Function-as-a-Service on Kubernetes using Knative

August 2019

AUTHOR:
Juan Carlos Gallegos Dupuis

Cloud Infrastructure Team

SUPERVISORS:
Belmiro Moreira
Ricardo Rocha

CERN openlab

INTRODUCTION

Background

The CERN Cloud Infrastructure team provides compute resources as a service to teams across CERN. Users can provision resources to process experiment data, host web applications, and accomplish other computing tasks.

The CERN cloud is built using OpenStack, an open-source suite of technologies for provisioning and managing a variety of computing infrastructure services, including: virtual machines, “bare metal” machines, container orchestration, storage, authentication, and networking. For container orchestration, we use OpenStack Magnum, which supports multiple container orchestration engines, such as Kubernetes. Kubernetes is a widely-used platform for deploying and managing applications in containers. Our team uses these technologies in part because they are open-source and developed by active communities with strong ties to information technology industries.

Motivation

Our team is interested in extending our infrastructure offerings to include serverless application deployment, which abstracts away even more infrastructure management from the user than traditional services like virtual machines. Amazon Web Services (AWS) popularized the notion of serverless computing with their introduction of AWS Lambda, which provides a Function-as-a-Service (FaaS) platform. Our team strives to offer a similar serverless computing service. We are particularly interested in creating a FaaS platform on top of Kubernetes, so that we can simultaneously reap the benefits of using a container orchestration engine.

Knative is a relatively new technology that extends Kubernetes to support serverless computing patterns. It is also an open-source project backed by major industry players like Google, IBM, and Red Hat. For these reasons, Knative is a natural candidate for our purposes. My project was to investigate Knative’s functionality and how our team could use it to add a FaaS platform to our suite of existing cloud services.

TABLE OF CONTENTS

1. INVESTIGATION

My project consisted largely of investigating and experimenting with a variety of technologies. As such, knowledge of these technologies comprise a large part of my project's results. I discuss Function-as-a-Service (FaaS) cloud services, Kubernetes, and Knative, since they played the main roles in the project. These descriptions aim to showcase the knowledge that I gathered and contextualize the rest of the discussion.

a. Function-as-a-Service (FaaS)

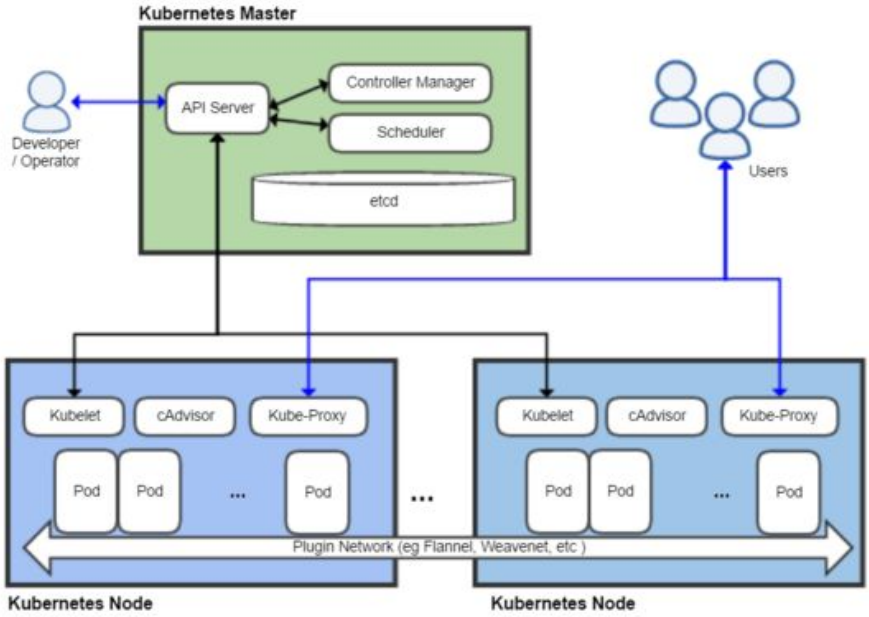
Cloud computing is based on the idea that compute resources can be provided as a service, so that customers do not have to own or even physically maintain the servers that they use every day. Cloud service providers manage the computing infrastructure behind the scenes and expose it over the internet.

