stripe

Autoscaling at Scale with KEDA

Ali Aqel
Senior Software Engineer, Stripe

KubeCon 2026

The Basics

Who am I?

4.5 years

on Stripe's Kubernetes Team

KEDA Core Contributor/Reviewer

Michigan State University

Go Green!



The Basics

What is Stripe?

Financial services platform

Helps all types of businesses accept payments, build flexible billing models, and manage money movement.

$1.9 Trillion

total volume in 2025

The Basics

What is KEDA?

Kubernetes Event Driven Autoscaling (KEDA) is a CNCF project extending the Kubernetes Horizontal Pod Autoscaler with external metrics sources, called scalers, and features like Scale to Zero.

External metrics

Prometheus, CPU, memory, custom scalers

KEDA

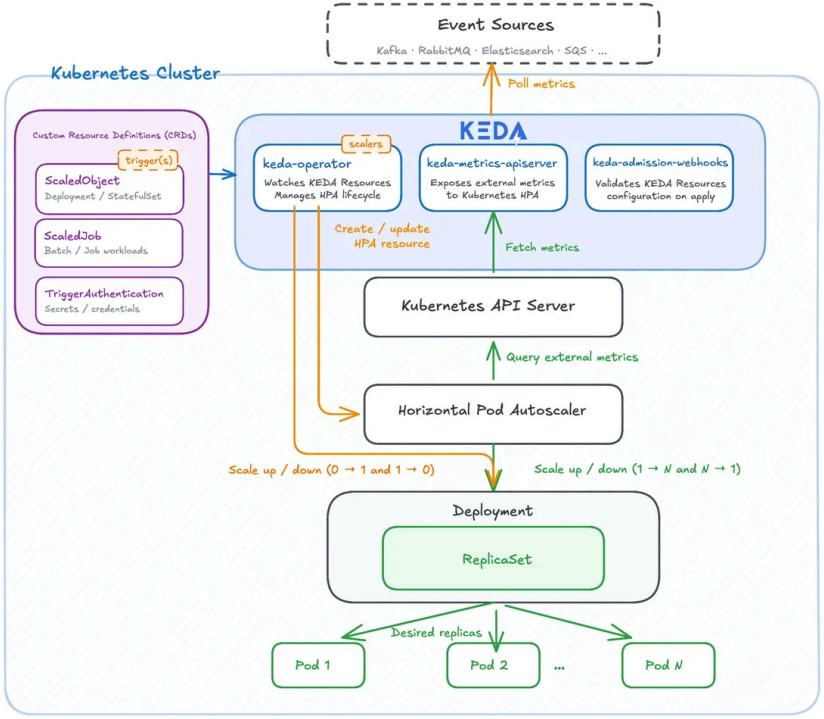
Events become scaling signals

HPA

Decides the replica count

Pods

Scale out, in, or to zero



This talk: how we integrate and manage KEDA at scale.

What we are autoscaling

6	90+	84k+	180k+	8600+
Regions	Self-Managed Clusters	Concurrent K8s Deployments	Application Pods	Unique Services

Stripe commits to customers **99.999%** SLO

Requirements

Autoscaling at Stripe's scale must be

01

Safe

Autoscaling should not cause a negative impact on uptime.

02

Stable

Rely on autoscaling infrastructure and respond quickly to issues.

03

Scalable

Handle existing and new services with a small team of infrastructure engineers.

04

Simple

No training product engineers in Kubernetes internals.

KEDA + HPA did not initially meet our production needs

01 **Poor rollout safety** — Not Safe

One color can scale down while traffic shifts, or the other can be undersized.

02 **Limited Incident Response tooling** — Not Safe

Fleet-wide pause operations are cumbersome.

03 **Limited visibility** — Not Stable

No confidence signal that the HPA is doing what is expected.

04 **Undersized** — Not Scalable

Defaults target small clusters.

05 **Configuration requires K8s know-how** — Not Simple

YAML, copied trigger configuration, and room for mistakes.

Protect your users and services

How we do it Deployment, Configuration, and Runtime Protections

01 **Scale is matched during deployments**

Blue/green colors stay the same size.

02 **Guardrails around configuration**

Scale to 0 only for allowed services; critical paths keep min_replicas ≥ 3; validating webhooks reject unsafe behavior.



