



Spin-Up, Test, Tear-Down

How **trivago** Runs Developer Preview Environments at Scale

Armin Aminian

SRE

trivago

Jan Wozniak

KEDA Maintainer

Kedify



KubeCon 2026 • Amsterdam





A travel **metasearch engine** that compares accommodation prices across
OTA (Online Travel Agency)

Expedia • Booking.com • Hotels.com • and many more

"Hotel? trivago." - You've heard this 10,000 times 🎵



Search

Compare prices



Save

Find best deals



Book

Multiple Platforms

trivago's Infrastructure

100+

Microservices



deployed across



Europe



North America



Asia



running on



POWERED BY

Google Kubernetes Engine



Search Infrastructure

(a.k.a. "How we find you that €49/night hotel in Barcelona")



What is Preview Environment?

Let's follow **Alex's** journey...

(Spoiler: It involves "works on my machine" 🙄)



Meet Alex

A developer working on the booking service

(Coffee-powered, Kubernetes-curious)

Alex's Challenge

Alex needs to add a **new feature** to the booking service.



Search Service

Booking depends on this



Booking Service

Alex's service ★



Payment Service

Booking depends on this

Booking service depends on Search & Payment - needs them all working together

(It's microservices all the way down 🐜)

It Works Locally! ✨



✓ Tests Pass

Everything looks perfect on Alex's machine

"It works on my machine!" - Famous last words 🤪



But will it work in **production**?

With all the real dependencies and configurations?

Narrator: "It didn't." 📖

What Doesn't Work 🥵

💻 Traditional Approaches

❌ Test locally?

Environment differences cause issues

"Works on my machine!" 🧑‍💻

❌ YOLO to production?

🔥 Bold strategy, Cotton

👥 At Scale (200+ devs)

❌ Staging Queue

100+ PRs waiting for one environment

"Free in... next sprint 📅"

❌ Unreliable Testing

Someone else's bug breaks your test

"It worked yesterday!" 🧑‍💻

❌ Blame Game

No isolation = finger pointing

"Whose changes broke it?" 💬

Alex Opens a Pull Request



github.com / trivago / booking-service / pull / 1247



feat: Add multi-currency support to booking

Open

#1247

alex wants to merge into `main` from `feat/multi-currency`

booking-service/src/currency.go · +47 files changed

+ Modified booking service logic

+ Updated API contracts

+ Added currency conversion

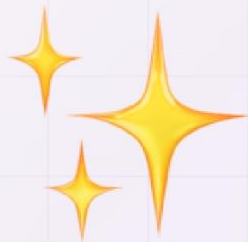
// 47 files changed, what could go wrong? 🤔



CI Pipeline running...



Time for coffee



Magic Happens!

A preview environment gets automatically deployed

`pr-1247.booking.internal.trv.cloud`

What Gets Deployed?

Production Environment



Preview Environment (PR #1247)



Only Alex's service is deployed. Dependencies use existing staging services!

Now Alex Can Test! 🎉

✓ Run Integration Tests & Catch Issues Early

Test against real dependencies - find bugs before production

✓ Share with QA Team

Send them a URL: `pr-1247.booking.internal.trv.cloud`

(No more "can you deploy to my machine?")

✓ Verify with Stakeholders

Product managers can test the feature before it goes to production

(They can finally see what "in progress" actually looks like)



Alex merges with confidence

Knowing the change won't break anything in production because it was already tested in an environment that mirrors production.

"Ship it!" - Alex, probably 



But Wait...

Why are preview environments **so important** for us?

*(Spoiler: We want every engineer to ship features with **confidence**)*

The Scale Challenge

200+

Engineers

Working daily

100+

Active PRs

Frontend repo alone

100+

Backend PRs

Across microservices



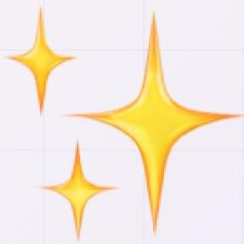
200+

Preview Environments

Active at any given time

(Our cluster is very popular 🥳)

🎯 So every engineer can ship features with confidence



The Solution: Preview Environments at Scale

Give every developer their own isolated environment to test changes and ship with **confidence**

No blocking • No conflicts • No drama 🤖

Our Starting Point



Frontend

Google Cloud Run

Preview ✓ Staging ✓ Production ✓



Backends

Kubernetes (GKE)

Staging ✓ Production ✓ **No Previews ✗**

What is Cloud Run?

