Serverless on your own terms

(with Knative!)

Evan Anderson
Staff Software Engineer, VMware Tanzu

October 2020

… so I've got some code …

(and it runs on the web)

Demo

`http://github.com/evankanderson/vmware-codeconnect-2020`

What we saw:

Knative extends and interoperates with the Kubernetes ecosystem.

You can mix & match Knative with standard Kubernetes constructs.

Two different languages and frameworks:

- Python (TensorFlow)
- Java (Spring Boot)

Two different deployment methods:

- Standard Kubernetes (Deployment + Service)
- Knative (Knative Service)

Scheduled via Kubernetes

Connected via OpenAPI / Swagger

Works the same on laptop, datacenter, or edge!

What is Serverless?

(and why do I care?)

What is Serverless?

Three definitions:

A development model where developers focus on a single unit of work.

Enabled by

and/or

A platform which automatically manages compute scale-out.

Enabled by

and/or

A platform with pay-for-use properties (granular billing per action).

What is Serverless?

Automatic scaling based on demand

Automatic request-level instrumentation

A platform which automatically manages compute scale-out.

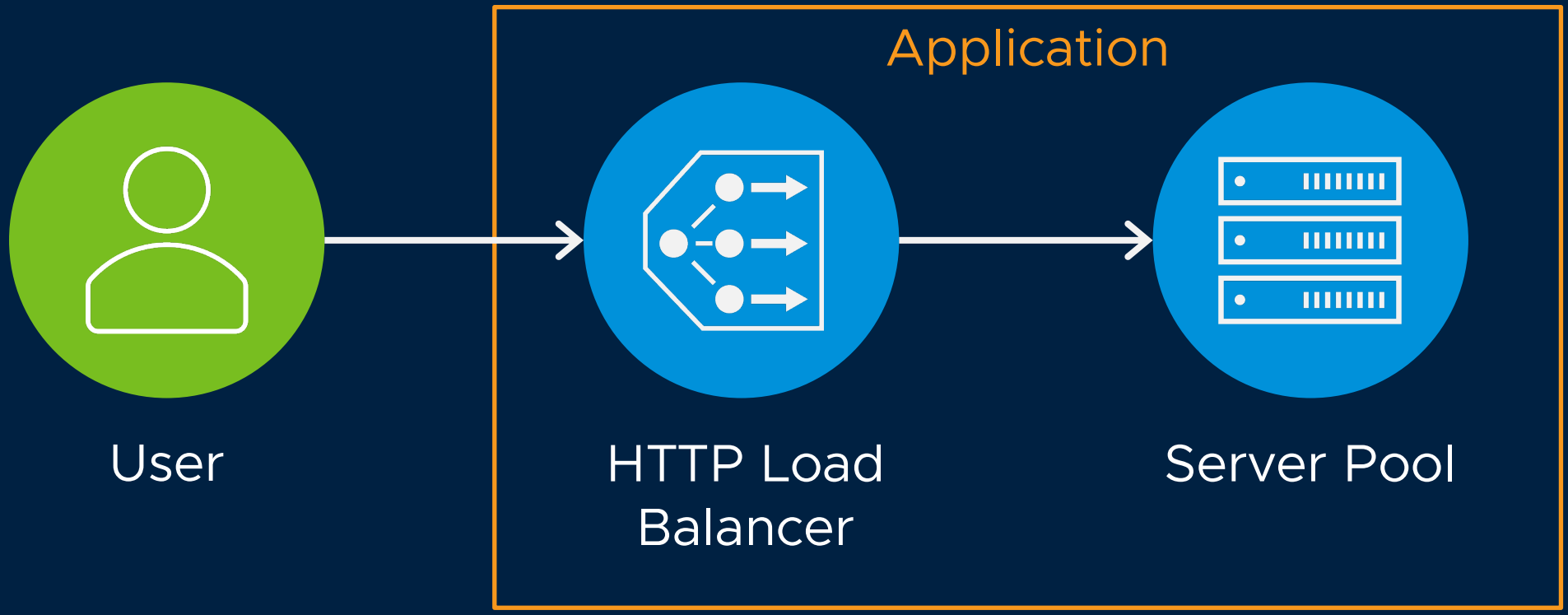
Typically *stateless* (think 12-factor apps) to support autoscaling

Knative Serving

Serverless HTTP-triggered containers

HTTP-triggered containers?

This just means a standard 12-factor type HTTP application that looks a bit like this:



Automatic Scaling

Autoscaling a 12-factor application is pretty simple, just increase the number of instances.

