HPA min replica above 1 to avoid single point of failure (e.g. 2 or 3)

Other considerations

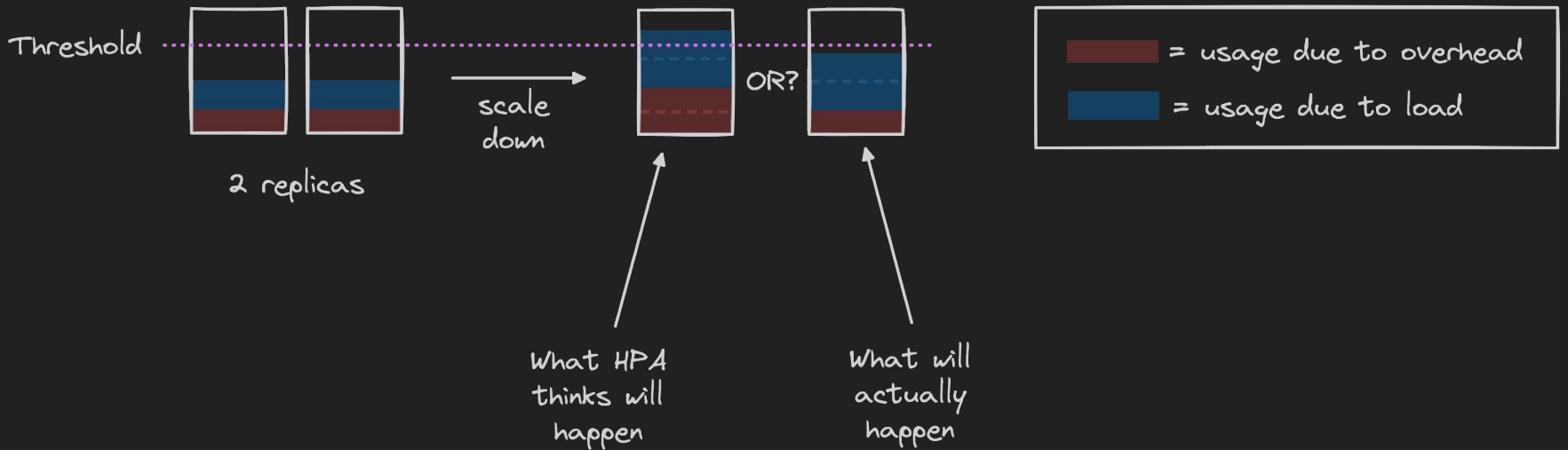
Requests must cover startup usage

- Some containers have high startup usage
- Requests must cover this, regardless of usage-based values
- Sidecar container is a good example

Sidecar Container	CPU (m)	memory (Mi)
Idle usage	~5	~150
Startup usage	~700	~150

Idle usage inhibits scaling down

- High idle usage relative to request will inhibit scaling down
- Threshold must be at least double idle usage to scale all the way down
- Resources with significant idle usage are not great scaling targets



Example: MFE Registry Service

Startup usage

Killed a pod and observed usage several times

POD	NAME	CPU(cores)	MEMORY(b
mfe-registry-service-b5cc4fc4b-dmtpc	mfe-registry-service	4m	134Mi
mfe-registry-service-b5cc4fc4b-dmtpc	sidecar	5m	149Mi
mfe-registry-service-b5cc4fc4b-f4574	mfe-registry-service	303m	175Mi
mfe-registry-service-b5cc4fc4b-f4574	sidecar	684m	150Mi
mfe-registry-service-b5cc4fc4b-s2x4x	mfe-registry-service	5m	133Mi
mfe-registry-service-b5cc4fc4b-s2x4x	sidecar	2m	152Mi

