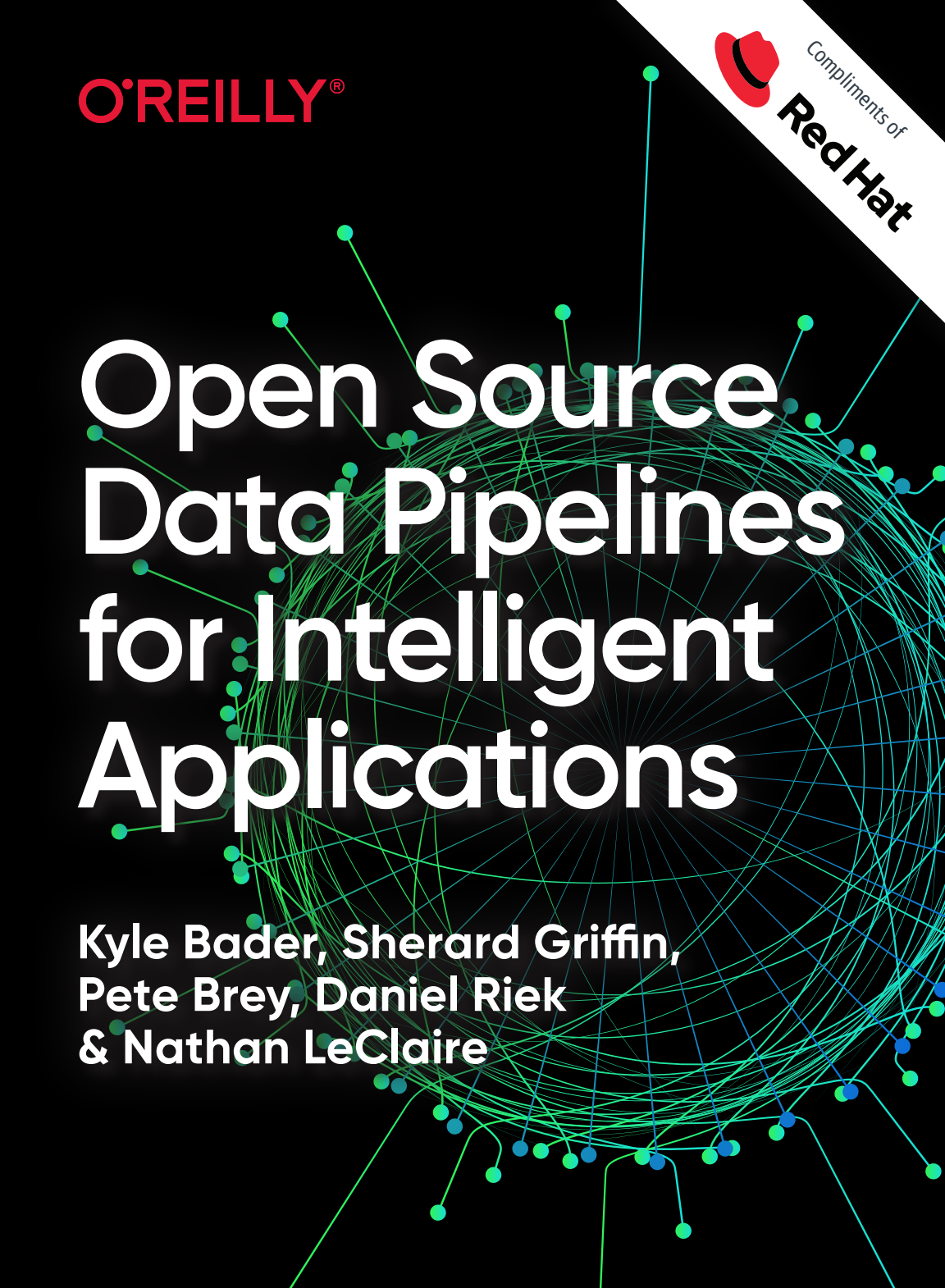
O'REILLY®

Compliments of Red Hat



Open Source Data Pipelines for Intelligent Applications

Kyle Bader, Sherard Griffin, Pete Brey, Daniel Riek & Nathan LeClaire

REPORT


Red Hat
Ceph Storage
Focus on Innovation
Store easier. Build better. Deploy faster.
redhat.com/ceph
openshift.com/storage

Open Source Data Pipelines for Intelligent Applications

Kyle Bader, Sherard Griffin, Pete Brey, Daniel Riek, and Nathan LeClaire

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Open Source Data Pipelines for Intelligent Applications
by Kyle Bader, Sherard Griffin, Pete Brey, Daniel Riek, and Nathan LeClaire

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Acquisitions Editor: Colleen Lobner
Development Editor: Sarah Grey
Production Editor: Kristen Brown
Copyeditor: Arthur Johnson

Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

March 2020: First Edition

Revision History for the First Edition
2020-03-04: First Release

978-1-492-07425-0

[LSI]

Table of Contents

CHAPTER 1
Introduction

Every day, businesses around the world make decisions, such as what they should stock in inventory, which items they should recommend to their customers, when they should email prospects, and more. At best, we guide these decisions with a careful process, analyzing available information with rigor. At worst, we make these decisions by striking out in the darkness with limited data, hoping that reality will match up with our mental models.

We live in a time where being data driven is not only table stakes; it's the future upon which a competitive advantage is built. It's no surprise, then, that we're seeing a renaissance of methods for processing and storing the ever-increasing amounts of information businesses generate. In this report we'll examine a number of aspects of running data pipelines, and intelligent applications, with Kubernetes. We'll look at:

Chapter 1, Introduction
: The history of open source data processing and the evolution of container scheduling

Chapter 2, A Platform for Intelligent Applications
: An overview of what intelligent applications are

Chapter 3, Object, Block, and Streaming Storage with Kubernetes
: How we use storage with Kubernetes for effective intelligent applications

Chapter 4, Platform Architecture
: How we should structure our applications on Kubernetes

Chapter 5, Example Pipeline Architectures
: Some examples, with concrete details, of deploying intelligent applications on Kubernetes

In this introduction, we'll take a look at where we've come from as an industry so we can understand the landscape of the modern ecosystem as well as where the future is headed. We'll see how data analytics infrastructure has evolved and how Kubernetes opens the doors to declarative data infrastructure.

Evolution of the Data Analytics Infrastructure

In the early 2000s, as systems began emitting more and more data, it became more difficult to process that data reliably. Working with that much data on only one commodity computer was too slow, and distributed processing was finicky. Tasks performed on generated data, such as server access or search query logs, had to be split over multiple machines to finish with reasonable speed, but this approach introduced all sorts of complications. The logic of processing the data and of scheduling that processing was intermingled—so people who needed to harness the power of a large cluster of machines also had to worry about state coordination and resource scheduling. A new way of processing this data in these periodic jobs was needed.

Mapping and Reducing

Engineers at Google noticed that many large distributed jobs had the same pattern: one phase to calculate keys based on underlying data, and a second phase to aggregate the keys. They implemented MapReduce, a system for describing these two-step jobs with a unified paradigm. Programmers could simply outline the "map" and "reduce" steps without having to coordinate the careful management of parallelization and fault tolerance across the system. A surprising number of desirable workloads could be described this way, such as:

Distributed Grep
: Emit the log line itself if it follows a given pattern—or emit nothing otherwise.

Indexing URL Popularity
: Emit one `<URL, 1>` pair for each reference to an indexed URL and then sum them.

In-House Analytics

Mark each distinct URL of the app from access logs and then sum them.

A groundbreaking paper by Jeffrey Dean and Sanjay Ghemawat of Google put MapReduce on the map as a method for describing such distributed workloads.

The MapReduce paradigm, which soon become popularized by the Apache Hadoop open source project, gained momentum as programmers were suddenly able to describe their multiple stages of data transformations (often known as a *data pipeline*) using a common workflow that freed them from managing distributed logic. Hadoop's distributed filesystem, HDFS, was minimal but effective in distributing the inputs and outputs of the MapReduce jobs (Figure 1-1). HDFS achieved the goals of the project—namely, to run well on commodity hardware where failures were common and to fit Hadoop's unique filesystem needs. For instance, instead of being flexible for general file read/write performance, HDFS optimizes for reading data out of files as fast as possible and tolerating hardware failures. Accomplishing this required making some tradeoffs, such as relaxing certain POSIX compatibility features.

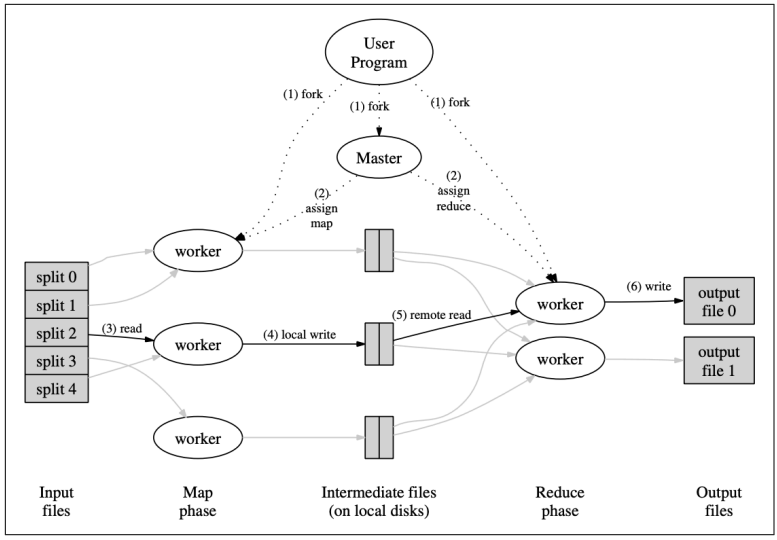


Figure 1-1. Hadoop's computation model; worker nodes map and reduce in parallel to crunch data

Hadoop gained huge popularity, and freshly minted startups clamored to commercialize this exciting new technology. They began applying Hadoop to every use case under the sun, including some that were not necessarily a good fit. As the old saying goes, when all you have is a hammer, everything looks like a nail—and many workloads looked like a Hadoop-shaped nail for a while. With Hadoop as a building block, innovations built on top of it began to emerge, along with improvements to its core technologies. One of the ways this manifested was in systems such as Hive. Hive provided an SQL-like interface on top of the Hadoop framework, democratizing distributed processing for those already familiar with SQL but who otherwise might not be up to the task of hand-crafting a MapReduce job to access the results they were interested in (Figure 1-2).

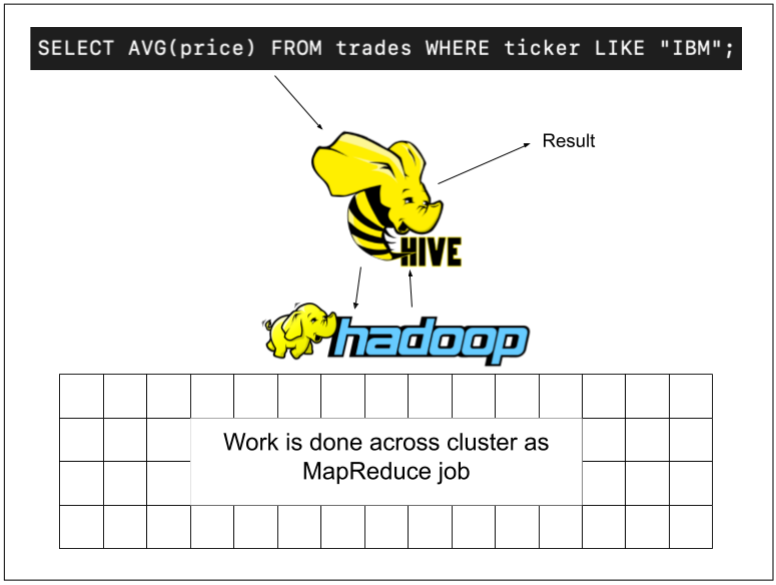


Figure 1-2. Hive was one of several innovations allowing users to access underlying Hadoop data more freely

YARN and Storage

YARN—short for "Yet Another Resource Negotiator"—was introduced in Hadoop 2.x as a new component powering some of Hadoop's decision making. The YARN documentation explains its purpose:

> The fundamental idea of YARN is to split up the functionalities of resource management and job scheduling/monitoring into separate daemons. The idea is to have a global ResourceManager (RM) and per-application ApplicationMaster (AM).

But why bother? Before Hadoop 2.0, compute resource management, job scheduling, job tracking, and job execution were all tightly coupled. The result was inflexible and had difficulty coping with the scale required by web giants like Yahoo! and Facebook. It also made building new tools on top more difficult. With YARN, job execution and the management of resources are decoupled.

Lately, the ability to decouple scheduling and applications has become even more accessible because of platforms like Kubernetes, representing a new logical step in this journey. Kubernetes gives users the ultimate freedom to run whichever applications best suit their needs; they can build data pipelines using innumerable technologies such as Spark, Presto, Kafka, NiFi, and more. While you do indeed need to become a talented YAML wrangler to fully enjoy Kubernetes, the abstractions and flexibility it provides once you are fully ramped up is tremendous. Likewise, innovations in both cloud and on-premises storage are shaking things up.

The storage needs of the modern era are multivariate and complex. As new computation paradigms evolved, the rise of stable and massive-scale storage solutions enabled those developing and operating data pipelines to move with agility. We'll look at various storage needs in Chapter 4.