Google's serverless container platform — deploy containers without managing infrastructure. **Scales automatically**, including down to zero when idle.

Cloud Run: The Good & The Bad



Pay Per Request

Only charged when handling
traffic

No traffic = No cost



Higher Cost

More expensive than K8S at
scale

Production workloads add up



Split Architecture

Monitoring · Logging ·
Deployment

All different for each platform

600+ Frontend Previews: <€100/month

Great for previews, but we needed a unified solution



The Strategy

Goal: **Unify on Kubernetes** — but how to keep cheap previews?



Solve Preview Environments for Backends First

If we crack this on K8S → frontend migration becomes easy

One solution unlocks both problems

But Kubernetes Doesn't Scale to Zero ⚡

How do we run 600+ preview environments efficiently?

💰 The Cost Problem

For **frontend alone**, running 24/7:

600+

Environments

1,200

Total Pods

1 CPU / 1 GB

Per Pod

= More expensive than Production! 💰💰💰

🤔 But these environments are only used 2-3 hours per working day!

Running 24/7 = massive waste. Cron-based scaling? **Developers don't follow 9-5 schedules** 🌙



Kubernetes 1.36 introduces native HPA scale-to-zero — great news! **For preview environments, we need scale-to-zero based on HTTP traffic** — idle when no one is browsing, instant-on when someone opens the link.



Enter KEDA

Kubernetes Event-Driven Autoscaling

Scale-to-zero for Kubernetes! 🎉

(Yes, it's as cool as it sounds)

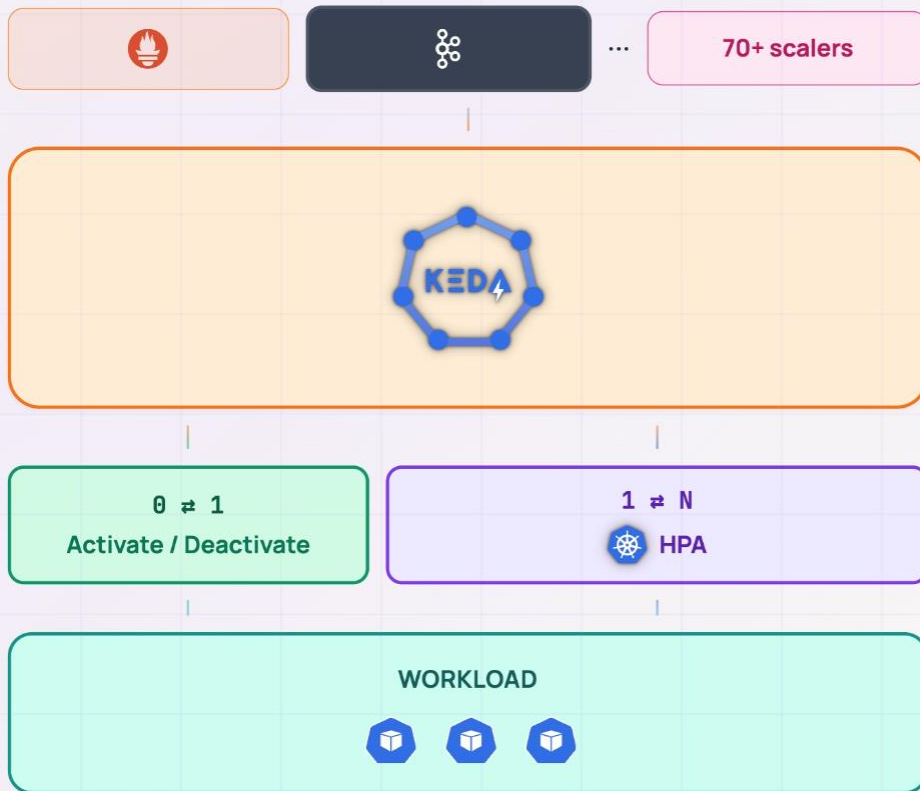
KEDA in 60 Seconds

The Project

- **CNCF Graduated** project
- Kubernetes operator extending **native autoscaling**
- **70+** built-in scalers (Kafka, Prometheus, AWS SQS, PostgreSQL, Cron, ...)
- `ScaledObject` & `ScaledJob` CRDs