Idle usage

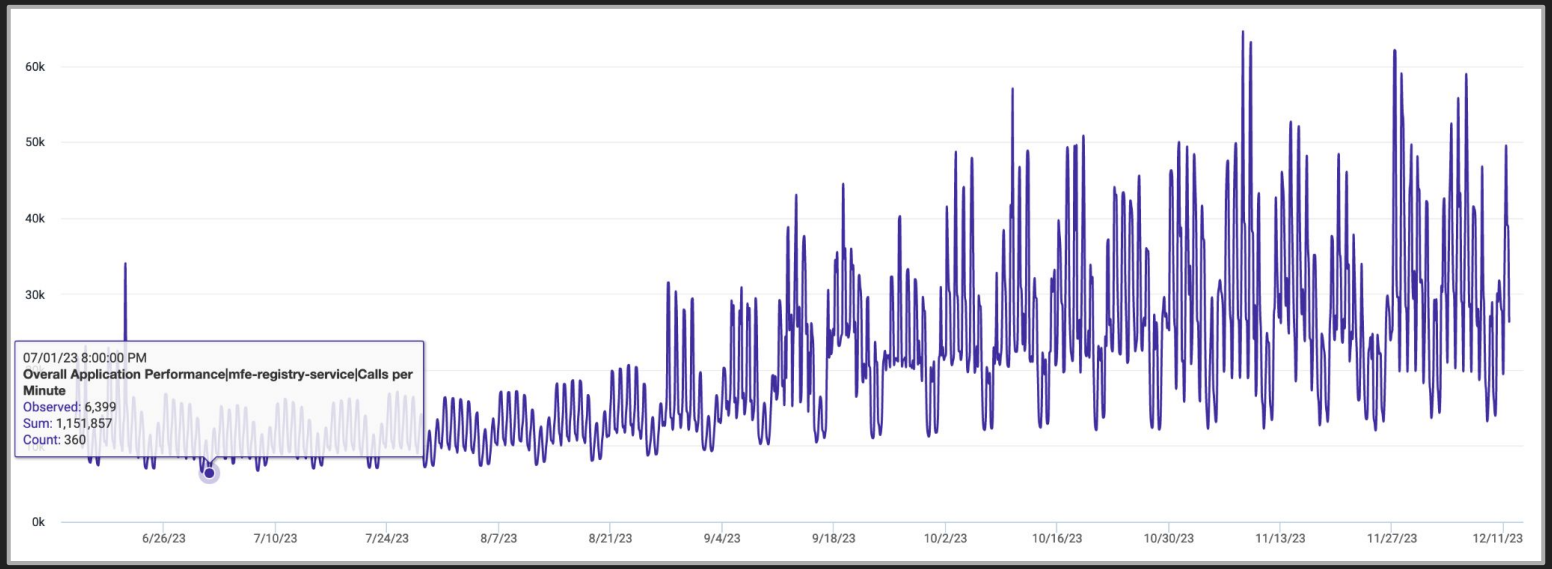
Observed usage on test with no traffic

```
POD                                      NAME                    CPU(cores)   MEMORY(bytes)
mfe-registry-service-6c544d8f9d-gx9dm    mfe-registry-service    4m           153Mi
mfe-registry-service-6c544d8f9d-gx9dm    sidecar                 2m           179Mi
mfe-registry-service-6c544d8f9d-tbccp    mfe-registry-service    3m           142Mi
mfe-registry-service-6c544d8f9d-tbccp    sidecar                 2m           164Mi
```

	CPU (m)	memory (Mi)
Pod idle usage	6	332
Min thresholds	12	664
Min thresholds with padding	20	700

Minimum load

- Looked in AppDynamics
- 6300 requests / min



Load test for minimum load

- Used K6
- Set target request rate to replicate min request rate
- Time unit is the same as AppD

```
export const options = {
  discardResponseBodies: true,
  scenarios: {
    contacts: {
      executor: "ramping-arrival-rate",
      startRate: 0,
      timeUnit: "1m",
      preAllocatedVUs: 50,
      stages: [
        { target: 6300, duration: "1m" },
        { target: 6300, duration: "5m" },
        { target: 0, duration: "1m" },
      ],
    },
  },
};
```

K8s config for min load load test

- Fix replica count at 1
  - Can be done with or without HPA
- Set requests high

```yaml
apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 1
  template:
    spec:
      containers:
      - name: mfe-registry-service
        resources:
          requests:
            cpu: 1000m
            memory: 1000Mi
```

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: mfe-registry-service
  minReplicas: 1
  maxReplicas: 1
```

Resource usage at minimum load