Evolution of Scheduling

Resource negotiators for both small local computation (like your CPU) and distributed computation (YARN's ResourceManager) have been the subject of research for decades. Just like they pioneered MapReduce and similar technologies as they wrangled with their data processing workloads, engineers at companies such as Google also drove innovation in how to reliably support critical web services at scale, and with minimal toil. Due to the immense scale of the workloads and computational power required to run its production web services, Google could not lean on traditional virtualization. Mostly due to scale, that method of isolating computational workloads would be too slow and expensive. The engineers at Google instead needed a solution that would allow them to isolate and

partition applications without the overhead of emulating a complete operating system for each. This drove innovations in the Linux kernel that would provide the foundations of what we know as Linux containers, popularized by technologies such as Docker and Podman.

Deploying at Google Scale

The development of containerization at Google opened up two distinct advantages for deploying production software. The first was distribution: since each process bundled the related files and metadata that it needed in order to run correctly, it could be migrated between various servers and scaled up or down with ease. The other was resource management: processes had clearly delineated boundaries and scheduling constraints, thereby ensuring that spare computing resources could be used efficiently for spare work without interfering with mission critical services such as search queries and Gmail.

Scheduling Constraints

Running such applications co-resident on the same host introduced a difficult dilemma. If each application were to run on its own dedicated machine, a significant amount of computing power would be wasted if the machine were sized to accommodate peak traffic demands. However, if a background job to process data were kicked off on a node that was already busy during peak traffic, it could easily cause chaos with mission-critical systems. Without the enforcement of entitlements to specific allocations of memory, CPU, and other resources, processes running on the same infrastructure would inevitably step on each others' toes and cause headaches for the system administrators. The Linux kernel feature known as *cgroups* (short for "control groups") allowed administrators to enforce such resource constraints in a particular way that made co-residency of applications safer through containment and preemption, enabling efficient bin packing of applications, reclaiming trapped capacity and driving higher efficiencies (Figure 1-3). Another kernel feature known as *namespaces* enabled each process to have its own "view" of the world, so that system APIs such as the network could be used without the usual challenges of running applications co-resident with each other.

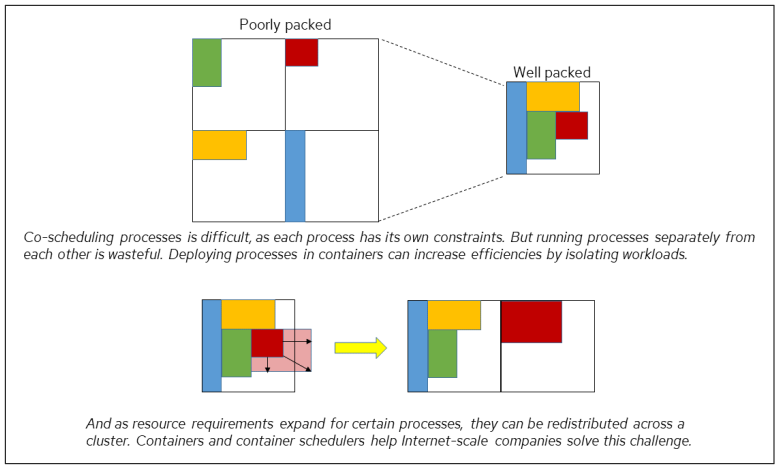


Figure 1-3. Workload scheduling using containers

Container Accessibility and the Rise of Kubernetes

While the underlying features of containerization had been available in the Linux kernel since the early 2000s, the technology was not particularly accessible to most developers until Docker was released to the public in 2013. Suddenly, just as MapReduce had become available to everyone through open source implementations such as Hadoop, building and running container images became a part of every engineer's toolset. This led Google to open-source a reboot of their internal container management software. The reboot was called Kubernetes, based on the Greek word for a helmsman of a naval vessel. While systems such as Docker could build and run one container at a time, Kubernetes could manage a whole fleet of containers in a declarative way (Figure 1-4). With Kubernetes, you can define the desired application or "stack" to the Kubernetes API, and Kubernetes will reconcile the system's actual state with the desired new state. This includes creating cloud infrastructure such as load balancers and block devices automatically for the new applications.

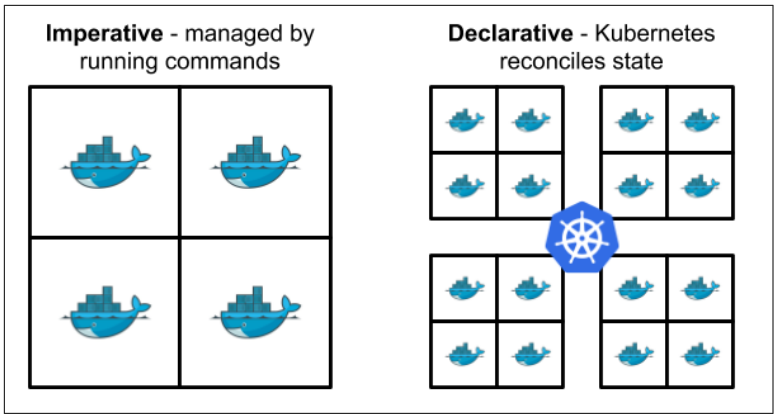


Figure 1-4. While Docker provides a set of imperative commands to run containers on a single host, Kubernetes offers a declarative API and can create your desired infrastructure across a cluster

Everyone Joins In on the Fun

Not long after Google open-sourced the Kubernetes project, engineers from companies such as Red Hat began contributing in the open and improving the project for all to benefit. Red Hat quickly became the number-two contributor to the project, and a large movement in cloud-native infrastructure began to develop. The industry began to rally around Kubernetes as a new general model for treating a cluster as a computer. Some have even gone so far as to claim that Kubernetes is the "distributed operating system" for the modern data center, much like Linux is the open source beating heart of the cloud.

Bringing It All Together

The democratized worlds of big data and warehouse-level computing inevitably began to collide. As scheduling became more general-purpose through projects such as Kubernetes, it found a natural complement in engineers' dissatisfaction with useful technologies being barred admission by specialized schedulers like YARN. New tools like PyTorch and TensorFlow began to proliferate, making it easier for developers to build and deploy sophisticated machine learning models. As Kubernetes assumed the role of declarative infrastructure manager and workload supervisor, data processing needed not be relegated to specialized silos of infrastructure.

Instead, it could be combined to co-exist in the same infrastructure supporting an organization's other applications, maximizing their often significant investment in infrastructure.

As the desire for new ways of creating, organizing, and administrating data pipelines has grown, the new movement to accelerate pipelines with Kubernetes is booming. As you'll see in the next section, Kubernetes is enabling businesses to embrace intelligent applications. New tools like Kubeflow and Open Data Hub will be integral to guiding businesses' decision making and managing the "digital gold" of data.

CHAPTER 2

A Platform for Intelligent Applications

Artificial intelligence is the new must-have for every organization—or at least that's what business leaders have been led to believe. Every day we are flooded with stories about advancements in self-driving cars, personal assistant devices, facial recognition technology, and more. Businesses in all verticals want to take advantage of newly accessible AI technologies, but few understand what exactly that means or where to start. Even fewer businesses understand intelligent applications and their place in the AI story. In this chapter, we'll define the terms *artificial intelligence* and *intelligent applications* and answer some commonly asked questions about them.

Defining Intelligent Applications

Sherard Griffin, one of the authors of this book, illustrates how clear the power of AI is becoming to even the youngest users:

> My 10-year-old son and I attended a conference for developers. As part of the conference, he was interviewed by a journalist who asked his thoughts on AI. After contemplating for a few seconds, he responded that he believes AI will simultaneously be the savior and destroyer of mankind. After a few curious looks from people around the room, he then went on to clarify his thoughts by giving examples of otherwise great AI technologies that exist today which are on the verge of being misused. The first example that came to his mind were virtual AI assistants, a recent addition to our home. He said, "It can really help us do a lot of great things, but

> [companies] can always listen to us and have access to our private data." He also explained how he feels it is our duty to be responsible with the technology and data that we use for building artificial intelligence.
>
> I wasn't surprised by his view of AI as something very promising and powerful. After a few laughs, I remembered that I've been in quite a few conversations where AI is talked about as if it's a mythical creature that can do wonders for a company. As long as you have some data and some type of infrastructure, anything is possible, right?

Here are some examples of the things the authors have seen or heard about AI:

> "We have massive amounts of data coming in and we can't seem to understand most of it. I know—let's try AI."
>
> "Something weird is happening with our systems but I can't find it in the logs. I bet AI can find the problem."
>
> "Our sales team says the customers are happy but contracts keep getting canceled. This sounds like something AI can figure out for us."

It's true that organizations sometimes think of artificial intelligence as a magic box that solves all problems. In reality, building successful AI applications that yield *actionable results* is often challenging and complex. Teams must have the right infrastructure, data, software tools, and talent. Most important of all, they must have clearly defined objectives for what AI is going to do to help their organization succeed.

What Is Artificial Intelligence?

Does everyone in the organization have the same definition of AI? Does everyone understand the limitations of these technologies? At Red Hat, we knew these questions had to be solved first before we could begin starting our AI initiatives. We put a virtual team together to decide on unifying company-wide definitions of *AI*, *machine learning*, and *deep learning*. Here is what our team agreed on:

Artificial Intelligence (AI)
: Computers leveraging code to perform tasks.

Machine learning (ML)
: Computers leveraging code to perform tasks *without being explicitly programmed*.

Deep learning
: Computers leveraging code to perform tasks without being explicitly programmed *and* with the network adapting to new data as it comes in.

With consensus on the definition of AI and its subsets, we were able to create focus areas that were important to our company goals and build initiatives around them, such as implementing AI models in our products, using AI to enhance business processes, and building tools that allow developers to use AI in their technologies. The foundation of all of these initiatives was the most important focus area: our data strategy. We put a team in charge of defining and managing the data strategy, and the outcome of that work was the genesis of Open Data Hub. Open Data Hub is a Kubernetes-based open source AI-as-a-service platform that helps organizations collect and derive value from their data at scale while providing developers, engineers, and data scientists flexible tools to leverage the data in intelligent applications.

With a unified definition of AI, a data strategy, and the right tools, all stakeholders in an organization are empowered to begin applying AI to challenges, while being cognizant of limitations of the technologies. Taking a practical view can help organizations navigate the often confusing landscape more effectively. Next, let's look at what it means to build intelligent applications and why Kubernetes has emerged as the platform of choice for these workloads.

What Are Intelligent Applications?

An *intelligent application* is any application that collects and applies artificial intelligence and machine learning techniques to improve fitness over time. To pull off the "intelligent" part, organizations often need massive amounts of data, and this data needs to be accessed and prepared as efficiently as possible. Without access to the right prepared data, an AI initiative will be thwarted before it starts.

This is where big data engineering, data science, and analytics become critical. Combine these concepts with scalable platform infrastructure and tools that enrich the application developer

experience, and you have the key elements for creating intelligent applications.

Pushing Boulders up Hills

What purpose do intelligent applications serve? If you're familiar with the story of Sisyphus, you'll remember that the Greek gods punished him for trying to cheat death by forcing him to roll a boulder up a hill. When he reached the top, the boulder would come crashing back down—he was doomed to repeat the same thing for all of eternity.

So what does that have to do with intelligent applications? Every organization, like Sisyphus, finds itself spending valuable time and money over and over again on tasks with enough variance that they require human input. The human element required prevents these tasks from being scripted with basic rules-based approaches. If this Sisyphean task were instead being done by an intelligent application, it might roll the boulder up the hill, but it would also learn from variations in the process to adjust to dynamic factors. Then Sisyphus himself could eventually focus on more important tasks, or whatever punishment the gods gave him next.

Intelligent Applications for Logs

Let's look at the use case of analyzing logs from complex build systems. At Red Hat, we create and curate quite a bit of software. This requires an extensive network of build systems that each generate massive amounts of logs. Because of the spider web of dependencies, any irregularity that occurs in upstream community projects may not surface until further down in the build pipeline. If someone or something could monitor the otherwise boring logs in real time, perhaps anomalies could be caught in time and corrected before they are incorporated into downstream product builds. This task could be incredibly valuable, since Red Hat needs to be able to ship new versions of software to production quickly, but giving it to a single person or team to do manually would indeed be a Sisyphean punishment.

We decided to address this problem using intelligent applications. We used Kubernetes to build an application that ingests logs, scans them for irregularities, and notifies responsible parties of potential issues early. (We'll discuss why Kubernetes was our platform of

choice in "Challenges of Using Traditional Infrastructure" on page 18.) We recognized the limitations of having humans doing this tedious yet important task and saw an opportunity for intelligent automation.

Let's take a look at the pipeline for building such an application.

Intelligent Application Pipelines

If you're building an intelligent application, you will need large amounts of data in an optimized format for training models. To store all of our unstructured information for AI workloads, we decided to build a data lake using Ceph Object Storage. (Chapter 3 dives more into which type of storage is ideal for different workloads and how to organize your data efficiently to maximize performance.)

In most cases, data engineers work to clean, process, and enrich data before data scientists can use it for purposes such as model training and validation. In our experience, the size of the data needed to train a model correlates to the algorithms being used and the complexity of the problem you're trying to solve. More variations in inputs, as you might expect, result in more accurate models.

With the system we built, there is no shortage of data available for analysis because our internal systems generate hundreds of gigabytes per day. Handling such quantities of data is no easy task. A good strategy for efficiently storing and processing your data is instrumental. You need the right infrastructure, compression, and encryption algorithms to handle the massive volumes of data. If any aspect of the system is insufficient, keeping up with the demands of users and applications will be a significant challenge. Such a system to transfer data across various locations, transformations, and data formats is often referred to as a *pipeline* (Figure 2-1).