The Scaling Lifecycle

- Scalers poll **event sources** and feed metrics
- **0 → 1** KEDA activates the workload
- **1 → N** Delegates to Kubernetes `HPA`
- **N → 0** Scales to zero when idle



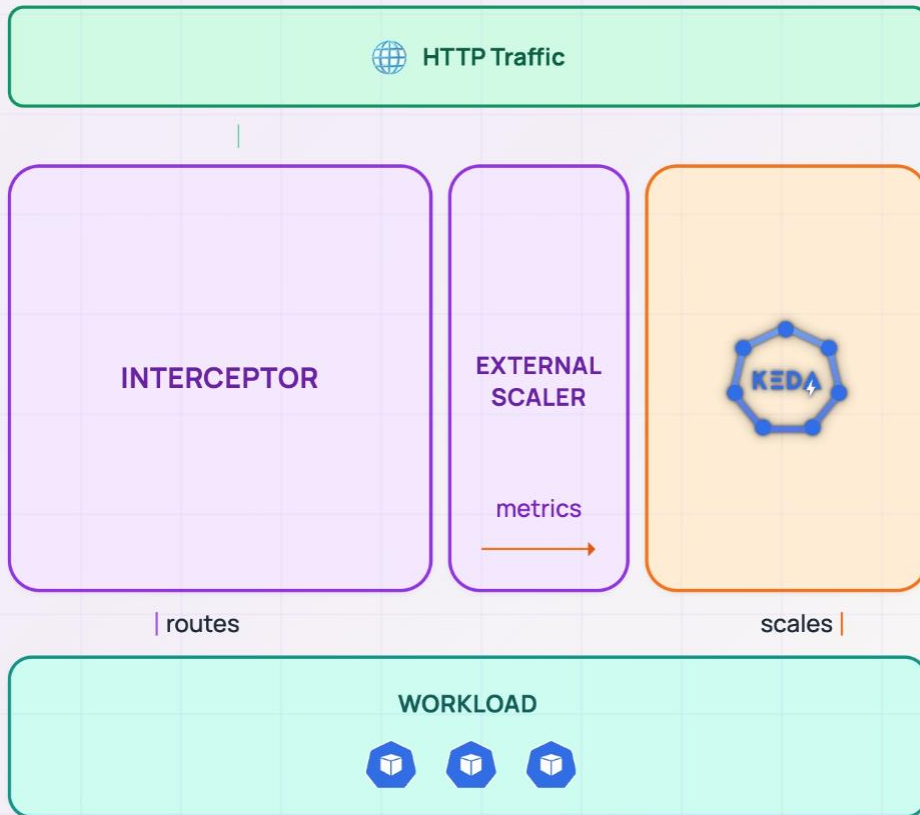
HTTP Add-on in 60 Seconds

The Project

- **Interceptor** - L7 proxy in front of workloads, holds requests while pods start
- **No loss of traffic** during cold-starts
- Reports **RPS** or number of **pending** requests
- New `HTTPScaledObject` CRD - scaling + routing

The Request Lifecycle

- **replicas = 0**: interceptor queues the request, metric rises, KEDA scales 0 → 1, request released
- **replicas > 0**: interceptor proxies directly to the app
- **no traffic**: metric goes to 0, cooldown triggers, KEDA scales N → 0



HTTP Add-on Timeline

Pioneering Years

Nov 2020

initial commit

2020 - 2022

v0.1.0

Jun 2021



Community Revival

Dec 2023

trivago PoC

2023 - 2024

v0.4.0

Jan 2023



Aug 2024

Kedify fork

v0.10.0

Feb 2025

Production Renaissance

Dec 2025

Red Hat joins

2025 - 2026

v0.12.2

Feb 2026



Why HTTP Scaling Is Different



Traditional KEDA Scaler

- Reads metrics from external sources
- Feeds decisions to the HPA
- Never touches app traffic

Scaler down → app keeps running



HTTP Add-on

- **Proxies every request**
- Generates metrics from live traffic
- Scales app and routes traffic

Proxy down → traffic stops 🤖

A **reverse proxy** and a **metrics source** - not just another scaler.