Cloud service providers such as Amazon Web Services, Microsoft Azure, and Google Cloud Platform offer infrastructure options with varying levels of abstraction. FaaS platforms, which are offered by all major cloud providers nowadays, are amongst the most abstract; that is, they provide the highest-level interface. In a typical FaaS offering, the user is agnostic about various levels of infrastructure underlying their application; the physical hardware, the virtual machine, and even the web server are all managed and encapsulated.

FaaS platforms are not tied to any specific kind of underlying infrastructure. And so, since our team makes heavy use of Kubernetes, we are evaluating its potential as the base of a new FaaS platform.

b. Kubernetes

Kubernetes provides container orchestration: it manages containerized applications throughout their lifecycle. It supports major operative tasks such as deployment, replication, scheduling, monitoring, and crash recovery. It achieves this in a cluster composed of multiple compute nodes, each of which runs many containers. Each node organizes containers into pods. A node may be of type ***master*** or ***minion***. A ***master*** node hosts the controllers that manage the cluster, including the Kubernetes controllers that provide the operative functionality mentioned above. A ***minion*** node hosts the actual applications.



Users of Kubernetes are typically software developers and operations personnel. Users deploy and manage an application on Kubernetes by declaring its desired state; Kubernetes continually ensures that the actual state of the application matches the stated desired state. For example, if a user states that they want three replicas of their application running at all times, Kubernetes will not only deploy the three replicas, but also replace them whenever they crash.

The users communicate the desired state by sending files containing configuration details and a link to their containerized application (e.g. Docker Hub URL) to the Kubernetes API. From there, Kubernetes takes care of deploying and managing the applications as necessary. Users define the desired state by specifying various different resources, like `Fgrnq{ogpv` and `Kpitguu`. The Kubernetes API understands a predetermined set of resources, but it may be extended using ***Custom Resource Definitions*** (CRDs). In addition, one may create custom controllers capable of interpreting CRDs sent to the Kubernetes API by the user and modifying the cluster's state accordingly.

c. Knative

As mentioned earlier, Knative extends Kubernetes. It does so by defining a set of CRDs and implementing controllers that instantiate those resources in the cluster. Once installed on a cluster, Knative is accessible through the Kubernetes API.

Knative's CRDs and controllers are packaged into two components: Serving and Eventing. (In version 0.8, Knative deprecated a third component: Build.) Knative Serving supports deployment of serverless applications; it is Knative's fundamental unit of functionality. Knative Eventing supports publisher-consumer patterns; publishers may exist within or outside the Kubernetes cluster, and consumers are Knative services. These components allow users to pursue a serverless architecture consisting of cloud functions that run in response to generic events, all without the need to directly manage the infrastructure. Communication between services is all handled by Knative and Istio.

i. Knative Serving

The Knative Serving component defines four CRDs. The `Ugtxkeg` CRD encapsulates the others: `Tqwvg`, `Tgxkukqp`, and `Eqphkiwtcvkqp`. When a user creates a Knative Service to deploy their application, Knative creates a corresponding revision and prepares routing to make the application accessible over the network. The relationship between these resources is reflected by the Knative `Ugtxkeg` specification, in which the `Tqwvg`, `Tgxkukqp`, and `Eqphkiwtcvkqp` specifications are embedded.

A single service is associated with many revisions, each of which is a snapshot of an application's code and configuration packaged together. Knative creates a brand new revision in response to simple changes, like modifying an environment variable, and more complex changes, like changing how traffic is distributed across revisions.

ii. Knative Serving

Knative Eventing encapsulates communication between publisher and subscriber. Knative supports various publishers, including GitHub, AWS SQS, and Kubernetes; any application running as a Knative `Ugtxkeg` can be a subscriber.