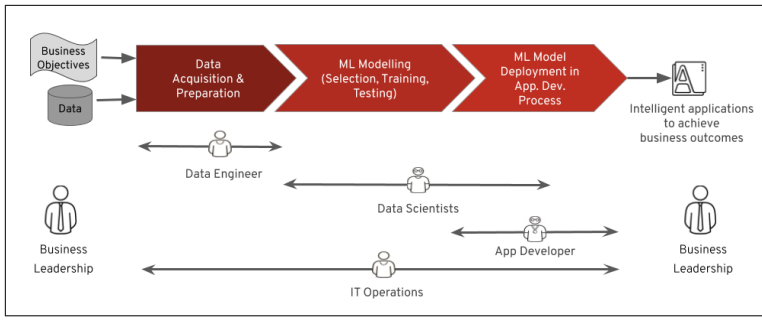


Figure 2-1. An example intelligent application pipeline

Data Science Workflow

What does a data scientist's workflow look like with such a pipeline? Data scientists need tools that allow them to build out complex algorithms and perform advanced computations by exploring vast amounts of data. Many of our own data scientists choose Apache Spark for its flexibility in accessing data from many different sources and for its highly scalable and performant distributed in-memory processing.

Apache Spark's integration with object stores, and in this case with Ceph Object Storage, means that massive amounts of unstructured data can be cataloged and queried in-situ, instead of rehydrating data from another storage system. Once the data is readily accessible, data scientists can build models using a rapidly evolving list of toolkits such as TensorFlow, scikit-learn, R, and more. You can see this architecture in Figure 2-2.

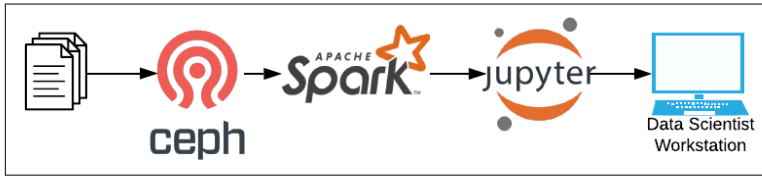


Figure 2-2. Exploring data using Jupyter, Spark, and Ceph

Often the data scientist will validate the model by running a series of benchmarks to evaluate its accuracy. An underperforming model may have its parameters tuned and retested until the results improve to an acceptable level of quality, or more data may be needed to generate "good" results.

Serving Trained Models

Simply preprocessing data and then training models on it isn't sufficient—we also have to make these models available to applications. This step—serving the model in an environment that can be used by developers as they build their applications—is one of the biggest hurdle for organizations to overcome. In fact, many data science initiatives end completely at this stage, before they even spread their wings. We recommend deploying the model as its own service running in containers. (We'll get into more detail on what that looks like in Chapter 5.) This decouples the model from other services and applications that leverage it and allows for independent life cycle management.

When serving a model in a production environment, data scientists must be able to version and sometimes roll back models with relative ease—without help from application engineers—so that they can maintain accurate models. They also need the flexibility of updating their algorithms based on new training data or changes in features. Kubernetes has a natural ability to scale in and scale out container workloads efficiently using ReplicaSets and pod replication strategies. As such, users are able to take advantage of multiversion model-serving engines and scaling with Kubernetes, and can operate different versions (each as their own replica set) and use Istio style A/B testing. Since it helps guarantee availability as requests increase and also controls rolling out new code, Kubernetes is a natural fit for data scientists needing to deploy models.

A Quick Glance at Architecture

Figure 2-3 shows an example data pipeline. In Step 1, Apache Kafka is used as a message bus to ingest data from external systems for processing. The Kafka consumer then takes the messages from a Kafka topic (hopefully in batches) and makes a call to the Flask web app serving the machine learning model (Step 2). The Flask web app applies the machine learning model to predict output based on unseen data and then stores the results in Ceph Object Storage (Step 3). In Step 4, Prometheus is used to gather metrics from the machine learning model's predictions.

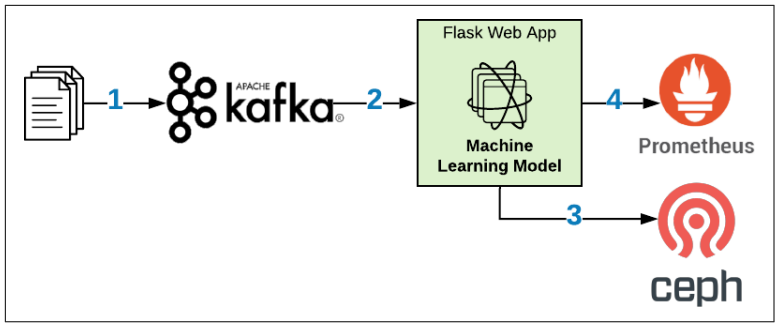


Figure 2-3. Our trained models can be deployed as microservices

A site reliability engineer can use these performance metrics about the number of errors, saturation, and total amount of work done by the system to ameliorate system stability. In tandem, data scientists can use metrics describing the performance of the model itself to improve accuracy and decrease the number of problems caused by anomalies. By splitting each component of the pipeline or overall system into small pieces such as this example microservice, we can limit the impact of any one change on the overall system and improve them all piecemeal.

In Chapter 5 we'll get into more detail on deploying similar use cases on Kubernetes. Kubernetes addresses the unique concerns of data scientists, site reliability engineers, and application developers by easing the deployment of small services and leveraging modern technologies to control rollout strategies. This leads to better intelligent applications for powering our business.

Challenges of Using Traditional Infrastructure

Cloud providers offer many resources for building intelligent applications, but what about those among us who are still deploying on physical machines on-premises? Bare metal server deployments, which do not use virtualization, are great for eliminating the noisy neighbor problem of shared resources, but there are significant limitations to consider. In this section, we'll go over some of the most common infrastructure issues and how Kubernetes can help resolve them.

As you learned in Chapter 1, containers are a lightweight deployment mechanism at the core of Kubernetes, and they provide you with the ability to develop rapidly and update frequently. This speed of innovation is critical for intelligent applications. Kubernetes is perfect for orchestrating these updates because all major cloud providers support it, and it can easily be deployed in on-premises infrastructure as well.

Let's take a look at the specific challenges of traditional infrastructure and walk through how Kubernetes solves them:

- What if we need periodic updates of the latest AI and machine learning tools?
- What if we need to scale up or scale out?
- What if we need to make the most of our resources?
- What if our data is too sensitive for the cloud?
- What if cloud GPU computing is too expensive?
- What if we need to understand how models make decisions?
- What if we want to use multiple cloud vendors?

What if we need periodic updates of the latest AI and machine learning tools?

Toolkits for building intelligent applications are evolving at a breakneck pace. Data scientists will need to rapidly increment software versions to access new capabilities. This need poses a significant challenge for traditional system deployments because they usually require manual updates of software packages. Complex package dependencies make it risky to manage the frequency of updates—and the versions deployed on the server very rarely look the same as the development environment in which the application was created.

With Kubernetes, container images can be updated to point to the latest versions of toolkits with minimal effort. Given the ability to test applications on a local machine in a similar environment to production, developers can ensure the new updates run smoothly. If a problem does arise with a new image, the container image can be reverted quickly.

What if we need to scale up or scale out?

Using the conventional approach of deploying a single Linux host to run your intelligent application means that you need to fully understand the types of workloads that will be running as well as anticipate the amount of input and output data. If your workload outgrows the resources of your server, a physical upgrade will be required. You'll then be stuck submitting a ticket to your IT department, waiting for it to work its way through the system, and incurring downtime when it is installed. Likewise, if you want a new instance or instances to load-balance an existing application, you'll have a similar provisioning headache.

Just as virtual machine orchestration platforms make scaling up easy for VMs, Kubernetes makes it easy for containers. With everything running as a container, a constrained container can easily be swapped out with a beefier one (as long as the underlying physical hardware has enough resources). Scaling out is just as easy. New containers can be added to a cluster with a quick revision of YAML, a markup language used to define resources in a cluster. Kubernetes even handles the networking for us, ensuring that our new container is load-balanced automatically.

What if we need to make the most of our resources?

Virtual machines are isolated instances that contain an entire operating system. If you have an intelligent application that needs to scale out to thousands of virtual machines, that means the virtualization overhead is multiplied a thousand times. If the stronger isolation of virtualization is not necessitated, this overhead amounts to a tremendous amount of waste. Worse yet, if your application is a light microservice, the amount of resources consumed by virtualization can outstrip those used for the microservice!

Kubernetes helps organizations circumvent this tax by gaining the positive benefits of isolating workloads. Containers share the same kernel, and you do not need to spin up a new VM for each running instance of a server or job. You can therefore run much more on the same hardware than you previously could.

What if our data is too sensitive for the cloud?

Not all data is meant for the cloud, and more importantly, not all data is legally allowed to be stored in the cloud. Depending on the industry and region, there can be strict government regulations preventing data from leaving its country of origin. Even if your business does not find itself subject to such regulatory regimes, there are times when storing massive amounts of sensitive personal data in a cloud provider is not preferred.

The naive approach is to host the data in a local data center and use the cloud services remotely. This creates two challenges. First, IT departments must allow the cloud services access to secure internal data centers with costly network pipes. Second, any data accessed by the cloud services must be transferred over the network, leading to suboptimal throughput and latency. Both issues are very difficult to overcome and are often discouraged by IT security.

With Kubernetes, you can develop your applications once and port them to any environment on any infrastructure (as seen in Figure 2-4). Instead of bringing your data to where it's needed, you can invert the normal procedure by bringing the platform and application containers to the data.

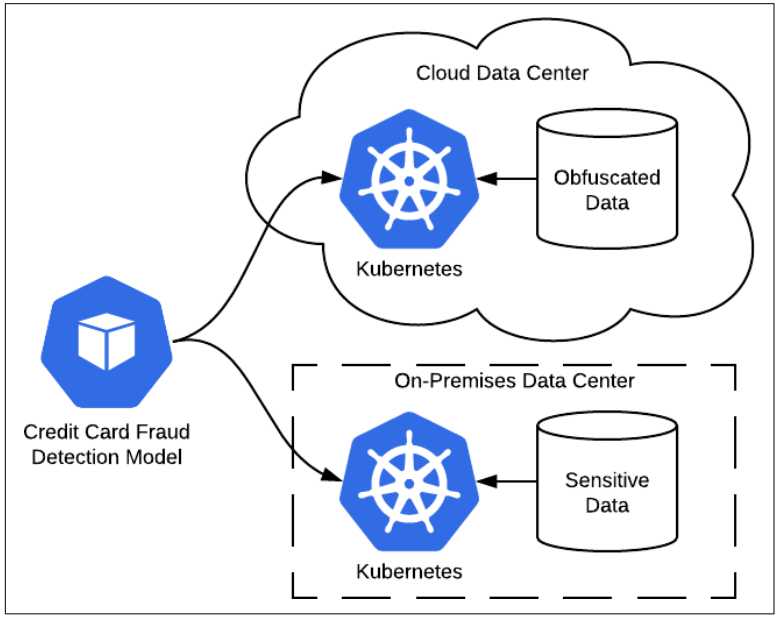


Figure 2-4. Deploying the same model in the cloud and on-premises

What if cloud GPU computing is too expensive?

Renting a single V100 GPU on one of the most popular cloud providers could cost up to $3.06/hour as of November 2019. That's almost $27,000 a year! In contrast, a V100 GPU can be readily purchased online for less than $10,000. Using these in one's own data center could provide similar utility at a fraction of the cost.

If you want a more cost-effective model for running intelligent applications that leverage specialized hardware such as GPUs, Kubernetes can provide that flexibility. You can add new GPU-enabled nodes to a cluster and share them among the data scientists rather than renting expensive hardware in the cloud. GPU manufacturers such as NVIDIA have collaborated with the Kubernetes community to bring GPU acceleration to container workloads.

What if we need to understand how models make decisions?

Every major cloud provider now has a set of cognitive services for its users. These typically include a plethora of use cases such as natural language processing (NLP), translation services, image/video recognition, and more. For reasons such as legal audits and compliance, understanding how decisions are made is critical. Given the black box nature of private cloud services, it is often very challenging to answer those questions. Running our own applications on Kubernetes can help provide an audit trail that can be followed when we need to peel back the layers of how an intelligent application arrived at a particular decision so that we can remediate issues.

Running your cognitive services on Kubernetes means you have the freedom to deploy and monitor your own intelligent applications. With access to the source code, your engineers can learn how any model makes its decisions and modify the code if necessary. Even more critical is that you have access to the data used to train your models and understand their outputs. This allows you to reproduce your results and explain how decisions in the models were made.

What if we want to use multiple cloud vendors?

Cloud service providers don't all offer the same cognitive services. Likewise, their interfaces to communicate with each offering are different. This could lead to vendor lock-in. If a vendor changes fea-

tures, pricing, or hardware options, moving to another cloud will be costly and will require additional development work. Likewise, just because you're deploying on-premises doesn't mean you don't want to use cloud services for some things. How, then, do you bridge that gap with applications you have on-premises?

Developing on Kubernetes means you're building applications and APIs that are independent of the cloud in which they run. You are in control of the APIs for your intelligent applications. There are a growing number of open source projects, such as this text sentiment classifier from IBM, that can be deployed to your Kubernetes cluster as a microservice to help get you started quickly. This frees you from having to rely on black box models from managed services in the cloud.

The Hybrid and Multicloud Scenario

Intelligent applications are data hungry, and users must feed them frequently. In a utopia for data scientists, all company data would exist in the same infrastructure, but most enterprise organizations don't have all of their data in one place. Often this is by design—if one cloud vendor or data center has problems, workloads can be spun up in other environments to offset the issues. At other times, highly dispersed data is due to organic growth in a company. For instance, one set of developers may prefer to prototype and develop in Azure, while another set prefers Amazon Web Services (AWS). Yet another group may have been given a mandate that all applications must be built internally due to regulatory compliance, ensuring that they cannot use *any* cloud.