Ensure AZ balance on scale up and down

We built a controller that uses Kubernetes Pod Deletion cost annotations to help preserve availability-zone balance on scale downs.

Provide Break Glass Tooling

How we do it Provide UX for Operators to Quickly Pause Autoscaling

Scale-down locks

Built on KEDA's scale-down pause annotation.

Full autoscaling locks

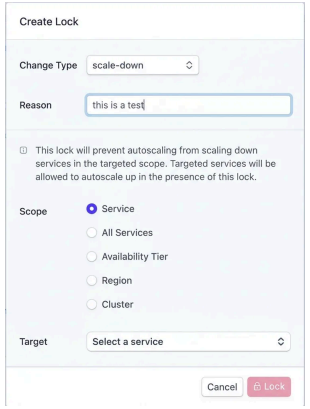
Built on KEDA's paused annotation.

Easily discoverable

Operators can find and apply locks in our developer portal.

APPLY ACROSS SCOPES

- service
- cluster
- region
- tier
- environment
- global



Upstreamed to KEDA

We contributed the Scale-In lock to KEDA, so anyone running KEDA can pause scale-ins.

Provide Break Glass Tooling

How we do it Incident Tooling for Operators if autoscaling is degraded

Manual scale up UX

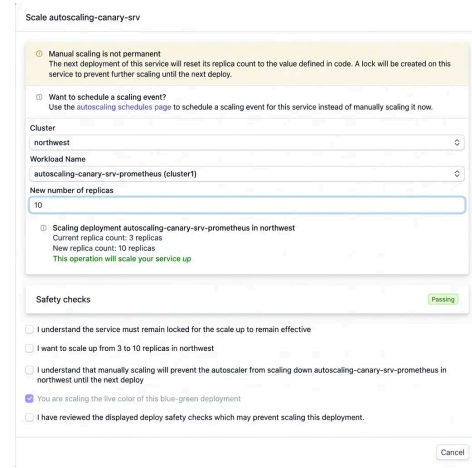
Allow service owners to manually scale their individual services

Bulk scale up CLI

Scale service cohorts together

Force Activation

Activate Scale to Zero workloads when an external scaler malfunctions



Upstreamed to KEDA

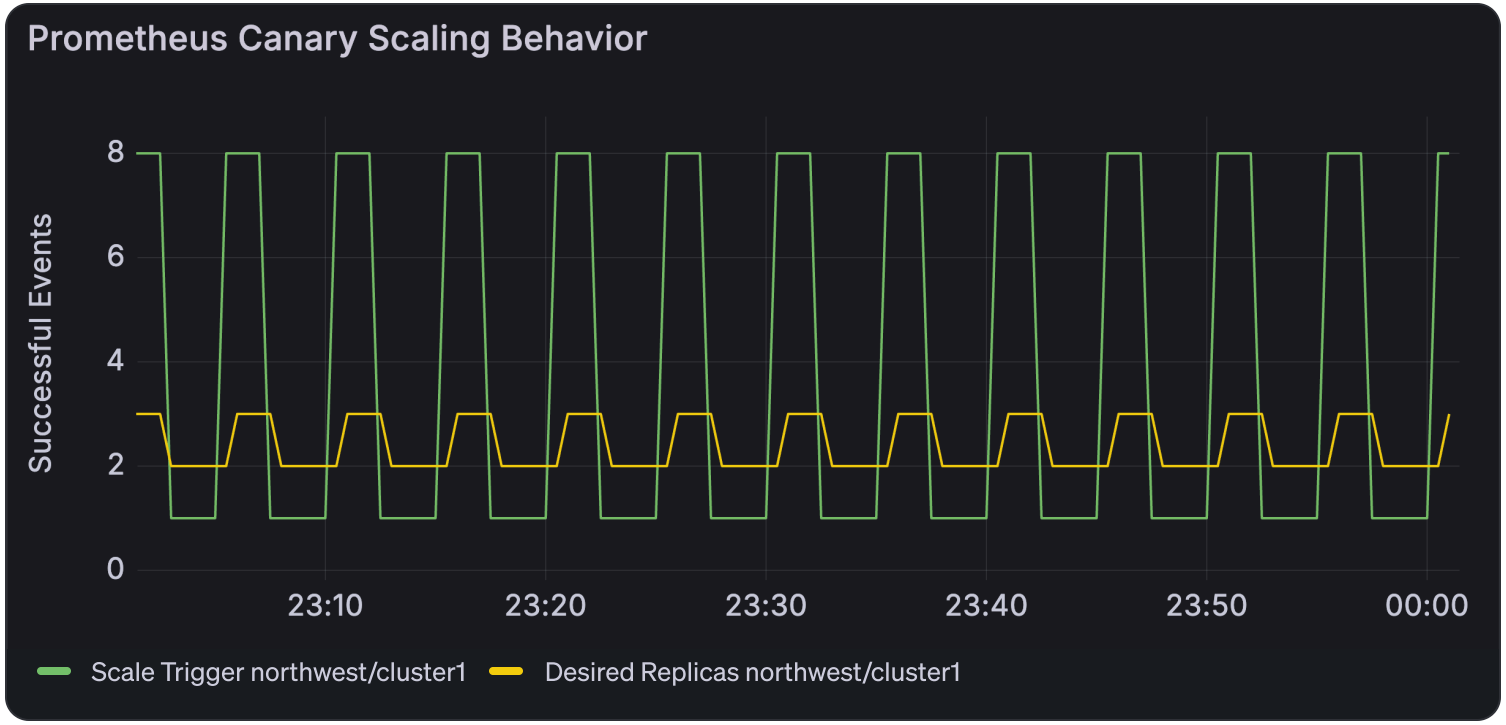
We contributed Force Activation support to KEDA, so anyone running KEDA can use it.