As mentioned earlier, events may originate from outside or inside the cluster. In either case, a custom resource represents the event source. For example, the `IkvJwdUqwteg` resource acts as a proxy for incoming events received from a GitHub repository. In the case of simple, scheduled events, the `EtqpLqdUqwteg` resource itself produces the events.

Event sources may emit a message directly to a subscriber or to an intermediate channel. Direct communication is simpler, but channels provide durability guarantees to cope with communication failures.

d. Istio

Istio is a service mesh: it manages applications distributed over the network. It aims to support implementation of microservice architecture. Once deployed on a cluster, Istio provides functionality such as service discovery, inter-service communication, telemetry and logging aggregation, and access control. Notably, Knative depends on it for managing routing and inter-service communication.

2. Working with Knative

The following sections detail my experience working with Knative, which included: installation on a Kubernetes cluster; deploying sample serverless applications; creating distinct revisions of a single service and routing traffic across them; creating publisher-subscriber patterns; and investigating how to apply Knative to a use case at CERN.

a. Installing Knative

Knative was entirely new to our team, so I wrote scripts and documentation to help other members of the team to get started quickly with it in the future. The scripts are for creating a cluster, installing Knative and Istio, and testing the installation, and they are available in this GitLab repo.

As mentioned earlier, Knative is most commonly paired with Istio to satisfy its dependency on a service mesh. The installation process for Knative and Istio consists of sending configuration files to the Kubernetes API; these files contain various resource definitions, including various CRDs. These resources include links to controllers that are downloaded and deployed onto Kubernetes pods. At the end of the installation process, there are many pods across new namespaces (e.g. `mpcvkxg/ugtxkpi`, `mpcvkxg/gxgpvkpi`, `mpcvkxg/oqpkvqtkpi`, `kuvkq/u{uvgo`) that implement Knative's and Istio's functionalities.

The installation guide that is linked above includes the deployment of a minimal "Hello World" application as a Knative Service.

b. Deploying an Application as a Knative Service

Deploying a serverless application with Knative is simple, but it has a few prerequisites. The application itself must be capable of receiving network messages; for example, it

might implement an HTTP API. Once the application is implemented, it must be containerized using Docker. Additionally, if the application sends outbound network messages, the Kubernetes cluster must be configured to allow it.

Allowing outbound network traffic is a matter of tweaking Knative's network configuration. By default, Istio intercepts all network traffic; it forwards any intra-cluster traffic, but blocks any outbound traffic. So, we must edit the `eqphki/pgvyqtm` Config Map (which is a Kubernetes construct) such that only traffic between cluster nodes is intercepted.

With all this in place, we can deploy a serverless application by providing the configuration for a single Knative `Ugtxkeg` resource to the Kubernetes API. In response, Knative will immediately deploy two pods, each containing an independent instance of the application. After several seconds of inactivity, Knative will shut them down. (It is possible to configure the amount of time that Knative waits before shutting down an inactive service.) At that point, we can trigger a new deployment by sending a request to the application. In response, Knative will run the application on two new pods and then forward the request to one of the instances. This just-in-time deployment functionality is fundamental to implementing a FaaS platform.

c. Routing Traffic Across Revisions

Any given deployment of a Knative Service corresponds to a single revision. As mentioned earlier, revisions encapsulate code and configuration at a given point in time. Any changes result in a new, distinct revision.

Knative allows developers to distribute traffic by percentage across different revisions of a service. Using this feature, developers can easily release new versions gradually and perform advanced operations techniques like canary deployment. For example, they might create a specific revision for a new version of their application and deploy it to handle only ten percent of traffic; this is a good way to validate a release before making it widely available.

Throughout its early stages of development, Knative's YAML specifications underwent various changes. And so, numerous tutorials written within the last year contain examples with deprecated fields. Fortunately, the v0.7 Serving API Docs detail the canonical definitions of Knative CRDs and their fields.

d. Using the Knative Eventing Component

The Eventing component was one of the most exciting parts of Knative. FaaS platforms owe much of their value to how easily they may be configured to integrate with other services. For example, AWS Lambda connects easily with other AWS services like S3 so that a developer can focus almost exclusively on writing the application logic.