However:

- Often requires manual setup and linking with metrics
- Requires ability to automate instance scaling (IaaS or CaaS)
- Need somewhere to run the autoscaler (second container or VM?)
- May not bother with a small application that won't scale above 1 instance

Kubernetes-as-Infrastructure solves these problems:

- HorizontalPodAutoscaler (HPA) can attach to any resource with a `/scale` property
- The HPA controller is multi-tenant, so you only need one for the whole cluster

Automatic Scaling

No, really

Knative takes this three steps further:

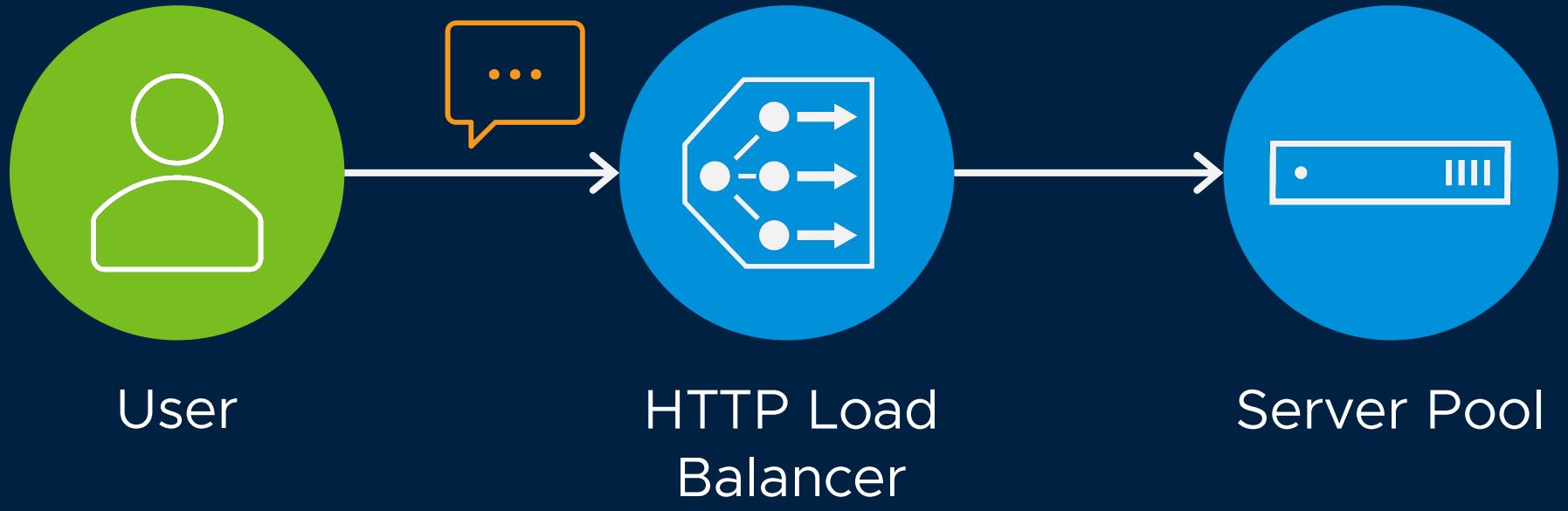
1. Every Service in Knative has autoscaling enabled by default, rather than needing to add a new HPA object to your YAML manifest
2. Knative ships with an additional request-based autoscaler that integrates with measurements of HTTP requests
3. Knative also supports *scale-to-zero*, where all the instances can be shut down when the service is idle (standard HPA needs one instance to measure current load)

Knative provides a documented set of resource annotations that can be used to tune the autoscaler if needed, but the default settings should work out of the box. The autoscaling team is very interested in making the default settings work correctly and is happy to debug issues on slack.knative.dev.

Scaling From Zero

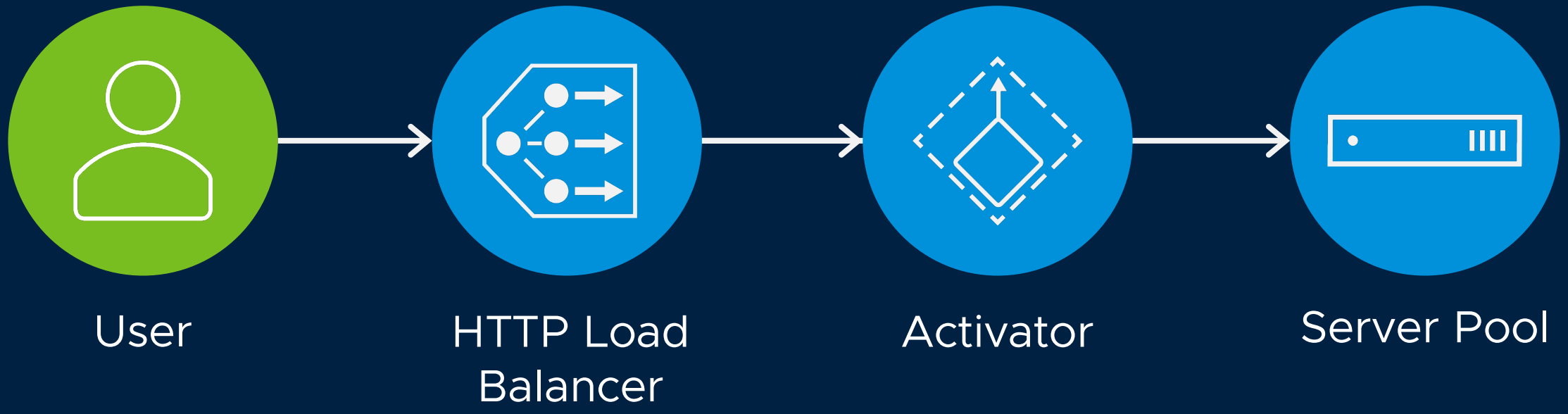
Stay on the line, please

The load balancer queues (pauses) the incoming request and asks the server pool to start an instance. When the instance is ready, the load balancer forwards the paused request



Scaling to Zero

Practically, Knative uses a shared (multi-tenant) component called the *activator* to perform the pause-and-forward for scale-from-zero. Scaling to zero means adding the activator to the load balancer before removing the server pool.