Know what your infrastructure is doing

How we do it Canaries to test and monitor Autoscaling infrastructure end-to-end

A fleet of canaries to monitor infrastructure

One canary for every supported trigger type; failures reveal scaler and other issues, like EC2 capacity problems or even node DNS.



Know what your infrastructure is doing

How we do it Alerts and SLOs on Autoscaling infrastructure end-to-end

01 **Monitoring and Alerts**
Pages on canary failure, system-wide success-rate alerts, and KEDA itself

02 **Service level alerts**
Let service owners know their autoscaling is not working

03 **SLO commitments to our internal customers**
End-to-end from load metric emission to the HPA decision to scale.

Upstreamed changes to KEDA

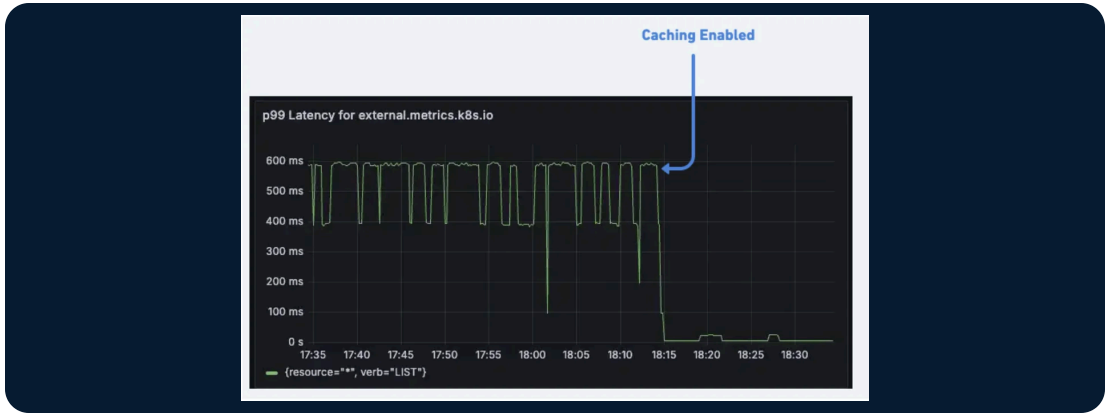
Prometheus and OpenTelemetry metrics for outbound HTTP and gRPC trigger requests, available in KEDA 2.20 and 2.21.