```
Every 1.0s: kubectl top pod -l ithaka/app=mfe-registry-service --containers

POD                                      NAME                   CPU(cores)   MEMORY(bytes)
mfe-registry-service-688dffb65d-c99jf    mfe-registry-service   303m         163Mi
mfe-registry-service-688dffb65d-c99jf    sidecar                2m           173Mi
```

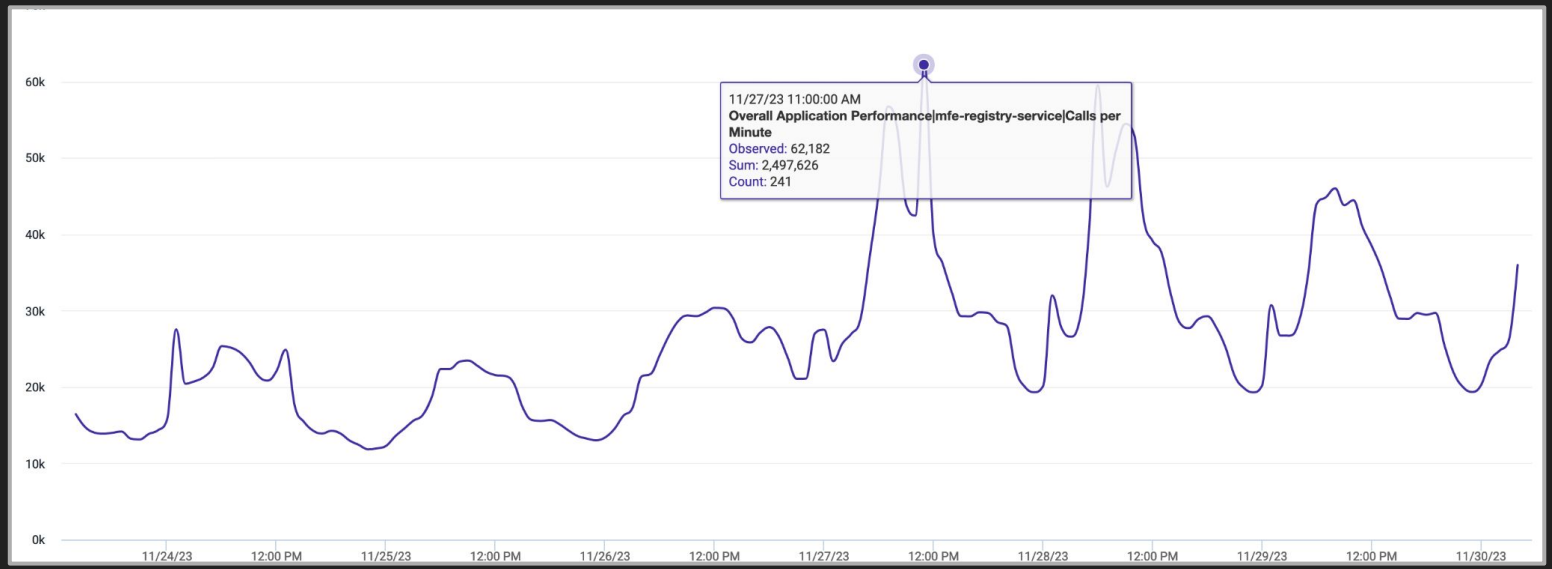
	CPU (m)	memory (Mi)
mfe-registry-service	311	167
sidecar	5	172
total	316	339
total with padding	325	350

Chose not to scale on memory

- CPU is the limiting factor
- High memory usage at idle makes it difficult to scale on
- Keep an eye on it to make sure that’s true

Maximum traffic ramp

- Looked in AppDynamics
- Sharpest increase went from 42,442 to 62,182 in 1 hour
- 334 requests per minute^2



Load test for max ramp

- Used K6
- Start just below min load
- Set target request rate to replicate the max ramp
- Time unit is the same as AppD

```
const startTargetRate = 6200; // requests per minute
const acceleration = 500; // requests per minute^2
const duration = 20; // minutes
const targetRate = startTargetRate + acceleration * duration;

export const options = {
  discardResponseBodies: true,
  scenarios: {
    contacts: {
      executor: "ramping-arrival-rate",
      startRate: 0,
      timeUnit: "1m",
      preAllocatedVUs: 150,
      stages: [
        { target: startTargetRate, duration: "1m" },
        { target: targetRate, duration: `${duration}m` },
        { target: 0, duration: "1m" },
      ],
    },
  },
};
```

K8s config for max ramp load test

- Set replica count high
- Leave high requests
- Set thresholds to usage at min load

```yaml
apiVersion: apps/v1
kind: Deployment
spec:
  template:
    spec:
      containers:
      - name: mfe-registry-service
        resources:
          requests:
            cpu: 1000m
            memory: 1000Mi
```

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
spec:
  minReplicas: 1
  maxReplicas: 20
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: AverageValue
          averageUtilization: 325m
```