Serverless *HTTP* applications

Knative works with applications which expose an HTTP server on a known port.

Why HTTP:

- Ubiquitous, well-understood
- Ongoing improvements with HTTP/2, Websockets, etc
- Shares resources (IP, etc) well
- If we know the protocol, we can do :magic:

Knative specializes Kubernetes for stateless HTTP applications. Kubernetes supports applications which use multiple protocols and ports, and have many scaling models. The tradeoff is that Kubernetes provides a lot of settings even for a basic deployment.

Serverless *HTTP* Applications

… less YAML, too

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - image: docker.io/test/myapp
        ports:
        - name: http
          containerPort: 8080
```

```
apiVersion: v1
kind: Service
metadata:
  name: myapp
spec:
  ports:
  - port: 80
    targetPort: h2c
  selector:
    app: myapp
```

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: myapp
spec:
  template:
    spec:
      containers:
      - image: docker.io/test/myapp
        ports:
        - name: http
```

This also includes an autoscaler and some other magic we'll talk about soon!

Serverless *HTTP* Applications

I promised some magic

What else do you get out of the box with Knative?

Autoscaling (already covered)

Automatic management of HTTP hostnames

- Map a wildcard address to your cluster, and no further need to touch DNS!

Automatic tracking of previous states (Revisions)

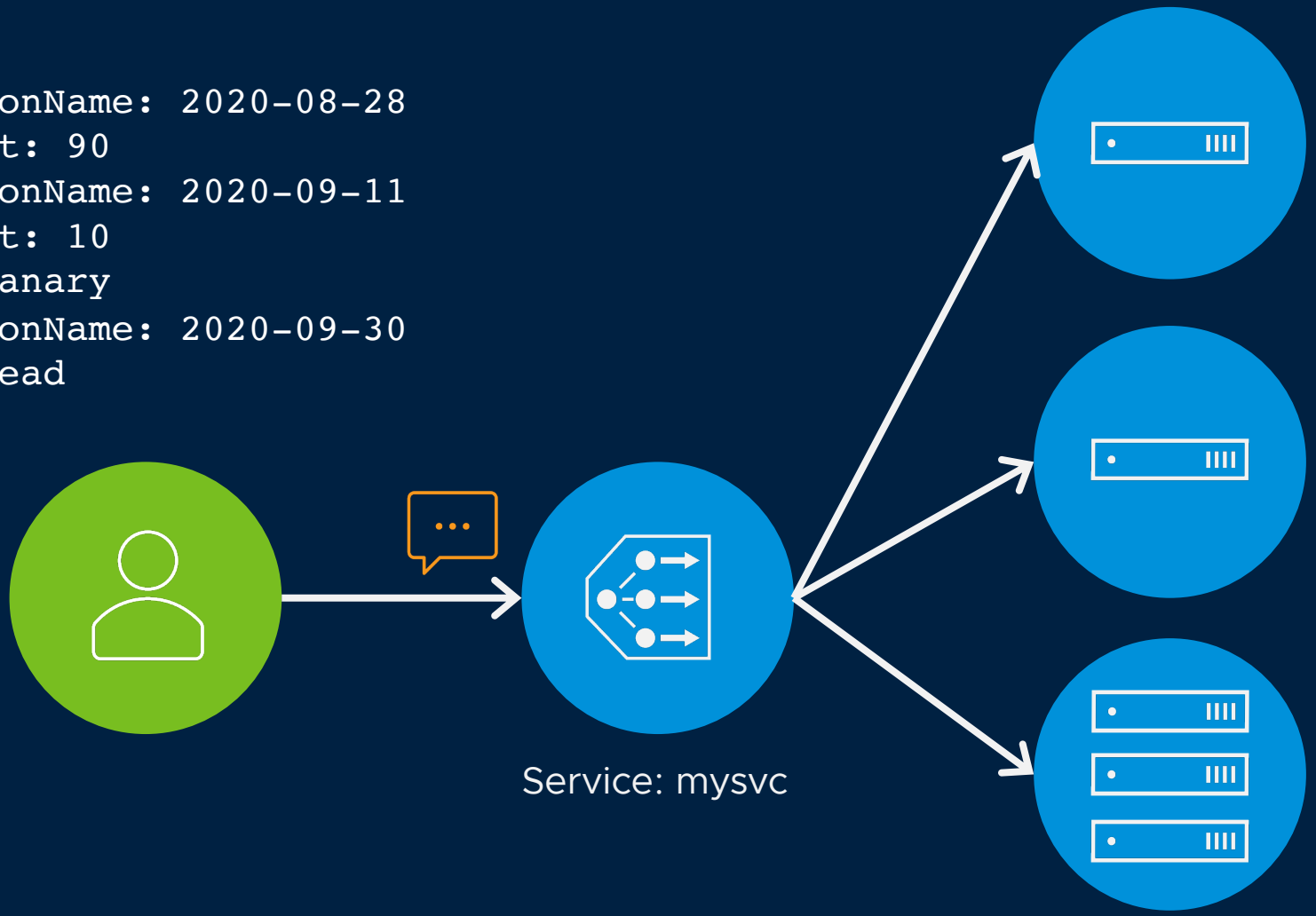
- Every update to the Service's spec creates a new Revision
- These Revisions are immutable and automatically garbage collected