Tune what you can

How we do it Increase Controller Concurrency, leverage Caching

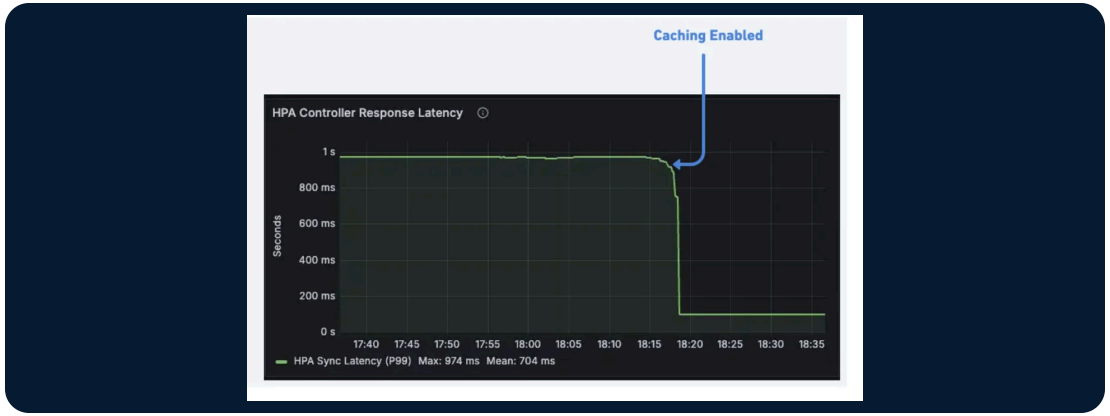
Controller tuning

Running self-hosted K8s allows us to tune not only KEDA concurrency, but the HPA concurrency and scaling loops as well



Metric caching

For scalers that support it, cached metrics cut request latency.



This lets us run ~3000 ScaledObjects per production cluster

Scale your scalers

How we do it Each cluster gets its own Prometheus instance, leverage other scaler types

Shard Prometheus

Global Prometheus Workspace
Hit QPS limits immediately as we started to scale

↓

Global Autoscaling Workspace
Kept autoscaling queries separate from alerting and Grafana traffic

↓

Isolated Cluster Workspace
Each cluster has its own autoscaling workspace

Support more than Prometheus

- CPU
- Memory
- Predictive scaling
- Scheduled scaling
- Temporal workers

Each trigger type is vetted before we support it, integrates with our internal circuit breaker system, and takes load off Prometheus.

Safe | Stable | Scalable | **Simple**

K8s should only be known by your K8s engineers

How we do it Provide templates, abstractions, and UX that hide k8s primitives from our service owners

Simple options

A few values in, Kubernetes YAML out

Templated triggers

Templates fill in the details; service owners just set thresholds

Hidden implementation

No ScaledObjects, HPA, or autoscaling internals to learn

Engineer UX

View, understand, and troubleshoot behavior

```yaml
autoscaling:
  cpu_scaling_trigger:
    utilization: 80
  java_memory_scaling_trigger:
    utilization: 85
  java_op_scaling_trigger:
    utilization: 80
  max_replicas: 15
  min_replicas: 3
```



Tying it all together

Surviving a scaler outage

01 Simple

All Temporal workers get autoscaling automatically

02 Stable

Alerting comes with it

Monitoring and alerts are set up for them

03 Stable

Service owner gets alerted

Their scaling is degraded

04 Stable

Temporal scaling canary confirms the outage

Temporal platform team is alerted

05 Safe

Temporal platform team force activates workers

All Temporal workers scaled up from 0, with a single command

06 Scalable

Workers keep scaling on CPU and Memory

Once they've been activated

07 Safe

Service owner can manually scale via UX

If their workload is critical

The Kubernetes team was not involved here.

Tying it all together

Surviving a regional capacity outage

01 Simple

Service owner adds autoscaling

A couple of lines of YAML

02 Stable

Alerting comes with it

Monitoring and alerts are set up for them

03 Stable

Service owner gets alerted

Autoscaling isn't scaling them up

04 Stable

We get alerted

Our canaries confirm a systemic outage in a region

05 Safe

We lock scale downs globally

One action from the Kubernetes team

06 Scalable

High Availability team scales up a new region

Ready for failover traffic, without autoscaling scaling it back in

07 Scalable

Traffic is failed over

To the newly scaled region

08 Safe

Stripe's users see no impact

All we did as the Kubernetes team in this scenario was lock scale downs. That's autoscaling at scale.

Wrapping Up

Key takeaways

Safe

Protect your users and services

Provide Break Glass Tooling

Stable

Know what your infrastructure is doing

Scalable

Tune what you can

Scale your scalers

Simple

K8s should only be known by your K8s engineers

Thank you

Questions?

Ali Aqel
Senior Software Engineer, Stripe

Email aliaqel@stripe.com
LinkedIn linkedin.com/in/ali-s-aqel

stripe