If you already have data stored in multiple places, Kubernetes is a great way to build applications that are portable across multiple data centers. Since each container image is exactly the same and Kubernetes manifests provide a reproducible specification for how the applications are meant to be run, the same application can easily be migrated to Kubernetes clusters running in distinct data centers.

In Figure 2-5, you can see an example of deploying across different environments. Data scientists train models using sensitive data that is not legally allowed to leave their private cloud. The model is then tested as a microservice in the same private cloud and deployed to production in the *public* cloud. Sensitive data never leaves the safety of the company's private infrastructure, but derivative results such as

classifiers can still be used to power applications running in the public cloud.

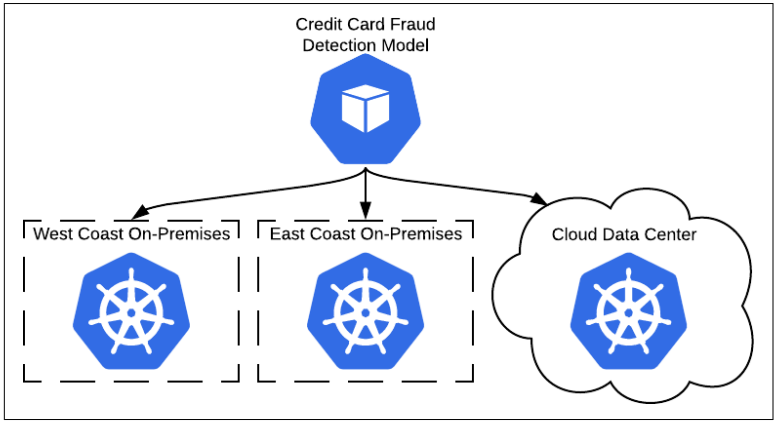


Figure 2-5. Deploying a model to multiple data centers

Kubernetes probably sounds great, but what if you already have workloads running in the cloud or are using a cloud provider's cognitive services? Does that mean you can't use Kubernetes? Not at all. You can still build the core parts of your application in Kubernetes and take advantage of cloud services.

Kubernetes Is a Great Choice for Intelligent Applications

Given what we've outlined here, you might be excited to get started with Kubernetes. There are several open source projects to help you jump-start Kubernetes data science initiatives. One of the most popular ones is Kubeflow, a Kubernetes-based workflow for machine learning. Kubeflow has an excellent community and includes tools for experimentation, model serving, pipeline orchestration, and more. Another project, Open Data Hub, is an open source AI-as-a-service platform that includes tools to help with building an even broader scope of intelligent applications than Kubeflow, including application monitoring, data governance, and data ingestion. The community's primary goal is to provide a blueprint for how to build an enterprise-grade platform for intelligent applications using popular open source tools, including components of Kubeflow.

While there's no one-size-fits-all solution for any problem, Kubernetes is a great choice for running intelligent applications in both

the cloud and on-premises. At Red Hat, we look forward to seeing Kubernetes continue to pay dividends on these use cases for our customers and for the open source community. The future looks bright for intelligent applications.

CHAPTER 3

Object, Block, and Streaming Storage with Kubernetes

How data is stored and processed will make a big difference to your team's quality of life. A well thought-out, optimized data pipeline makes teams more productive. While Kubernetes makes deploying new technologies easier, the storage systems we choose to rely on still remain a crucial determinant of long-term success. With many options available, it pays to be well informed on the trade-offs of each type of storage and how best to deliver persistent data services to a Kubernetes cluster in use.

In this chapter, we'll explain what object, block, and streaming storage are and how best to put them to use in data pipelines built with Kubernetes. Chapters 4 and 5 will explain in more detail how to tie these components together in practice with a Kubernetes architecture.

Object Storage

Object storage is one of the most essential types of storage for intelligent and cloud native applications. Many use cases rely on a storage system that offers a stable API for storing and retrieving data that can then be scaled massively to downstream clients who need the data. As seen in Figure 3-1, object storage is deployed into "buckets" (`ourcompany-bucket` in the diagram), which store objects in a flat namespace, though separators in the key such as `/` can allow end users to organize and interact with objects in a nested fashion.

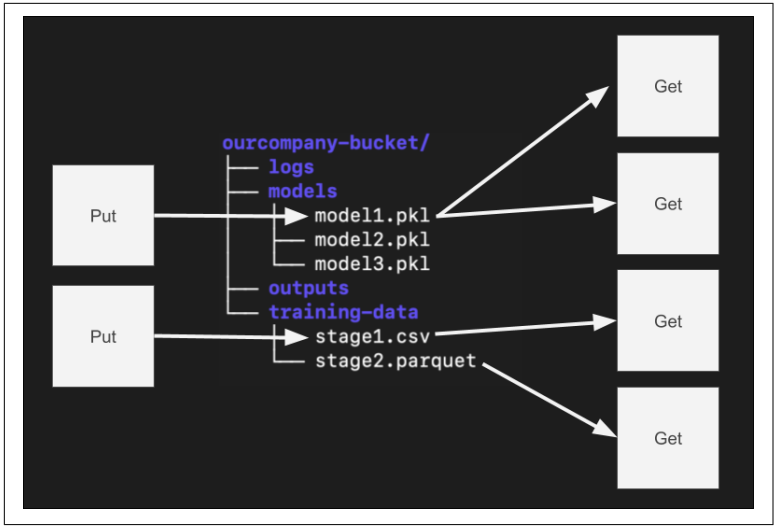


Figure 3-1. Object storage is used for static files and read-intensive workloads

Just as the filesystem on your laptop provides an API for reading and writing to files, object storage provides the same features at a massive scale over the network for a low cost. Common use cases for object storage include:

- Serving static files such as audio, video, and pictures
- Keeping logs or other files around for a long time
- Powering storage for container image registries

Why use object storage instead of having a server somewhere that users transfer files to and from? There are many reasons, such as massive scalability, resilience, and a predictable API—and often features such as versioning, tagging, and encryption. Not only is object storage designed to be reliable, but it also provides sophisticated capabilities for auditing data access: a comprehensive trail of who modified or accessed objects and when.

Object Storage for Intelligent Applications

Authors of intelligent applications are likely to lean heavily on object storage to power their pipeline. A full data pipeline will have many stages that use object storage as an interface for sourcing and sinking their inputs and outputs. Naturally, the data for training or

querying has to be stored somewhere in the first place, and object storage is a perfect place to dump data in whatever format you currently have it. Getting that data into a format that downstream consumers can understand can be taken care of later (and hopefully automated as part of a more developed pipeline).

Flashy technologies such as TensorFlow, an open source framework for building neural networks and other models, get a lot of attention in the AI community, but the truth is that a big chunk of data science work is cleaning up data and making it usable in the first place. Strings need to be munged, outliers need to be considered carefully, structure has to be parsed out of messy data, and so on. Some of this preprocessing work will be interactive and manual and might be a good fit for using tools like Jupyter to shuffle data back and forth, and Spark to perform iterative algorithms against an object data store

Other use cases will be automated and might involve several dependent stages that each feed their inputs into the next one. For instance, data might arrive in a format such as JSON, which is good for humans but inefficient for use by computers. Projects such as Apache Parquet serve as a much more efficient way to serialize data, and one step in a pipeline might well be to convert incoming data to this more efficient format to make consumption easier.

Another great example of this type of transformation stage is converting training data to a format such as TFRecord. TFRecord is a simple binary format optimized for use with training data in TensorFlow. Since data will be loaded into memory for training, converting the data to TFRecord, especially for use over the network, pays dividends.

For both manual and automated workloads, object stores provide an ideal format to ingress data from different sources. Once the data has been preprocessed, the fun can start. You can load data into systems such as Spark for querying, for instance. You can also train models and then pickle them, allowing you to load a classifier object that has been trained already during a future run of any program.

A Note on Consistency

Consistency, the "C" in the famous CAP theorem, is an important consideration for any user of an object storage system. For instance, Ceph is a strongly consistent storage system—when you write something to storage, there's a strong guarantee that any subsequent readers will not get a stale read. Ceph is also the storage system that provides file, block, and object storage to users, and this emphasis on consistency is necessary to achieve all three.

Other systems, such as S3, provide "eventual consistency"—they will *eventually* converge all readers to the same new value. In the interim, readers might get stale reads. Depending on your use case, a strongly consistent storage system like Ceph might well be a better fit—after all, it can be frustrating to build pipelines where downstream consumers need an accurate view of the output from their upstream producer.

These critical features make massive-scale object storage "just work" for every company with a credit card or some open source chops. New features such as sophisticated access controls continue to evolve on top of the existing offerings.

Shared APIs

The S3 storage service, created by AWS, has become a popular web service for object storage and is considered a standard for many object storage systems.

Ceph Object Storage, for example, incorporates S3 APIs, in addition to other popular APIs, to enable portability of workloads across various environments.

Just as Kubernetes enables portability for compute workloads, a common object storage API means that migrating or deploying applications across various environments becomes much easier.

Object storage is a key piece of the modern intelligent application. There are even offerings to query the data directly on the object storage of cloud providers without needing to spin up a server or a container. The public clouds will run these queries for you. If you're thinking of building an intelligent application, become familiar and fluent with the various capabilities and usage of object storage systems.

Block Storage

While block storage is not typically relevant for intelligent applications, you will almost certainly encounter SQL and NoSQL databases providing structured data for applications running in Kubernetes. Queries against this data are sometimes used by data scientists to build relevant data sets, and thus it is useful to provide an overview of the block protocol.

Block storage chops data into chunks of data and stores them as separate pieces. Each block of data is given a unique identifier, which allows a storage system to place the smaller pieces of data wherever is most convenient.

Because block storage doesn't rely on a single path to data, it can be retrieved quickly. Each block lives on its own and can be partitioned so it can be accessed in a different operating system, which gives the user freedom to configure their data. It's an efficient and reliable way to store data and is easy to use and manage.

While block storage does have its merits for certain use cases, such as databases, it is typically more expensive than object storage, making mass scalability prohibitive. It has limited capability to handle metadata, which means it needs to be dealt with in the application or database level, and is commonly mounted to the local filesystem of a container or virtual machine (Figure 3-2).

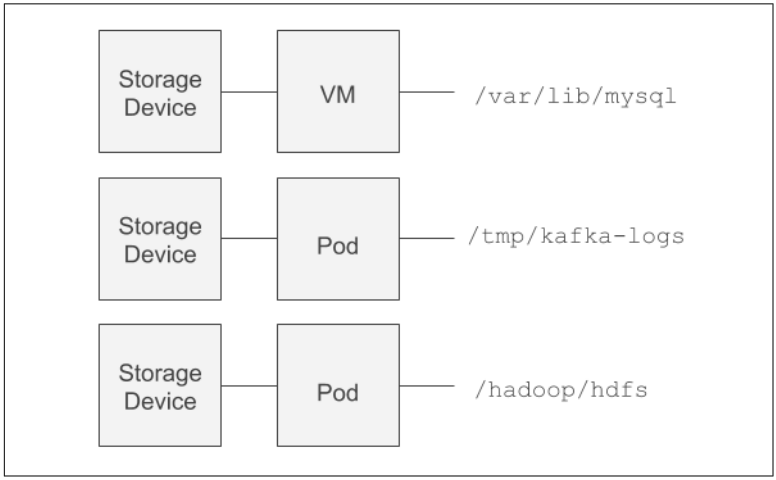


Figure 3-2. Block storage is fixed-size, often networked storage mounted in for use by compute workloads

Common uses for block storage include:

- Persisting data when rolling stateful virtual machines or containers such as databases
- Providing storage with special performance in cloud applications
- Easily snapshotting and recovering filesystems

Offerings such as AWS Elastic Block Store give users a powerful amount of flexibility in using various types of underlying storage for their applications. Often, "block storage" implies cloud block storage or a similar abstraction. If you are relying on storage, you need to be prepared for the possibility of many kinds of failures in the storage system. Block storage abstractions help you bring a degree of stateful stability to applications that otherwise would be at risk of going haywire because of these failures, which can even be unleashed deliberately using systems like Netflix's Chaos Monkey to ensure system reliability. Cloud block storage opens up a variety of colorful possibilities. Storage can be created, destroyed, and updated programmatically. Such programmatic access not only makes managing the storage easier for operators and for Kubernetes but also means that it can be tracked and versioned in an infrastructure tool such as Terraform.

Cloud block storage has a variety of options for configuration and fine tuning—different backing storage types, such as fast solid-state drives, are available to match differing use cases. Encrypted cloud block storage lives independently of the lifetime of any particular server, so it can be detached from servers and attached to new ones at will, making upgrades, experimentation, and provisioning infrastructure easier. In cloud workloads using block storage, even if a particular server's functioning is trashed beyond repair, its underlying block storage can be disconnected and reconnected to a pristine server, thus saving the day.

Cloud block storage can also easily be snapshotted and backed up directly using the cloud provider's APIs and tools—features that are critical for the modern enterprise, where a loss of user or compliance data could be disastrous. These snapshotting features make creating golden virtual machine images and protecting your organization from data loss approachable, compared to running your own storage hardware.

Block storage is often referenced in Kubernetes with the `PersistentVolume` resource, "claimed" by pods for their own usage with a `PersistentVolumeClaim`, and sometimes created on the fly for apps using dynamic volume provisioning. We'll get more into details and examples of this in practice in "Object Storage in Kubernetes" on page 35.