Peak resource usage during scaling

```
Every 1.0s: kubectl top pod -l ithaka/app=mfe-registry-service --containers

POD                                      NAME                    CPU(cores)   MEMORY(bytes)
mfe-registry-service-6cd689f7d-hw5dw     mfe-registry-service    353m         159Mi
mfe-registry-service-6cd689f7d-hw5dw     sidecar                 2m           202Mi
```

	CPU (m)	padded CPU (Mi)	memory (Mi)
mfe-registry-service	353	400	159
sidecar	2	20	202

Final K8s config

```yaml
apiVersion: apps/v1
kind: Deployment
spec:
  template:
    spec:
      containers:
      - name: mfe-registry-service
        resources:
          requests:
            cpu: 400m
            memory: 300Mi
```

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
spec:
  minReplicas: 1
  maxReplicas: 20
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: AverageValue
          averageUtilization: 325m
```

Staying optimized with monitoring

Request and HPA optimizations don’t last forever

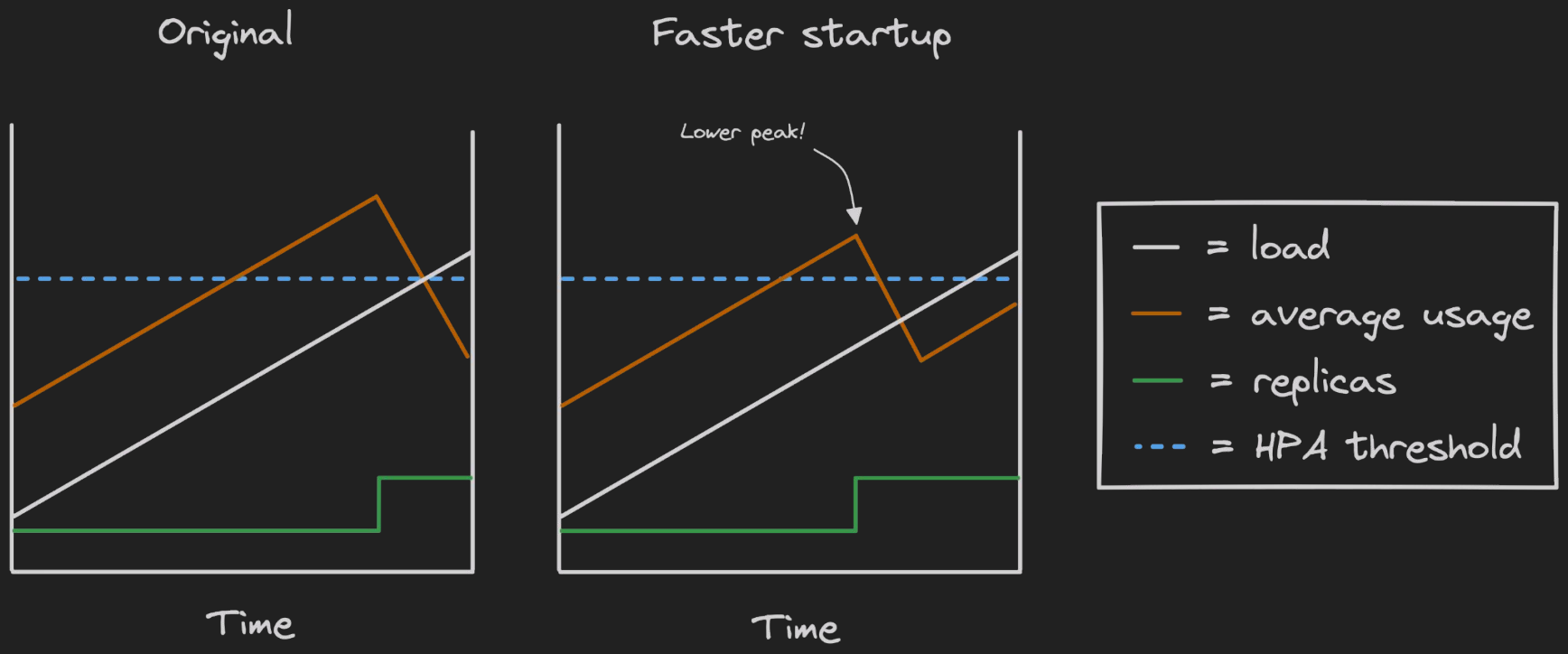
- Traffic patterns change
- App resource usage changes