Life on the Data Path



Cold start is user-facing

Scale-from-zero means **a real user is waiting**



Request-aware routing

Host, path, WS, gRPC - at **full proxy throughput**



The scaler must scale too

Proxy ceiling becomes the app ceiling



Kedify: Traffic Autowiring

Proxy **FAILS** → fallback replicas, **traffic bypasses the interceptor**

Proxy **HEALS** → autoscaled replicas, rewires back **traffic through interceptor**



Demo Time

Let's see it in action

(a.k.a. the part where we pray to the demo gods 🙏)

The Flow: It Starts With a PR 🚀



Developer

Makes changes



Pull Request

Opens PR on GitHub



GitHub Actions

Workflow triggers

Every PR automatically triggers a CI workflow

"No manual intervention needed!"

The Flow: Build & Push to GitOps



GitHub Actions



Build Docker Image

Compile & package



Push to Registry

GCP Artifact Registry



Render K8S Manifests

Helm + Skafold



GitOps Branch

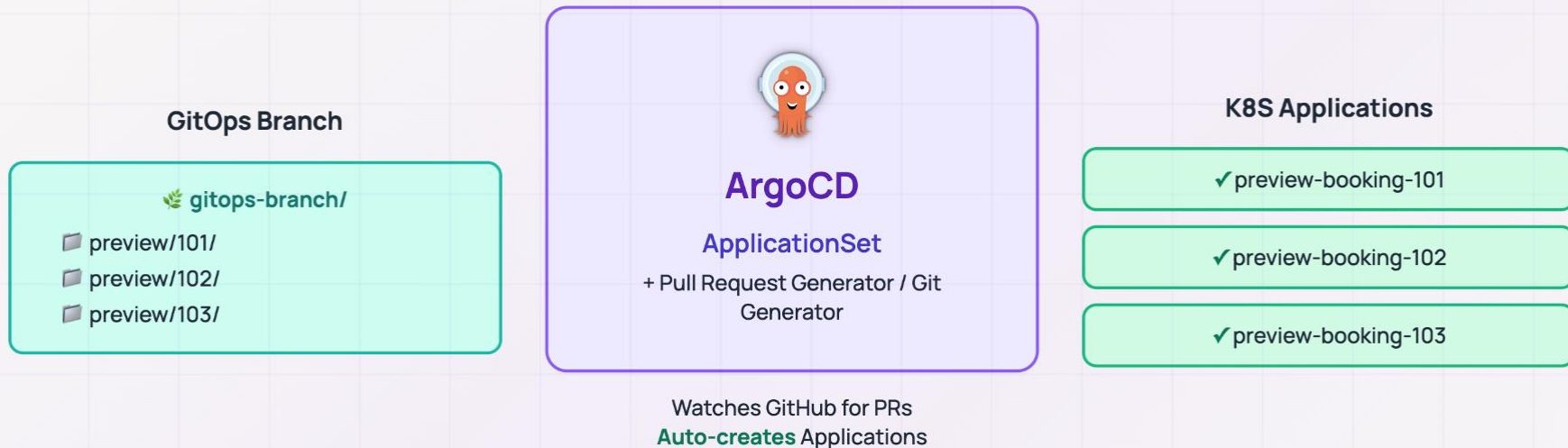
preview/123/

preview/456/

One branch, **separate directories** per PR

"Git is our source of truth"