Streaming Storage

While object and block storage are good building blocks for intelligent applications, sometimes higher-level abstractions are needed. One need that has emerged is dealing with streams—the voluminous data points that are constantly streaming into our systems. While it's great that systems like Hadoop can do large-scale computation across existing data, multiple pipeline steps take a while to generate results, and sometimes you can't wait.

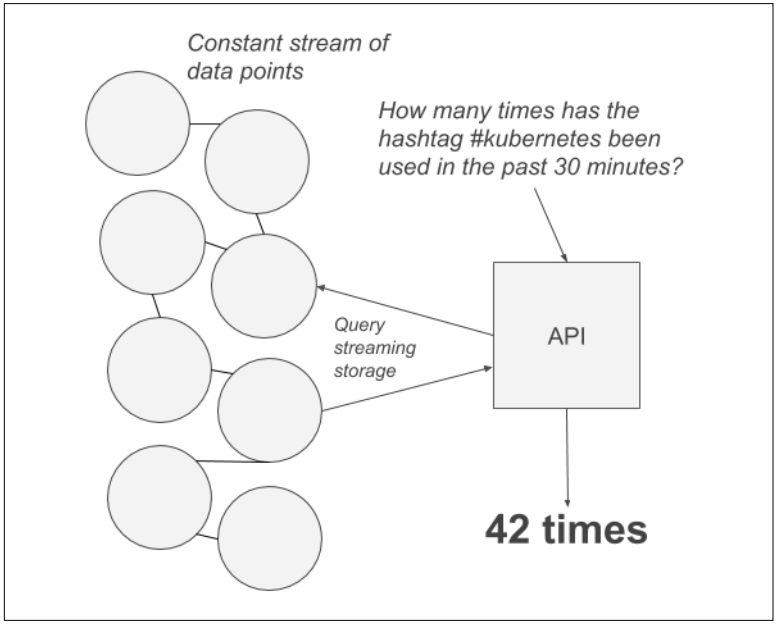


Figure 3-3. Streaming storage is used for real-time data processing workloads

Instead, you might need to be able to get answers from a streaming storage engine on the fly. Streaming storage systems can transform, filter, and query data *as it's coming in* without needing any inter-

mediate steps (Figure 3-3). That can help you build resilient systems that answer questions in real time. Systems such as Apache Flink, Kafka Stream, and Spark Streaming are examples of open source streaming storage.

Common uses for streaming storage include:

- Rapid access to data without the need for preprocessing
- Performing transforms on data at the point of ingest
- Guaranteeing that modification of streaming events happens exactly once

Streaming Storage Resiliency

Most businesses are not direct experts on distributed systems nor storage—they know their customers, their employees, and their market. It's good luck, then, that the particular sophistication required for effective streaming storage lets these businesses take advantage of the hard-won knowledge of hundreds of experts and open source contributors.

Instead of writing their own fault tolerance layer(s), they can instead more aggressively pursue business goals. Streaming allows users to perform otherwise complicated queries and transforms without having to worry about things like temporary node failure or a data transform getting applied more than once, causing incorrect results.

Processing Data at High Speed

In some cases, waiting a while to run analysis on your incoming data is just fine. Many use cases are not urgent and can wait patiently for the next hourly run of the cron job. Increasingly, however, businesses need to analyze large quantities of data, such as product recommendations for customers on an e-commerce website, in real time or close to it.

It's urgent in recommending products that customers receive the absolute best suggestions possible to increase revenue, and you might want a variety of factors to influence which things you recommend to them, such as what they have previously clicked on in their current session, and which products they bought recently, which items similar users have bought.

Streaming storage is perfect for these types of use cases due to its ability to compute large workloads in near real time. In Chapter 5, we'll take a more detailed look at producing recommendations in real time for users, including using Spark Streaming.

Using Storage in Kubernetes

Let's take a brief look at integrating the various types of storage with Kubernetes.

Object Storage in Kubernetes

Users of object storage in Kubernetes will have two primary concerns. One is ensuring an object store is available in the first place. The other is ensuring that clients have the correct permissions for bucket access.

In the case of the cloud, the first concern is taken care of for you, as every major cloud provider has some type of object storage system available. If running on-premises, a system such as Ceph can be deployed on top of Kubernetes. You can see some examples of how to deploy Ceph on Kubernetes in Ceph's documentation. Following the instructions there ensures that Ceph block devices are available for creation as a dynamic volume (due to the creation of the `ceph-rbd StorageClass`). This also installs the Ceph Object Storage Daemon, ensuring that Ceph Object Storage is available.

The permissions concern is usually addressed by using Kubernetes Secrets. Kubernetes Secrets allow administrators to store values such as object storage access keys in Kubernetes. These secrets are later injected into pods as environment variables or readable files. Secrets can be encrypted at rest to ensure that only the proper consumers can decrypt and use them.

Block Storage in Kubernetes

Block storage in Kubernetes, as noted previously, can usually be accessed through the use of Kubernetes Volumes. Kubernetes out of the box supports a huge array of different volume options, including Network File System (NFS), local volumes circumventing the layered container filesystem, cloud provider volumes such as Elastic Block Storage, and more. Thanks to the magic of open source, there's a good likelihood that if you're using a popular platform,

support for accessing block storage there has an integration written for it already.

A Kubernetes `PersistentVolume` using this block storage can either exist already or be provisioned dynamically by Kubernetes as needed. The former might be a good fit for a team that knows its storage needs well ahead of time. Such a team might mandate that infrastructure be created in a declarative tool like Terraform. Administrators would provision the volumes themselves ahead of time, manage their life cycle using direct tooling instead of through Kubernetes, and grant Kubernetes pods access to them using a `PersistentVolumeClaim`.

Dynamic volume provisioning, on the other hand, allows Kubernetes to take the wheel and create devices for volumes as they are needed. This could be a better fit for a team that isn't quite sure what its storage and volume requirements will look like ahead of time, or a team that wants to enable easy provisioning of such resources for experimentation. In this model, any user with the correct access to the Kubernetes cluster could provision storage without needing to think about the underlying APIs too much.

Streaming Storage in Kubernetes

Streaming storage in Kubernetes will likely be utilized as an application deployed on top of Kubernetes. In Chapter 5, we'll see the concepts to build out such a system in practice. For directly stateful applications such as Kafka, users should make use of Kubernetes StatefulSets as well as building on top of a flexible underlying block storage layer, if offered.

Summary

This tour of storage should help set you up to understand the concepts for pipeline architecture and practical examples we'll cover in the next two chapters. As you can see, there's a lot of detail to take in when it comes to storage, but an effective understanding of what's available will help your team achieve its goals and avoid pitfalls. Kubernetes lets you experiment with many different storage offerings so you can find the right fit.

CHAPTER 4

Platform Architecture

This chapter will dive deeper into the details of what it means to build out a platform architecture that enables organizations to create applications, run them in an optimized infrastructure, and efficiently manage application life cycles. For data pipelines, this includes ingestion, storage, processing, and monitoring services on Kubernetes.

In this chapter we'll cover the container orchestration, software, and hardware layers needed to create a complete platform that eliminates data silos and enables workflows. It will serve as a general architecture overview. Chapter 5 will cover specific examples that would be implemented on such an architecture and how they might be deployed.

Hardware and Machine Layers

Figure 4-1 shows the architecture for OpenShift, an enterprise Kubernetes distribution. We won't go into detail about all of the layers shown in this diagram—detailed information can be found in the OpenShift documentation. We will use the OpenShift architecture to highlight some key Kubernetes layers that play a critical role in the platform architecture for data pipelines.

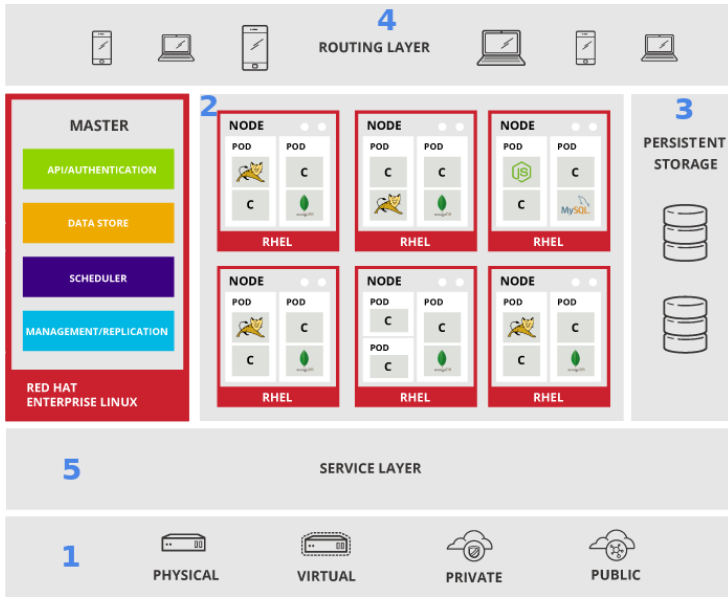


Figure 4-1. OpenShift architecture

Hardware

Layer 1 describes the type of hardware in which Kubernetes can be deployed. If you have an on-premise data center, this would be your bare metal hardware or virtual servers. If you're deploying to the public or private cloud, you may choose AWS EC2 instances, Azure Virtual Machines, or other options.

One benefit of Kubernetes is the ease with which it allows you to combine hardware types. In our internal Red Hat data center, we run our on-premises workloads on a mixture of bare metal nodes and virtual servers in one Kubernetes cluster (see Figure 4-2). This allows more flexibility for processes that demand lower latency performance optimizations such as Elasticsearch on bare metal nodes that are backed by NVMe storage.

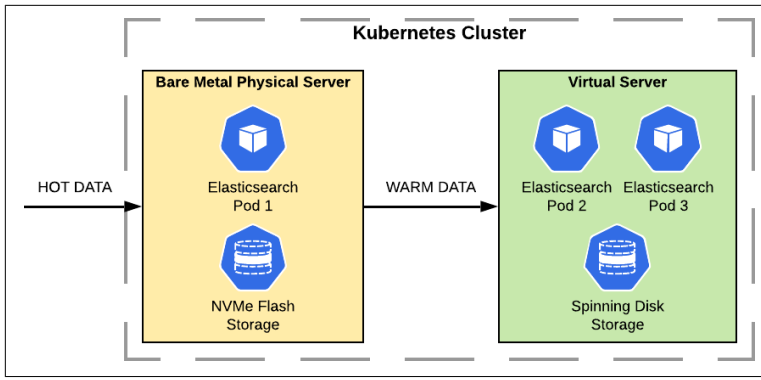


Figure 4-2. Mixing bare metal and virtual servers in Kubernetes

In Figure 4-2, *hot data* contains the most recent indexes, which demand high read and write throughputs due to the frequency of data coming in and of users querying the results. An example of hot data would be the last seven days of systems log data. *Warm data* might be systems logs older than seven days that are read-only and only occasionally queried. In this situation, spinning disks with higher latency and cheaper costs are sufficient.

Physical and Virtual Machines

Let's look next at the node layer (layer 2 in Figure 4-1. In Kubernetes, physical and virtual machines are mapped to nodes (Figure 4-3). In most instances, each node contains many pods that are each running specific container applications. For example, a single node can have a mashup of pods running Kafka brokers, Flask web applications, and application microservices.

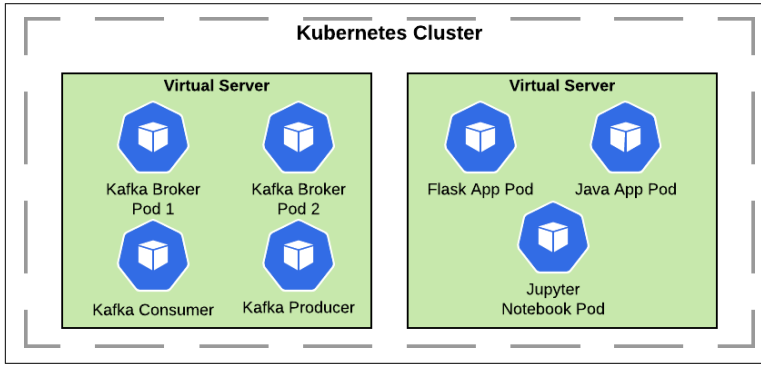


Figure 4-3. Running multiple types of apps on a single node

You can let the Kubernetes native scheduler take care of prioritizing the node in which an application should reside, or you can create a custom scheduler. (If you want to know about Kubernetes schedulers, you can learn more in the Kubernetes documentation.) We'll expand this layer a bit more to show how you can deploy applications and microservices to create a data processing pipeline.

Persistent Storage

Depending on the application you're running in a pod, you may need persistent storage. If you have a stateless application or model being served as a microservice, more than likely persistent storage isn't required. If you are running a database or a message bus or simply need workspace for an interactive online workbench, then layer 3 (fee Figure 4-1) is required. Each container in a pod can have one or more persistent storage volumes attached to it (see Figure 4-4). In the case of a database, you might have one for log files, another for temporary disk space, and yet another for the actual database data.