Metrics can warn us when optimizations are stale

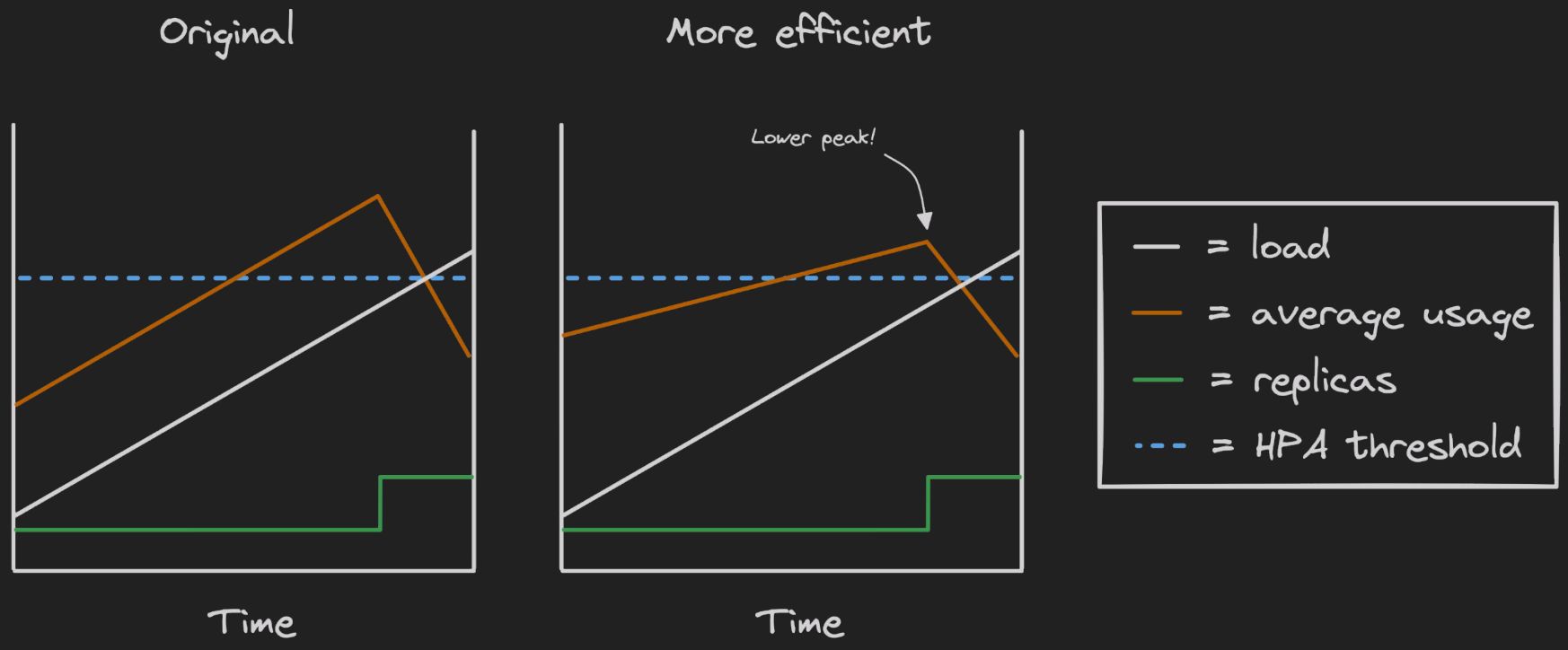
- Consistent CPU throttling > CPU request too low
- Regular OOM kills > memory request too low
- How to monitor when requests & thresholds are too high?

Effects of app optimization

Startup time optimization



Resource usage optimization



HPA redundancy

Horizontal scaling is not redundancy

- True redundancy requires a hot standby receiving no traffic
- With HPA, all replicas get traffic
- Min replica higher than necessary provides some redundancy
- Losing a pod matters less at large replica counts

Further reading

Literature on how stuff works

- Making Sense of Kubernetes CPU Requests And Limits
- GitHub issue: Clarify HPA Value vs. AverageValue

Literature on how we should use stuff

- For the love of god, stop using CPU limits on Kubernetes (updated)
- Why You Should Keep Using CPU Limits on Kubernetes
- What everyone should know about Kubernetes memory limits, OOMKilled pods, and pizza parties

Slack threads

- Thread with CORE about node autoscaling
- Thread with CORE about CPU limits