I investigated how to use Knative's Eventing component and created a tutorial for producing scheduled events using Knative's event sources and consuming them with Knative services. All went smoothly when following the tutorials for simple eventing examples, in which the event originated within the cluster: for example, scheduled events using a cron-like tool. However, when I worked on connecting Knative to GitHub and GitLab, I came across various issues.

i. GitHub as an Event Source

In the case of GitHub, I discovered after some investigation that the necessary `IkvJwdUqwteg` CRD was not included in the original installation of Knative. I found a link to the appropriate file in the Knative documentation and installed it. Afterwards, I ran into a new issue.

It was clear that something went wrong when I reached the end of the tutorial, and my GitHub repository, to which I attempted to connect my cluster, showed no registered webhooks. After some investigation, I found out that the `IkvJwdUqwteg` resource could not communicate with the subscriber, which was a Knative `Ugtxkeg`. Specifically, I found a cluster event stating that the event ***sink*** (i.e. the subscriber) could not be found because it contained an "empty hostname". Unfortunately, very few online resources existed for this particular problem, so troubleshooting was difficult. I eventually came across the same error message with the `IkvNcdUqwteg`.

ii. GitLab as an Event Source

In the case of GitLab, a broken link to its Kubernetes controller delayed my progress for over a week. I submitted an issue to the GitLab repository and it was resolved 11 days later. At that point, I started the GitLab tutorial, but ran into an issue very similar to the one I saw when trying to work with the GitHub event source: the target service had an "empty hostname".

After some investigation, I concluded that perhaps the empty hostname problem could be fixed by using the latest released version of Knative, which was released partway through my project. This inference is based on a GitHub Issue submitted to the Knative Eventing repository.

The problem could have also been caused by the lack of a custom host name for the cluster; however, I think this is unlikely since the host name for the cluster would, in theory, only be necessary ***after*** the webhook was registered with GitHub, so that GitHub could publish events to the cluster.

Lamentably, numerous hurdles slowed my progress with GitHub and GitLab eventing. However, it is worth mentioning that these event sources are still labeled as Proof of Concept by Knative. As Knative matures, it should become easier to work with them.

e. Applying Knative at CERN

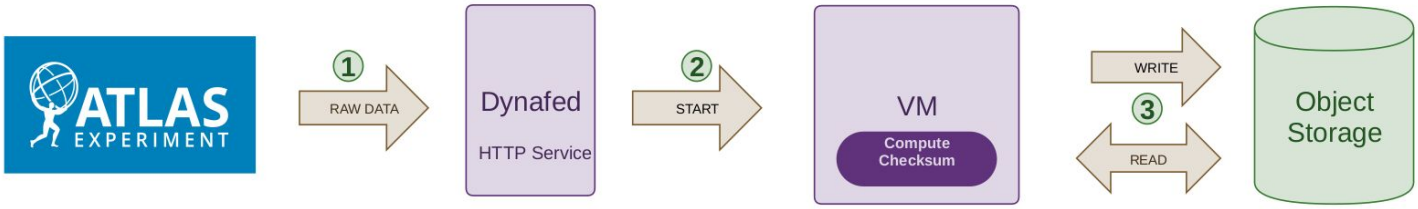
Of course, a main motivation for this project was to determine how FaaS could be employed by users of the CERN Cloud. Before we even started looking for candidates, a developer on the CERN IT team took the initiative to contact us regarding Knative. He wanted to explore Knative for cluster-based FaaS infrastructure that would scale his service automatically.

His service, called ***Dynafed,*** provides an HTTP API for storing files of experiment data in object storage. Specifically, it exposes endpoints for reading, writing, copying, and computing the checksums of files. It currently runs on a virtual machine, where it spawns local worker processes as Docker containers to do the work. The developer was considering using cloud functions, as provided by Knative, to do the work instead.