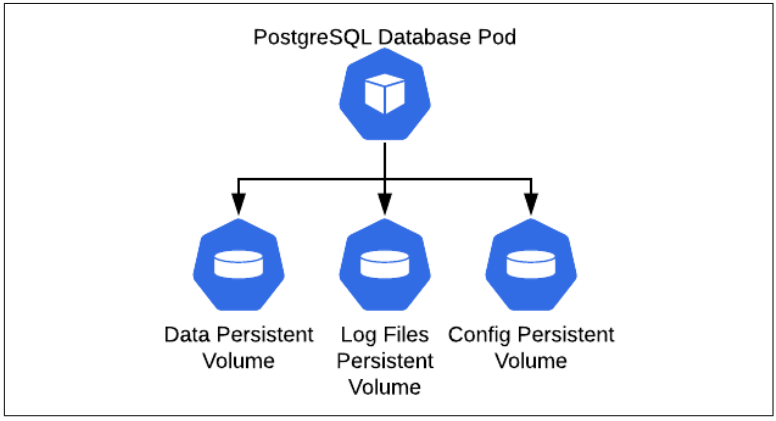


Figure 4-4. Persistent storage example for pods

As we discussed in Chapter 3, picking the right storage for your workloads can be very important. With Elasticsearch, it was critical that we deployed hot data to NVMe-backed nodes because of the extremely high volumes of reads and writes that are required. Slower disks would cause significant bottlenecks in performance and eventually lead to the Elasticsearch application nodes losing data due to lack of throughput (which we experienced firsthand).

Your typical web application PostgreSQL database doesn't need as much low-latency storage, so regular spinning disk storage might be fine. In Kubernetes, you can mix and match storage classes and request which one to use as part of the deployment configuration for applications.

Networking

Last, we have layers 4 and 5 in Figure 4-1, the routing (networking) and service layers (see Figure 4-5). These layers enable access to the applications running on the pods to external systems. The service layer provides load balancing across replicated pods in an application deployment. The routing layer gives developers the option to expose IP addresses to external clients, which is useful if you have a user interface or REST API endpoint that needs to be accessible to the outside world.

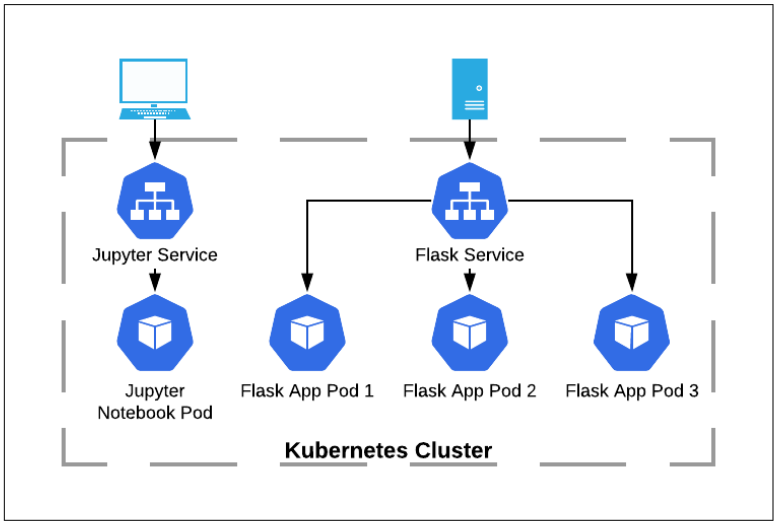


Figure 4-5. Networking and load balancing

If you want to learn more about Kubernetes architecture in general, visit Kubernetes Concepts. A deeper dive into OpenShift's architecture can be found here.

Data and Software Layers

Now that you know a bit more about the Kubernetes platform architecture, let's get into what it means for data. A good platform architecture should remove barriers to processing data efficiently. Data engineers need an environment that scales to handle the vast amounts of information flowing through their systems. Many intelligent applications are built with machine learning models that require low-latency access to data even as volumes grow to massive scales.

Hadoop Lambda Architecture

One of the more popular pipelines for low-latency processing of big data stores is the lambda architecture. With the increased popularity of streaming data, IoT devices, system logs, multiple data warehouses, and unstructured data lakes across organizations, new challenges have surfaced for keeping up with demands for analyzing so much information.

What if we wanted to analyze the sentiment of incoming tweets from a live event on Twitter? A first attempt at building an intelligent application for this might be a batch job that runs nightly. But what if key business decisions need to be made in minutes instead of days in order to react quickly during the event? Even if your nightly job can keep up as the volume of tweets increases over time, it still isn't capable of reacting in minutes.

With a lambda architecture, engineers can design applications that not only can react to incoming streams of data in real time but also can gain additional insights by running more detailed data processing in batch periodically. Various Hadoop distributions have become a fixture in many data centers because of their flexibility in providing such capabilities. If you were to use Hadoop to deploy a lambda architecture for sentiment analysis detection, it might look something like Figure 4-6.

In the Hadoop lambda architecture, applications are typically deployed on physical or virtual machines. Kafka and Spark Streaming are used to process tweets and analyze sentiments in real time. The results are then sent to an external system where actions can be

taken immediately. In this example, an Apache Cassandra database stores the results and is queried in real time.

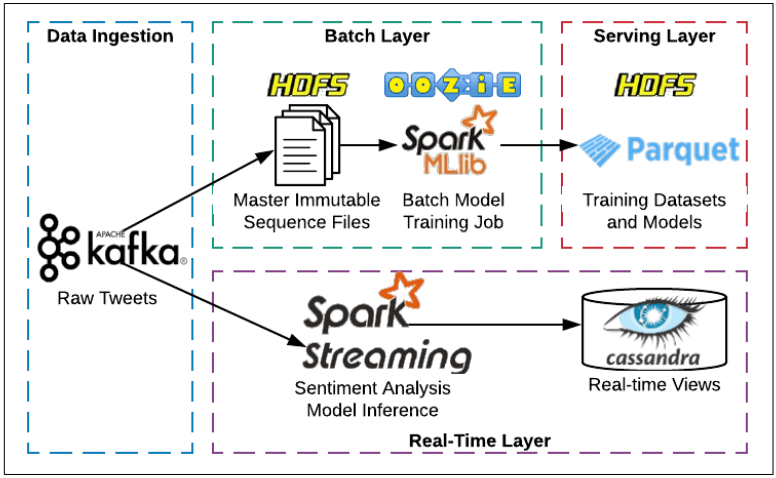


Figure 4-6. Lambda architecture for Twitter sentiment analysis using Hadoop

The batch pipeline of the architecture is used to train the sentiment analysis model periodically as new data comes in and to validate the changes against a set of known data points. This helps to ensure that the model maintains accuracy over time. The new model is then redeployed to the real-time pipeline, which is a Spark Streaming application in this case.

Now let's see what this same architecture looks like with Kubernetes.

Kubernetes Lambda Architecture

On the surface, these two implementations look very similar (see Figure 4-7), and that's a good thing. You get all the benefits of a lambda architecture with Kubernetes, just as you could if you ran these workloads on Hadoop in the public cloud or onto your own data center. Additionally, you get the benefit of Kubernetes orchestrating your container workloads.

One change from our two lambda example implementations is that an object store is used to collect data instead of HDFS. This allows for separating storage from compute in order to scale out the system hardware independently based on demand. For example, if you need more nodes due to increased traffic in Kafka, you can scale out

without impacting where the actual data might be stored. Another change is using containerized Kubernetes pods. This results in resources of the various services being managed and allocated on-demand both for real-time and batch pipelines.

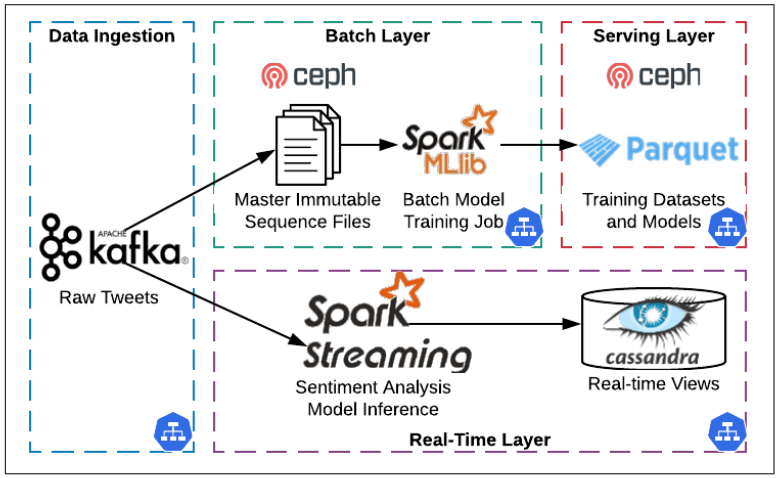


Figure 4-7. Lambda architecture for Twitter sentiment analysis using Kubernetes

Finally, since Kubernetes has orchestration built into it, we can use a cron job for simple model training instead of a workflow scheduler such as Apache Oozie. Here is an example of what the YAML definition for the training job might look like:

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: sentiment-analysis-train
spec:
  schedule: "0 22 * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: sentiment-analysis-train
            image: sentiment-analysis-train
            env:
            - name: MODEL_NAME
              value: "${MODEL_NAME}"
            - name: MODEL_VERSION
              value: "${MODEL_VERSION}"
            - name: NUM_STEPS
```

```
            value: "${NUM_STEPS}"
          restartPolicy: OnFailure
  parameters:
  - name: MODEL_NAME
    description: Name of the model to be trained
    value: SentimentAnalysis
  - name: MODEL_VERSION
    description: Version of the model to be trained
    value: "1"
  - name: NUM_STEPS
    description: Number of training steps
    value: "500"
```

For more information on `CronJob` objects, visit the documentation.

Each part of the pipeline is also deployed as a microservice. This includes the Kafka cluster, object store, Spark MLlib, Spark Streaming application, and Cassandra cluster. Peeking into the deployed pods on the nodes may reveal a workload orchestration similar to Figure 4-8.

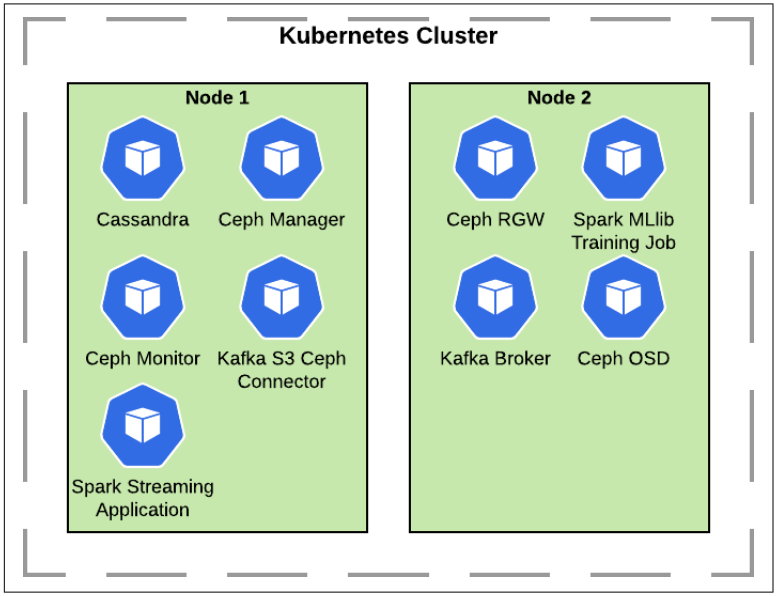


Figure 4-8. Example pod distrubution on Kubernetes

In a production Kubernetes cluster where performance is critically important, having your Kafka pods on the same node as the object store pods could create CPU or memory resource contentions during high volume workloads. To maximize the usage of your

nodes and minimize contention with applications of drastically different usage patterns, Kubernetes features such as nodeSelector and affinity and anti-affinity rules can be used to ensure only certain workloads are deployed to specific nodes indicated by labels. A better, more scalable orchestration might look like Figure 4-9.

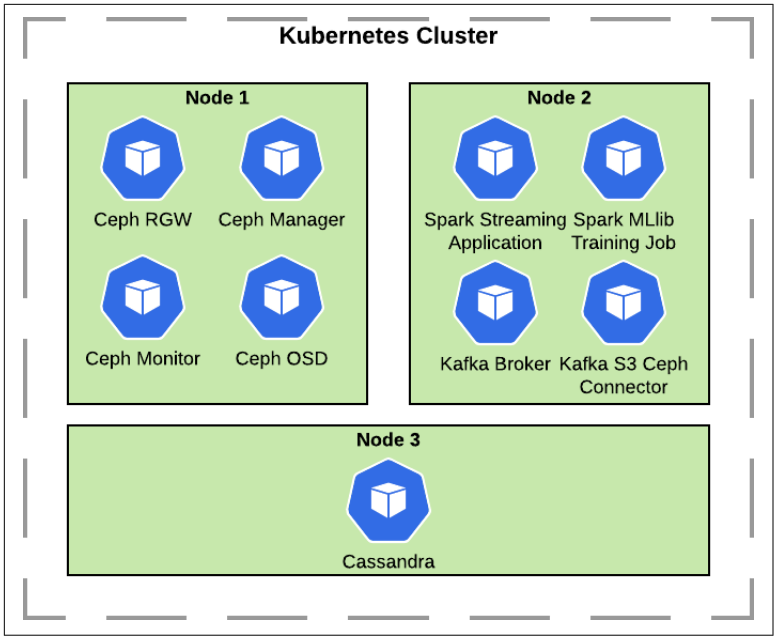


Figure 4-9. Assigning pods to nodes

You can read more about this in the documentation on pod assignment.

Summary

Not every architecture will look exactly the same, but some useful and reusable patterns will emerge in many places that look quite similar to one another. Some of the suggestions here should help you structure your architecture properly. If you have to make changes as you go, don't sweat it—one of the greatest advantages of deploying with Kubernetes is its flexibility.

In Chapter 5, we'll take a look at some specific examples that build on these architecture concepts to power intelligent applications.

CHAPTER 5
Example Pipeline Architectures

Let's take a look at some example architectures that illustrate concepts presented here, including:

- *Product recommendations* using Spark, including some resources to investigate for real-time recommendations with Spark Streaming
- *Payment processing* using Python's Flask and scikit-learn to detect fraud and other financial crimes
- A *site reliability* use case analyzing log data for anomalies using Elasticsearch, Logstash, and Kibana
- *Inference at the edge* using classifiers taking advantage of special Nvidia hardware at embedded locations for computer vision

With the examples outlined here, we will go over a few aspects. First, the use case: what problem are you trying to solve? As for the general architecture, how you will structure the various pieces of the data platform to achieve the desired end result? Last, we will look at concrete ideas for implementing the solution in Kubernetes.

Sample code and Kubernetes resource definitions (in YAML) have been provided to help guide implementers in the correct direction.

E-commerce: Product Recommendation

In e-commerce, recommendations are one of the most reliable ways for retailers to increase revenue. We can suggest items to users that they might buy based on data that we've observed in the past about their own behavior or the behavior of others. For instance, we can use collaborative filtering to recommend products to users based on the items that profiles similar to the user in question liked.

Normal batch processing using a system like Hadoop is one possible solution to this, but we could also use Spark, a unified analytics engine for large-scale data processing. Spark processes data in-memory and achieves high performance using a variety of optimizations. Spark supports a variety of client bindings in various languages, and data scientists can easily connect and tinker with their data embedded on a Spark cluster using tools like Jupyter. Spark also has support for *streaming analytics*; as noted in Chapter 4, this can help open up a variety of use cases that allow our intelligent applications to react to things more quickly than traditional batch processing. For instance, Netflix uses Spark Streaming to recommend content to users in near real time and to generate custom title screens that the model suspects will convince users to play a given piece of content.

This video presentation from Red Hat engineer Rui Vieira goes into some of the specific mathematical details of implementing least mean squares for collaborative filtering with Spark Streaming. Here, we'll look at how to implement such a system, including how you might want to structure a Spark deployment on Kubernetes.

Implementation

Our implementation will rely on Ceph Object Storage to retain information that will later be used for making predictions, on Spark Streaming for the predictions themselves, and on a Spring Boot app to relay these predictions to and from the user-facing front end (Figure 5-1).

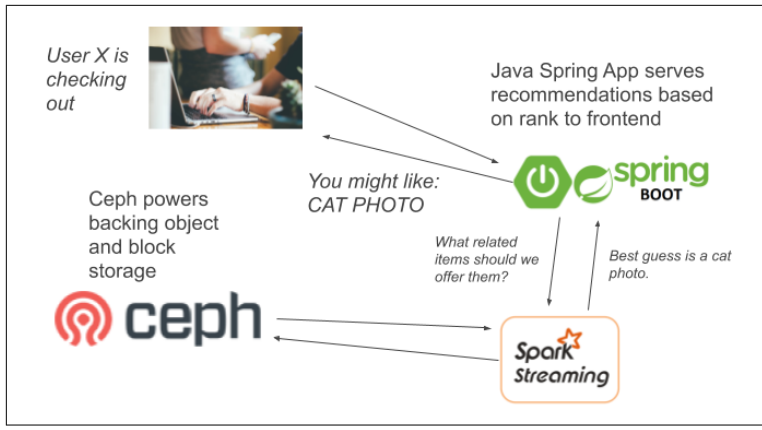


Figure 5-1. Spark and Spark Streaming can recommend items to users, based, for instance, on things that have happened recently

In normal (nonstreaming) Spark, we could use Spark's machine learning library (MLib) to train and use an alternating mean squares (AMS) model from data already stored in an S3-compatible object store, as in the following example based on Spark's documentation:

```
import java.io.Serializable;

import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.ml.evaluation.RegressionEvaluator;
import org.apache.spark.ml.recommendation.ALS;
import org.apache.spark.ml.recommendation.ALSModel;

public static class Rating implements Serializable {
  private int userId;
  private int productId;
  private float rating;
  private long timestamp;

  public Rating() {}

  public Rating(
    int userId,
    int productId,
    float rating,
    long timestamp
  ) {
    this.userId = userId;
    this.productId = productId;
    this.rating = rating;
    this.timestamp = timestamp;
  }
```

```
  public int getUserId() {
    return userId;
  }

  public int getProductId() {
    return productId;
  }

  public float getRating() {
    return rating;
  }

  public long getTimestamp() {
    return timestamp;
  }

  public static Rating parseRating(String str) {
    String[] fields = str.split("::");
    if (fields.length != 4) {
      throw new
      IllegalArgumentException("Each line must contain 4 fields");
    }
    int userId = Integer.parseInt(fields[0]);
    int productId = Integer.parseInt(fields[1]);
    float rating = Float.parseFloat(fields[2]);
    long timestamp = Long.parseLong(fields[3]);
    return new Rating(userId, productId, rating, timestamp);
  }
}

JavaRDD<Rating> ratingsRDD = spark
  .read()
  .textFile("s3a://ourco-product-recs/latest.txt") ❶
  .javaRDD()
  .map(Rating::parseRating);

Dataset<Row> ratings =
  spark.createDataFrame(ratingsRDD, Rating.class);
Dataset<Row>[] splits =
  ratings.randomSplit(new double[]{0.8, 0.2}); ❷
Dataset<Row> training = splits[0];

ALS als = new ALS()
  .setMaxIter(5)
  .setRegParam(0.01)
  .setUserCol("userId")
  .setItemCol("productId")
  .setRatingCol("rating");
ALSModel model = als.fit(training);
```

```
model.setColdStartStrategy("drop"); ❸

final int NUM_RECOMMENDATIONS = 5;
Dataset<Row> userRecs = model.recommendForUserSubset(
  activelyShoppingUsers,
  NUM_RECOMMENDATIONS
); ❹
```

❶ Note the use of `s3a://` protocol—this signals to Spark that it should load an object directly from an object store rather than from the local filesystem.

❷ We split the training data so we can test our model on data not included in the original training data once it's finished.

❸ This will drop any rows in the dataframe that contain unhelpful `NaN` values.

❹ `activelyShoppingUsers` is a Spark `Dataset<Row>` object (whose creation is left out for brevity) representing the subset of users who are currently on the site and hence that we want to produce predictions for.

Spark's `ALSModel` class's `recommendForUserSubset` method produces a `Dataset<Row>` describing the top production recommendations for each user. We don't have to stop there. We can also implement a system using a streaming-based algorithm such as the one in this article from Rui Vieira, who goes into fantastic detail about the math behind it. Not only could we use new product recommendations quickly after they come in, but we could also consider incorporating other types of data, such as what the user has viewed recently, as factors for our model.

For those interested in learning more or implementing such a model, Project Jiminy is worth a look. Project Jiminy is a ready-to-deploy solution for product recommendations that can run on OpenShift. It has comprehensive considerations for a production solution, including serving the resulting predictor(s). In Figure 5-2, "Analytics" refers to the backing Spark cluster.

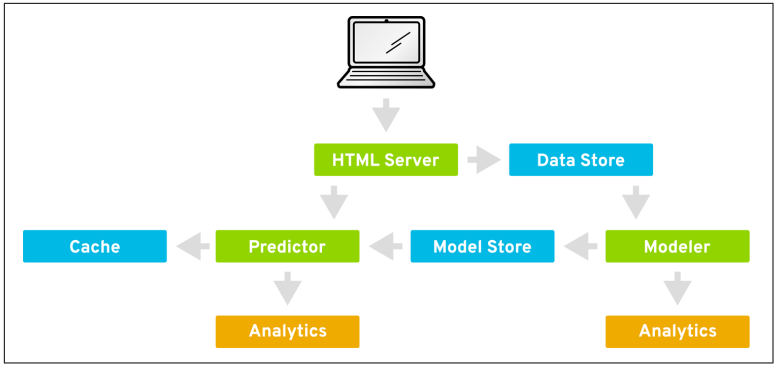


Figure 5-2. Architecture of Project Jiminy; various components such as the modeler and predictor are split into multiple microservices with REST endpoints

Kubernetes Ideas

We could structure a Spark cluster on Kubernetes as:

- A Spark master `Deployment` to manage the worker nodes
- A Spark worker `Deployment` for replicating worker nodes so we can have as many workers as we need
- A `Service` exposing the Spark master for the workers to connect to

Since Spark runs in-memory, we don't need to concern ourselves with using a `StatefulSet`, as data will be loaded into the cluster as needed. This can come from a variety of sources such as object storage, as demonstrated above.

The deployment for the Spark master would look like this. When the pod starts up, we kick things off by writing the proper hostname to `/etc/hosts` and invoking `/opt/spark/bin/spark-class org.apache.spark.deploy.master.Master`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: spark
    component: master
    team: datascience
  name: spark-master
spec:
```

```
  selector:
    matchLabels:
      component: master
  template:
    metadata:
      labels:
        app: spark
        component: master
    spec:
      containers:
      - args:
        - echo $(hostname -i) spark-master >> /etc/hosts;
          /opt/spark/bin/spark-class
          org.apache.spark.deploy.master.Master
        command:
        - /bin/sh
        - -c
        env:
        - name: SPARK_DAEMON_MEMORY
          value: 1g
        - name: SPARK_MASTER_HOST
          value: spark-master
        - name: SPARK_MASTER_PORT
          value: "7077"
        - name: SPARK_MASTER_WEBUI_PORT
          value: "8080"
        image: k8s.gcr.io/spark:1.5.1_v3
        name: spark-master
        ports:
        - containerPort: 7077
          protocol: TCP
        - containerPort: 8080
          protocol: TCP
        resources:
          requests:
            cpu: 100m
            memory: 512Mi
```

To expose this master internally to the Kubernetes cluster, we can create a Kubernetes `Service` for it. Once the `Service` is present, and assuming a proper Kubernetes DNS setup, we can access it at any time at the URI `spark-master`:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: spark
    component: master
  name: spark-master
spec:
```

```
  ports:
  - port: 7077
    protocol: TCP
    targetPort: 7077
  selector:
    component: spark-master
  type: ClusterIP
```

With that set up, we can use a `Deployment` that will create an arbitrary number of Spark worker pods connected to the master:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    component: worker
    release: sparky
    team: datascience
  name: spark-worker
spec:
  replicas: 3 ❶
  selector:
    matchLabels:
      component: worker
  template:
    metadata:
      labels:
        component: worker
    spec:
      containers:
      - command:
        - /opt/spark/bin/spark-class
        - org.apache.spark.deploy.worker.Worker
        - spark://spark-master:7077 ❷
        env:
        - name: SPARK_DAEMON_MEMORY
          value: 1g
        - name: SPARK_WORKER_MEMORY
          value: 1g
        - name: SPARK_WORKER_WEBUI_PORT
          value: "8080"
        image: k8s.gcr.io/spark:1.5.1_v3
        imagePullPolicy: IfNotPresent
        name: sparky-worker
        ports:
        - containerPort: 8081
          protocol: TCP
        resources:
          requests:
            cpu: 100m
            memory: 512Mi
```

❶ The `replicas` key lets us define how many copies of the Spark workers pod we will deploy. If we need more Spark workers, we can edit the manifest to increase this value and run `kubectl apply` on the cluster again.

❷ The worker connects to the created `spark-master` service using the `spark://` protocol.

We can check the status of our deployed cluster using the Spark built-in dashboard (Figure 5-3).

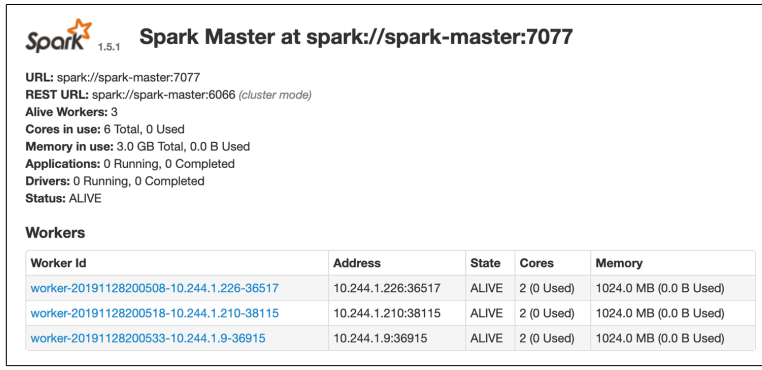


Figure 5-3. Once our Spark cluster is up and running, we can check the dashboard to see what's going on

Kubernetes can support all sorts of interesting Spark use cases—for example we could use Horizontal Pod Autoscaling to add new workers as the cluster begins consuming too much CPU. We could also deploy Spark clients such as Zeppelin and Jupyter to connect and start interacting with the Spark cluster.

Payment Processing: Detecting Fraud

Let's look at a Flask app serving a scikit-learn model that detects whether or not a given transaction is fraud. Flask is a Python microframework that makes building out small APIs and web apps refreshingly simple. New data about fraud will be coming in regularly, so we need to continuously retrain our classifier. We might also want the option to specify which version of the classifier should be used in production when we deploy the service. Kubernetes can help us address these needs and more, and we will look at some ideas for how to structure our deployment on it.

Implementation

As new training for classifiers happens, new versions of the classifiers will be serialized and stored in an object store such as AWS S3. Our Flask app will then download serialized scikit-learn models when it starts up. This service could then be exposed to clients in the cluster, or to the outside world, using a Kubernetes Service. This is illustrated in Figure 5-4.