Ability to roll back and canary traffic between Revisions

- Can route traffic on a percentage basis, regardless of how many instances
- A traffic route can refer to a specific Revision, or the last Ready Revision
- Can also add tags to reach a specific Revision by a unique hostname – handy for testing

Traffic Splits and Tags

```
traffic:
  - revisionName: 2020-08-28
    percent: 90
  - revisionName: 2020-09-11
    percent: 10
    tag: canary
  - revisionName: 2020-09-30
    tag: head
```



`Host: head.mysvc.example.com`

percent: 0

tag: head

(Never reach this via `mysvc.example.com`)

`Host: canary.mysvc.example.com`

percent: 10

tag: canary

`Host: mysvc.example.com`
(10% of time)

(No hostname to force this version)

percent: 90

(no tag)

`Host: mysvc.example.com`

(90% of time)

Serverless HTTPS applications

… one more bit of magic

Knative Serving can be configured to automatically provision certificates for your automatically-created hostnames.

- Out of the box, this uses LetsEncrypt and certmanager
- You could also integrate this with your own CA
- Supports both DNS (wildcard) and HTTP certificate solvers

Requirements:

- Your serving domain must be reachable by LetsEncrypt (i.e. on the internet)
- LetsEncrypt supports about 50 certs/week/domain prefix.

Knative Eventing

CloudEvents delivery over HTTP

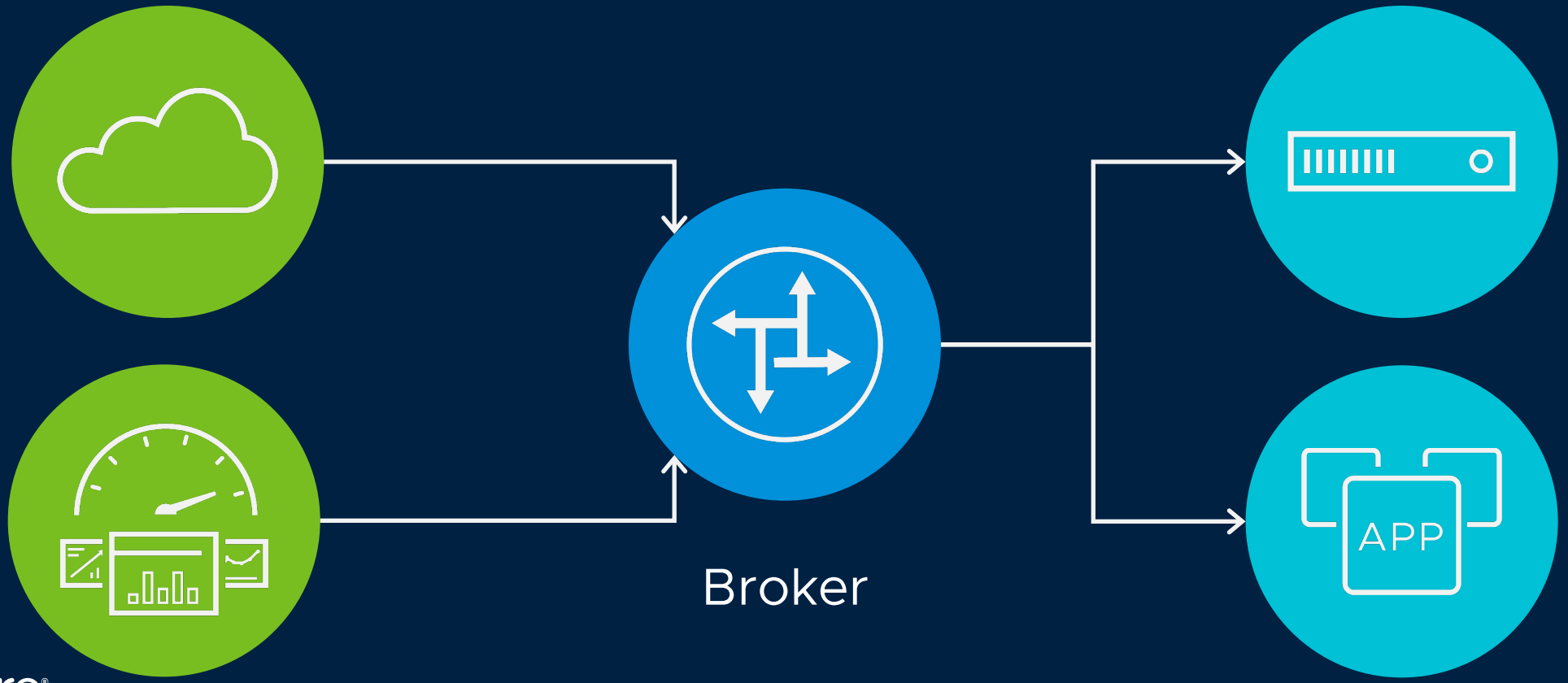
CloudEvents Delivery over HTTP?