The Flow: ArgoCD Takes Over 🎯



One ApplicationSet manages **all preview environments**

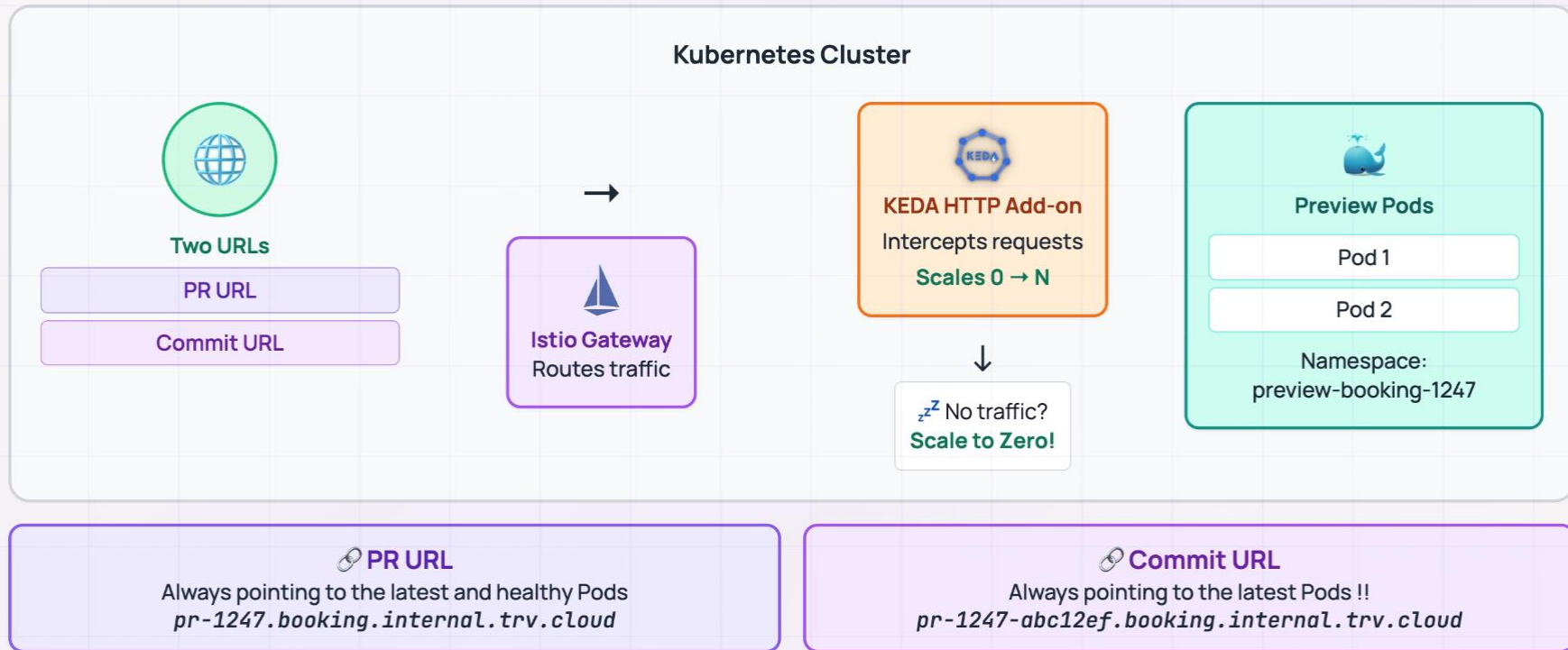
"PR merged? Application auto-deleted!" 🧹



Curious about the YAMLs? - I put together a full step-by-step guide for setting up preview environments on Kubernetes. Check it out at the end!



The Flow: Inside Kubernetes



The Complete Picture 🎨



🚀 The Magic

- **PR opens** → Environment created
- **PR merges** → Environment deleted
- **Idle Preview** → Scale to zero
- **PR URL** + **Commit URL**

🧰 The Stack

- **ArgoCD** + ApplicationSet
- **KEDA** + HTTP Add-on
- **Istio** for routing
- **Helm** for templating

💡 All automated. All GitOps. All cost-efficient.

What We Achieved 🏆



Parallel Development

200+ engineers work simultaneously

Finally, peace in the #deployments channel 🕊️



Cost Efficient

KEDA scale-to-zero saves resources



Unified Platform

Same tools for preview & production



Faster Delivery

Deploy and test immediately

Time-to-production goes brrr 🚀



The Bottom Line

Preview environments at scale = **happy developers** + **happy ops** + **happy CFO**

Extra Resources

Complete Guide to Preview
Environments on Kubernetes



bit.ly/3P59Nag

From Always-On to On-Demand:
Scaling Kafka Sinks with KEDA



bit.ly/3P1vgkA



Thank You!

Questions & Answers

Thank you for listening! 🎉

Appendix: Why Not Knative? 🤔

🎯 Knative is Great For...

Scale-to-zero and HTTP-based autoscaling

But we faced some challenges:



Inconsistent Standards

We wanted to maintain a **consistent standard** for both production and previews. Knative's `serving` model differs from our standard Kubernetes deployments.



Health-Only Traffic Routing

Traffic routes **only to healthy Pods**, making it hard to debug broken previews.
(*"Is it broken or just... healthy elsewhere?" 🤖*)



Single Ingress Limitation

We have two Istio ingresses: `public` and `private`. Knative only supports one.