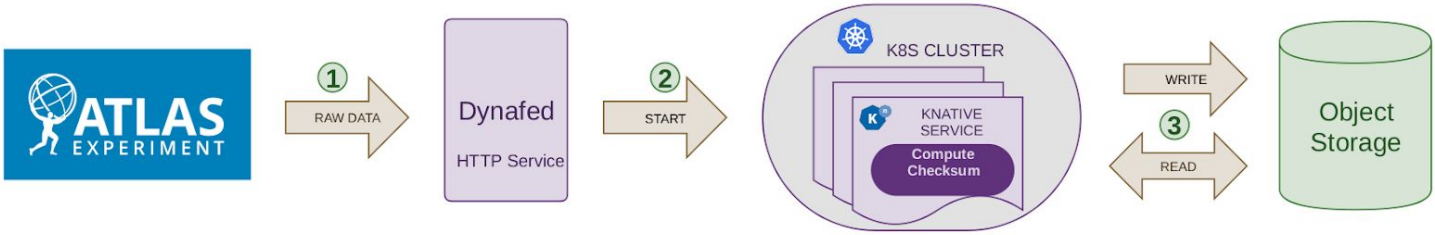
FaaS is a good candidate for this scenario because it automatically scales up and down. If ***Dynafed*** were to experience a high volume of requests, Knative would create multiple functions to cope with the load. On the other hand, if Dynafed was experiencing few requests, there would be few or no instances actually running.

The developer was particularly interested in implementing the file copying and compute checksum tasks with cloud functions. The difficulty with the file copy task was that it required that the progress be continuously reported. We were unsure what implementation approach to take in that case, so we focused on the compute checksum

task. That way, we could create a prototype to determine whether the FaaS architecture suited the use case before committing to more complex work.



I implemented a prototype to show how to deploy an HTTP service using Knative Serving. It consists of a Python web application built using Flask and Gunicorn, packaged into a Docker container. It demonstrates how Knative could be used to deploy a service meant for performing a simple, self-contained task. Deploying it as such relieves the developer from infrastructure management responsibilities. However, this benefit is not truly gained unless the existing service no longer runs on a virtual machine.



3. CONCLUSION

All in all, the project demonstrated potential in Knative as a tool for building a FaaS platform. While Knative is still somewhat immature, it shows promise as a Kubernetes-native platform for providing FaaS infrastructure solutions. Our team is excited to continue investigation into Knative and FaaS in general, as they could help us offer a substantially new type of cloud service.

a. Future Work

Our team must define the type of interface we would want to expose to users of a FaaS solution. For example, we must determine whether we would encapsulate the details of working with Knative or if we would require the user to interact with Knative directly. Hiding the implementation details from the user would be more complex, but it would provide a better user experience and allow us to change implementation details without requiring the user to adapt their workflow.

Moreover, our team must determine whether multiple users would share a single Kubernetes cluster in the default case, or if all users would have their own. While sharing a cluster would allow for more efficient usage of computing infrastructure, it also raises questions about the level of isolation that would be necessary to secure each application on the cluster.

Finally, our team must continue working with Knative's Eventing component to integrate with GitLab and other resources that we commonly use, like OpenStack. Event-driven processing is where FaaS solutions stand out amongst the multitude of other cloud services and thus merits investment.

4. REFERENCES

a. Bibliography

Knative V0.7 Official Documentation

Python 3 Documentation

Flask Documentation

Gunicorn Documentation

Kubernetes Overview, by Daniel Pereira, Just Another Dev Blog, 22/02/2017

Admission Controllers in Kubernetes, by Malte Isberner, Kubernetes Blog, 21/03/2019

Knative Basic Tutorial, by Kamesh Sampath, Red Hat OpenShift Blog, 05/10/2018

Knative: Configuration, Routes, and Revisions, by Kamesh Sampath, Red Hat OpenShift Blog, 05/25/2018

Knative Tutorial, by Mete Atamel, Google, 09/07/2019

Kubernetes.png, by Khtan66, Wikipedia, 25/11/2016

Knative Logo, from Knative's GitHub page

Kubernetes Logo, property of Kubernetes

b. Links to Project Deliverables

Scripts and Notes:

- Setting Up Knative
- Distributing Traffic Across Revisions
- Consuming Cluster Events

Other:

- CERN Cloud Documentation