CloudEvents is a Cloud Native Compute Foundation effort to standardize event processing metadata. It defines a simple envelope (attributes + body) for an event, and describes how to encode that data in 8+ formats, including AMQP, AVRO, JSON, and HTTP.

Knative Eventing provides a set of Kubernetes-native abstractions that work well with Knative Serving (or anything else that speaks HTTP) for interacting with CloudEvents

Easy Event Delivery

The Broker provides a central place to send and receive events between different applications.



Easy Event Delivery

The Broker provides a simple abstraction for event publishers and subscribers.

For publishers:

- A simple default address to persist events (low configuration)
- High availability endpoint to avoid complex queues on the publisher side

For subscribers:

- Don’t need to coordinate with publisher on channel names, etc
- Easy to subscribe to multiple events or a subset of events
- Broker will ensure at-least-once delivery by retrying HTTP requests if needed

The default configuration is a single Broker per namespace, but other configurations are also supported at some additional cost.

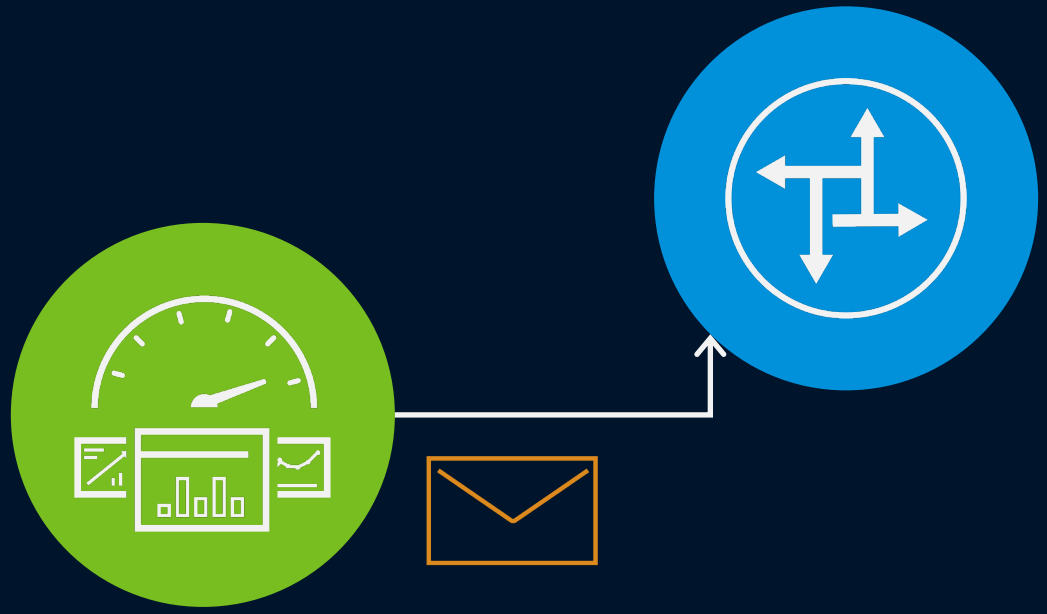
Sender example

(Java 11)

```
CloudEvent evt = CloudEventBuilder.v1()
    .withId(auditLogId)
    .withType("com.example.userupdate")
    .withSource(serverEnv)
    .withSubject(getUserUrl(user))
    .withExtension()
    .build();

HttpClient client = HttpClient.newHttpClient();
var request = HttpRequest.newBuilder(
    URI.create(brokerUrl));
var writer = HttpMessageFactory.createWriter(
    request::header,
    body -> { request.POST(
        BodyPublisher.ofByteArray(body)); });

writer.writeBinary(evt);

var response = client.send(
    request.build(),
    BodyHandlers.discarding());
```

Goal:

Easy to add events to existing stack



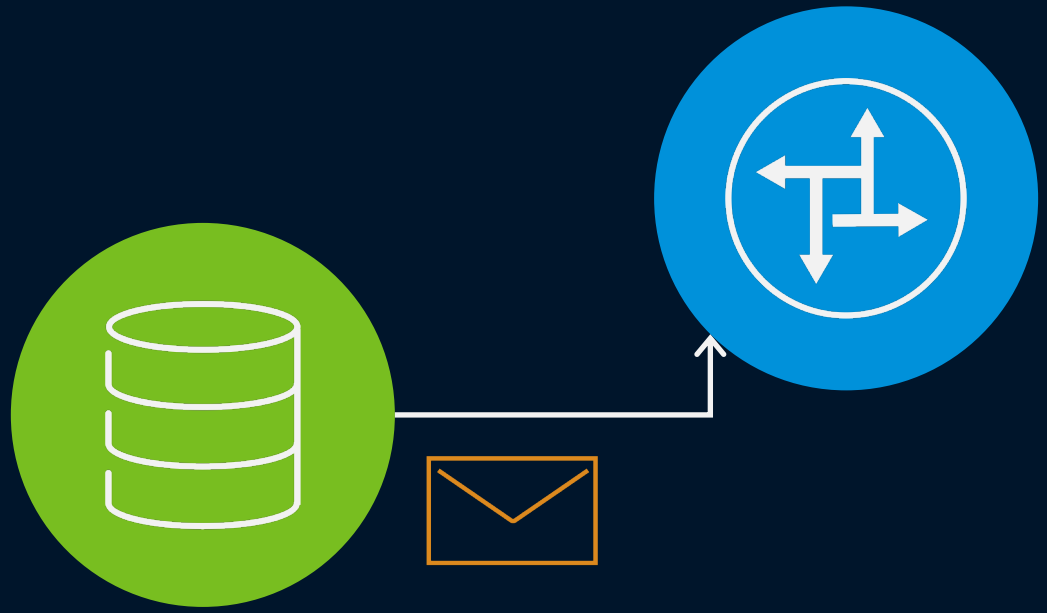
Sender example, part 2

(Kafka Event Source)

```
apiVersion: sources.knative.dev/v1beta1
kind: KafkaSource
metadata:
  name: read-events
spec:
  consumerGroup: read-events
  bootstrapServers:
  - my-cluster-bootstrap.kafka:9092
  topics:
  - ops-log
  sink:
    ref:
      apiVersion: eventing.knative.dev/v1
      kind: Broker
      name: default
```

Goal:

Provide out-of-the-box tools for common event sources



Subscribing to events

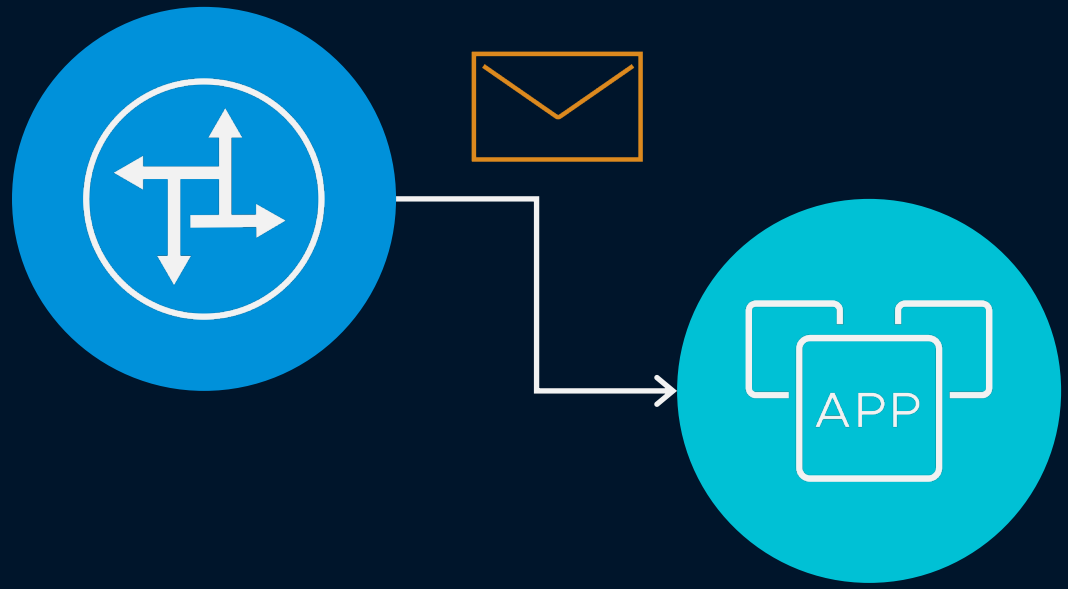
Run a webserver that accepts CloudEvents.
You can use Knative Serving if you want. 😉

Create a Trigger:

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: user-audit
spec:
  broker: default
  filter:
    attributes:
      type: com.example.userupdate
      source: https://prod-login.example.com/users
  subscriber:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: user-audit
```

Goal:

Easy to subscribe to events



Where are my events?

Why didn't you just use Kafka, RabbitMQ...

Knative Eventing uses a pluggable model to integrate with multiple messaging systems, including Kafka, NATS, RabbitMQ, GCP PubSub and an in-memory system for development environments.

Two pluggable abstraction layers:

Channel

Fan-out message persistence

Underlying integration with log or pubsub systems like Kafka or NATS.

No filtering, just message storage and retry. Also useful for building workflows like a Sequence.

Broker

Content-based event delivery

Several implementations manage an internal Channel for fanout and apply filtering at the egress.

Other implementations integrate directly with message router features (RabbitMQ).

Replying to Events

A common pattern in event-driven architectures is chained event processing -- one component processes an event, then emits a new event to be processed by the next step.

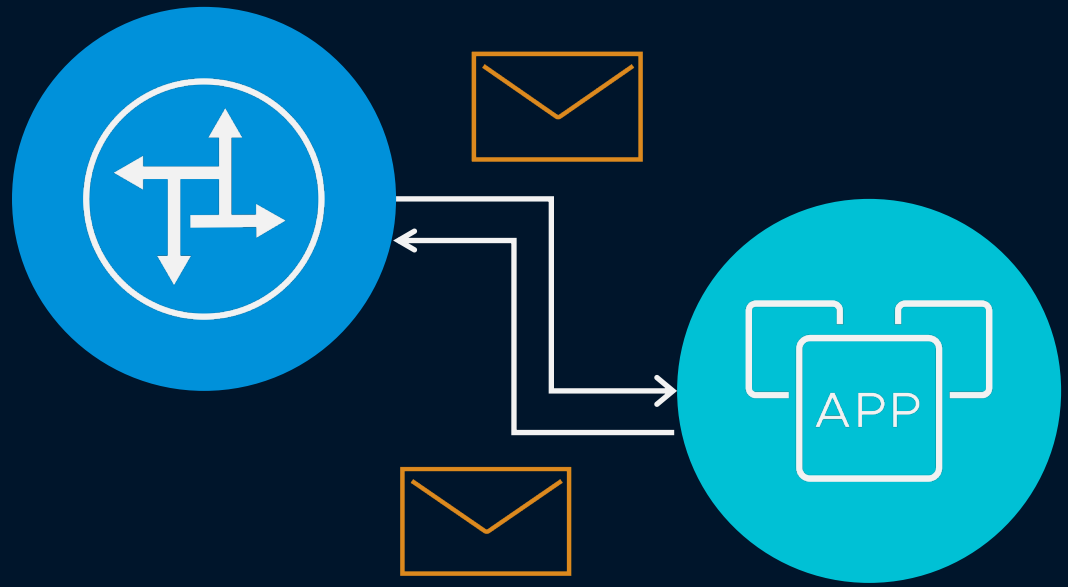
Knative uses CloudEvents' HTTP reply encoding to allow an event recipient to respond with a new event. Using an HTTP reply (rather than having the function enqueue the next event) makes it easy to externalize the event routing and reuse a function in different environments. (For example, to use the in-memory Channel during development, and switch to Kafka in production.)

Replying to Events

```
POST / HTTP/1.1
CE-Specversion: 1.0
CE-Type: com.twitter.tweet
CE-Source: https://twitter.com/
CE-Id: 3a411b72
Content-Type: application/json

{ ... }

----
202 Accepted
CE-Specversion: 1.0
CE-Type: com.example.analyzed-tweet
CE-Source: tweet-analyzer
CE-Id: llab4:448821
Content-Type: application/json

{ ... }
```

Goal:

Easy to subscribe to events



Demo

http://github.com/evankanderson/vmware-codeconnect-2020

What we saw:

Knative extends and interoperates with the Kubernetes ecosystem.

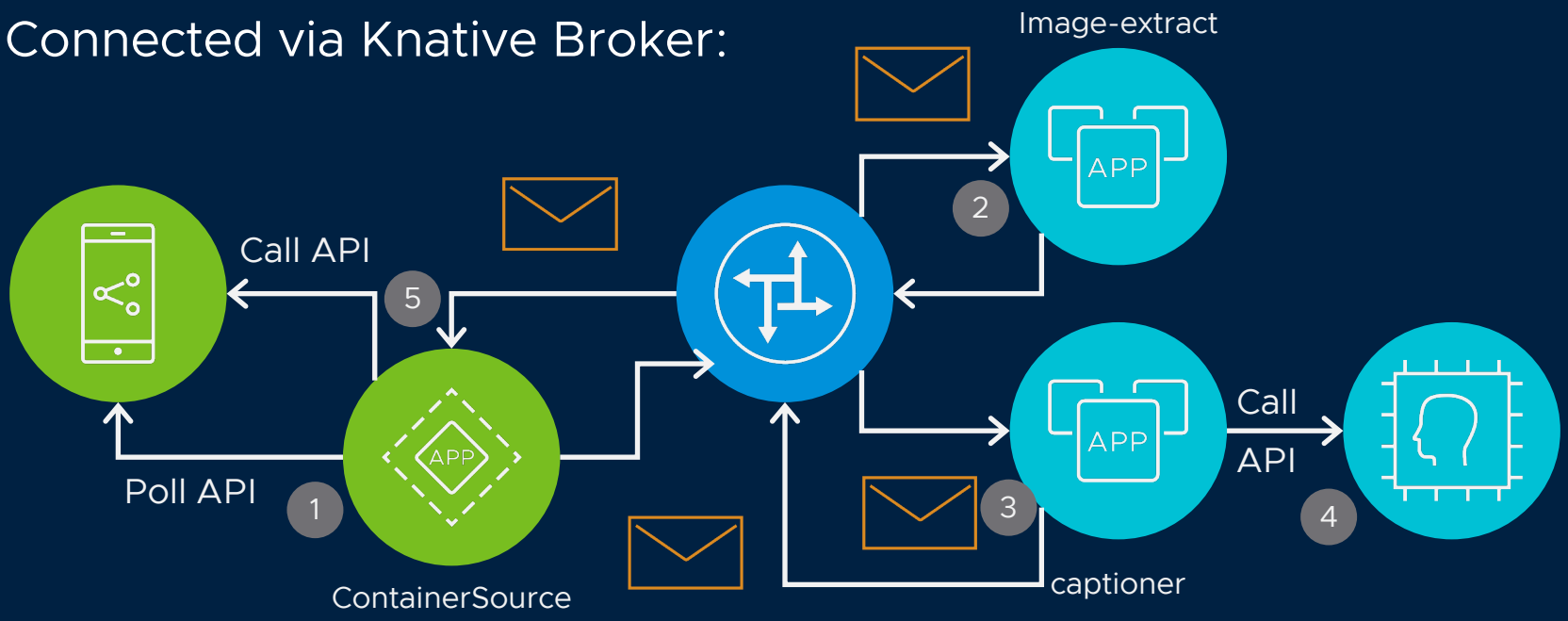
You can mix & match Knative with standard Kubernetes constructs.

Added two more languages:

Node.js

Golang (via packaged ContainerSource)

Connected via Knative Broker:



Three Triggers, one Broker

Using event type to route stages

Fetch image from tweet

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: extract-image
spec:
  broker: default
  filter:
    attributes:
     type: com.twitter.tweet
     source: https://twitter.com/
  subscriber:
    ref:
     apiVersion: serving.knative.dev/v1
     kind: Service
     name: tweet-extractor
```

Read caption, return captioned image

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: caption-image
spec:
  broker: default
  filter:
    attributes:
     type: demo.twitter-image

  subscriber:
    ref:
     apiVersion: serving.knative.dev/v1
     kind: Service
     name: captioner
```

Send captioned image to Twitter

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: send-tweet
spec:
  broker: default
  filter:
    attributes:
     type: demo.captioned-image
     datacontenttype: image/jpeg
  subscriber:
    ref:
     apiVersion: v1
     kind: Service
     name: tweet-sender
```

Standard Kubernetes Service!

Serverless rules of the road

Serverless Code

Generally, prefer smaller functions and microservices:

Faster cold-start times when a new container is needed

Globals can be okay if the scope of the container is small

Containers scale horizontally quickly – make sure your backend can, too

- Patterns like connection pools can make the problem worse
- The Knative autoscaler has a few tools for this (containerConcurrency, max scaling)

Small units can provide more built-in observability (by-component measurement)

When to combine functionality?

Released at the same time

Using the same backing services

Serverless Events

Sending good events is an art!

Send enough data in the payload that clients can operate on the event itself

Highlight interesting payload data as attributes for routing

CloudEvents attributes should be a summary of the payload, not replace it!

When sending events with PII, consider the Claim Check pattern

- Claim Check is also useful when sending events about large objects (>512KB)

Route events between Knative Brokers using Triggers to cross namespaces

I’m sold!

Where do I get some of that?!

I just want to run some code…

… where?

In the cloud!

Autoscaling
Internet attached

On my laptop!

Fast dev cycle
Hard to share

At the edge!

Low resource
Mixed network

In my DC!

Fixed resource
Corp policy

Knative for all environments

In the Cloud!

Managed K8s
Integrated by cloud provider

Managed K8s integrates with provider's network and storage technologies

Provides the same high-level interface across clouds

Notes on deploying Knative

Defaults may be internet-exposed

Cloud provider versions of K8s may lag behind upstream

On my laptop!

Minikube / Docker / KIND
Kubernetes on a local VM

Run locally and disconnected with the same interface as you deploy to

Dev can own their environment and deploy tools like debuggers

Notes on deploying Knative

SSL and ingress into the cluster can be tricky

Hard to show off to others except by "check out my laptop"

At the edge!

Bare Metal or Virtualized
…that's a lot of Kubernetes

Deployment of one K8s cluster per location. Invest in tools for multi-cluster management

Networking and storage are DIY and may be a challenge

Notes on deploying Knative

Defaults may be internet-exposed. May hit LetsEncrypt rate limits

Backing storage and messaging systems for eventing may introduce additional constraints

In my DC!

Virtualized nodes
Vsphere 6.7 and 7 supported

Reuse your existing hardware and infrastructure investments

Optimize hardware usage and visibility; unify development patterns for on-prem and cloud

Notes on deploying Knative

SSL auto-provisioning may be harder

May experience hardware constraints

Thank You

Thank You

Evan Anderson

@e_k_anderson

Knative: https://knative.dev/

These demos: https://github.com/evankanderson/vmware-codeconnect-2020