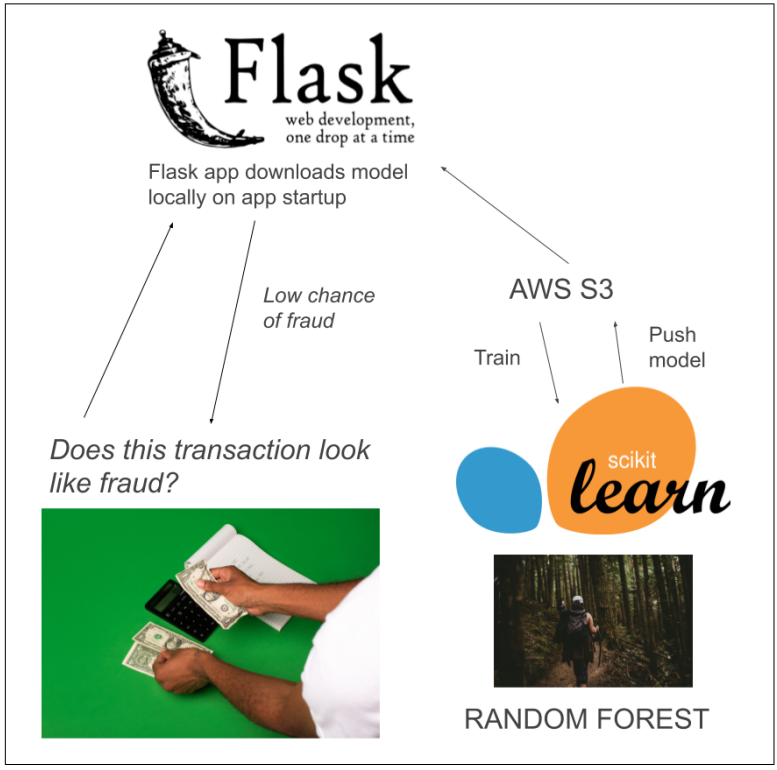


Figure 5-4. A Python Flask app can download models trained by a cron job so it can classify fraud

Random forests, which are available out of the box in the Python library scikit-learn, are a good fit for many machine learning use cases. In this example, the model is pretrained in the background using scikit-learn, and the saved result is pushed to S3. When the application starts up, it downloads the model from S3 and loads it into memory as a Python object. Code for the API would be along these lines:

```
import tempfile
import boto3
import joblib
import os
from flask import Flask, escape, request, jsonify

clf = None
with tempfile.TemporaryFile() as fp:
    s3_resource.download_fileobj(
        Fileobj=fp,
        Bucket='myco-ff-clf',
        Key=f'{os.getenv('FF_CLF_VERSION')}.pkl',
    )
    fp.seek(0)
    clf = joblib.load(fp)

api = Flask(__name__)

@api.route('/healthz')
def alive():
    return 'Healthy'

@api.route('/classify')
def classify():
    # Data is passed to the API as features in the query string,
    # like so:
    #
    # GET /classify?f1=0.3&f2=0.6&f3=0.9
    txn_amount = float(request.args.get('f1'))
    amount_spent_past_mo = float(request.args.get('f2'))
    txns_same_merchant = float(request.args.get('f3'))
    return jsonify(list(clf.predict([[
        txn_amount,
        amount_spent_past_mo,
        txns_same_merchant
    ]])))
```

Users of the API could issue an HTTP `GET` request to the service, passing in the features influencing the classification as query string parameters, such as to `fraud-model:3000/classify?f1=0.3&f2=0.4&f3=0.3`.

Kubernetes Ideas

We could have this API deployed in Kubernetes as a `Deployment` having an associated `Service` and configure a `CronJob` to do the training. The core will also expose a `Service` to allow clients within the Kubernetes cluster to access the API.

Here is a sample of a `Deployment` for the API. Note the use of environment variables to configure the containers, such as specifying the classifier version:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ff-api
  labels:
    app: ff-api
spec:
  replicas: 3
  selector:
    matchLabels:
      app: ff-api
  template:
    metadata:
      labels:
        app: ff-api
    spec:
      containers:
      - name: ff-api
        image: quay.io/mycompany/ff-api
        ports:
        - containerPort: 5000
        env:
        - name: FF_CLF_VERSION
        # magic value - change to test
        # different versions of classifier
          value: 42
        - name: ACCESS_KEY_ID
          valueFrom:
            secretKeyRef:
              name: access-creds
              key: access-key-id
        - name: SECRET_ACCESS_KEY
          valueFrom:
            secretKeyRef:
              name: access-creds
              key: secret-access-key
        livenessProbe:
          httpGet:
            path: /healthz
```

```
            port: 5000
            initialDelaySeconds: 5
            periodSeconds: 5
```

As you can see in the `FF_CLF_VERSION` environment variable setting above, we can configure which version of the classifier will be used for the service when it is deployed to production. This could be invaluable to you if you need to roll back to a previous version of a model, or to A/B test across different model versions.

Training with CronJob Objects

Given that we can specify the classifier version for our `Deployment` and access classifiers using a shared bucket, we can configure Kubernetes to train models on a schedule. For instance, the `CronJob` outlined below would call a training script at the 20th minute of every hour, which we could structure to train classifiers with newly published data, and upload the results to S3 as serialized Python objects when finished.

Using Kubernetes and the various monitoring tools it offers, including the venerable `kubectl logs` command, we can check up on things when they go wrong (as pods may stick around for a while after a failing execution):

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: classifier-trainer
spec:
  schedule: "20 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: classifier-trainer
            image: quay.io/mycompany/classifier-trainer
            args:
            - train.py
          restartPolicy: Never
```

Once you `kubectl apply` the YAML, the cron job will run on schedule. Members of the team will be able to observe the results as well as the abstraction that is generating them. Team members could even download results published by the job for testing and experimentation locally. As you can see, various pieces come together in Kubernetes to create the final polished pipeline.

Site Reliability: Log Anomaly Detection

Next, we'll look at an example architecture using machine learning and the ELK stack (short for Elasticsearch, Logstash, and Kibana) to identify and examine anomalies for rapid remediation. For instance, they might be Envoy access logs, or they might be logs related to the core functioning of the OpenStack cluster—logs emitted by services such as Cinder, Swift, Nova, and so on.

One challenge of logging what comes out of these services is that the volume makes it difficult to separate points of interest from the normal "vanilla" logs such as health checks that tend to fill up our log storage. Using Kibana's machine learning feature, we can more easily identify potentially alarming patterns emerging in our logs, as shown in Figure 5-5.

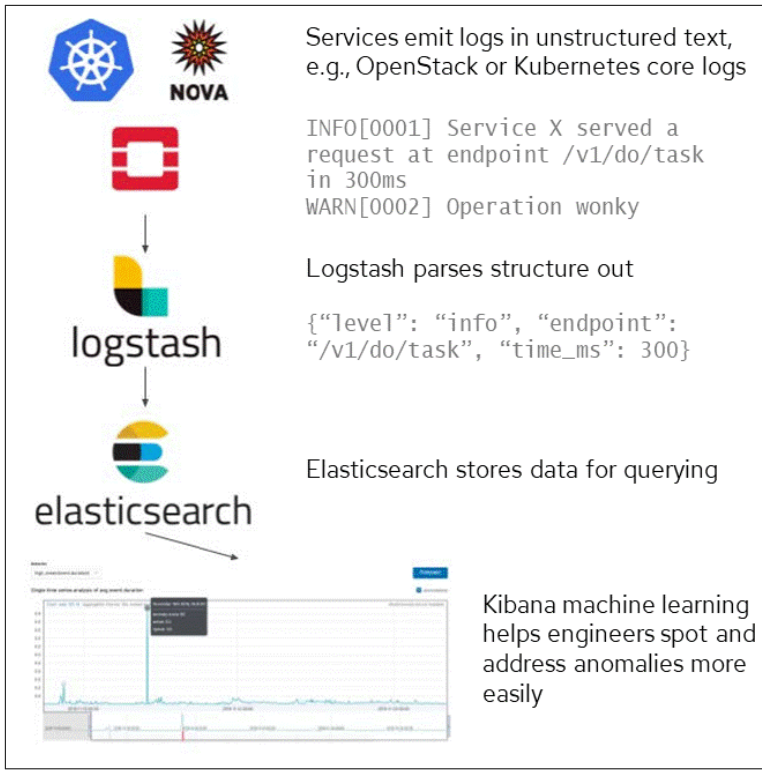


Figure 5-5. Using Logstash and Kibana powered by an Elasticsearch storage engine, we can spot out-of-the-ordinary activity in our logs

Implementation

Let's take Kubernetes API server logs as a starting point to examine how we might build out something similar to the ideas presented in this video about anomaly detection on OpenStack logs. All the core Kubernetes services, including the API server that responds to `kubectl` commands, spit out a truckload of unstructured text logs that look similar to the following truncated examples:

```
Caches are synced for APIServiceRegistrationController
OpenAPI AggregationController: Processing item
OpenAPI AggregationController: action for item : Nothing (rem
OpenAPI AggregationController: action for item k8s_internal_l
created PriorityClass system-node-critical with value
created PriorityClass system-cluster-critical with va
all system priority classes are created successfully
```

We can use the open source Filebeat to ship these logs to Logstash, a layer for collecting, parsing, and transforming logs. Logstash's capabilities allow us to *structure* the data coming out of our logs—instead of flat text, we can highlight properties such as the log level, associated IP addresses, which node of the cluster is acting up, and so on.

The Grok Debugger and Logstash syntax reference can help us in structuring our data (Figure 5-6). Formatting your logs in a machine-friendly format to begin with can help immensely in making this process easier.

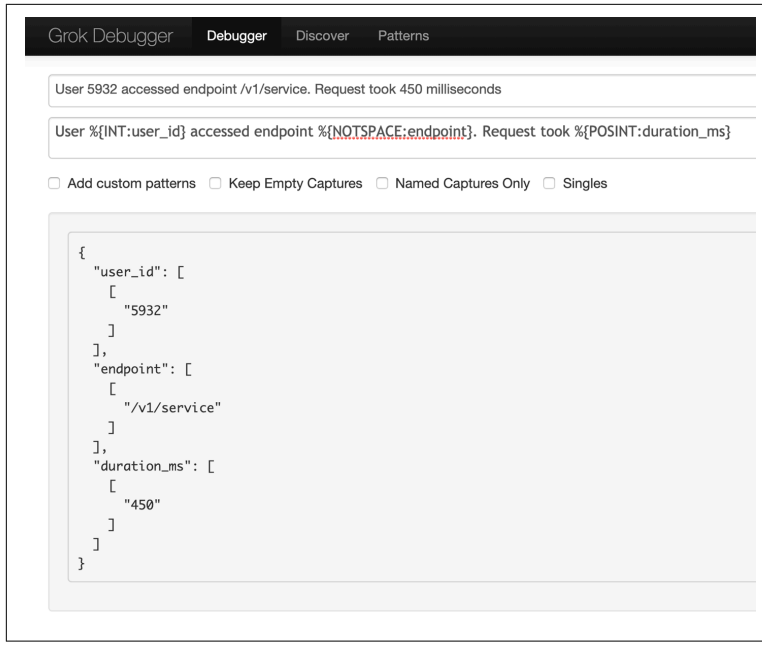


Figure 5-6. Using the Grok Debugger, we can workshop filters for parsing custom structure out of our logs

Structuring Data

A sample Logstash configuration to understand logs formatted with klog, the Kubernetes internal logging tool, is described below. It illustrates how Logstash's `grok` filter can be used to parse structure out of otherwise structured text. Logstash will transform `%{SYNTAX:NAME}` blocks into regular expressions that capture groups of structured text. As you can see below, if Logstash does not

understand the data within our strings as a syntax that it recognizes, we can also embed capturing groups from regular expressions directly:

```
input {
  # Filebeat sends Kubernetes API server logs to Logstash
  beats {
      port => "5044"
  }
}

filter {
  grok {
    match => {
      "message" => "(?<level>.)\
(?<DAYOFMONTH>\d\d)(?<MONTHDAY>\d\d) \
%{TIME:timeofday} *%{INT:thread_id} \
%{NOTSPACE:filename}:%{INT:line_number}] \
%{GREEDYDATA:msg}"
    }
  }
}

output {
  elasticsearch {
    # Elasticsearch is running as a Kubernetes StatefulSet
    # available via a Service using a CNI-compatible plug-in.
    hosts => ["http://ourcompany-es-service:9200"]

    # We will build time-based indexes so no index gets
    # too large.
    index => "{+YYYY.MM.dd}"
  }
  stdout {
    # If Logstash itself has errors, we can get some
    # debugging info.
    codec => rubydebug
  }
}
```

We can then see that a log such as this:

```
W1124 00:58:17.986343       1 lease.go:223] Resetting endpoints
for master service "kubernetes" to [192.168.65.3]
```

will be transformed to have structure such as the following before being ingested into Elasticsearch:

```
{
  "level": "W",
  "DAYOFMONTH": "11",
  "MONTHDAY": "24",
  "timeofday":, "00:58:17.986343"
  "HOUR": "00",
  "MINUTE": "58",
  "SECOND":, "17.986343"
  "thread_id": "1"
  "filename": "lease.go",
  "line_number": "223",
  "msg": "Resetting endpoints for master service ...."
}
```

This unlocks a new universe of creative possibilities. For one thing, we can explore logs by grouping and filtering on the new structured fields such as log level, filename, and so on. We could also parse further structure out of our logs by dissecting the catchall `msg` field, which could allow to us to extract, say, a `reset_master_service` field in the `WARN`-level log message used as an example above. Logstash, as you may note in the config, is configured to push the results to Elasticsearch, where we can then query them in a variety of ways, such as by using the open source Kibana tool.

Detecting Anomalies

In addition to simply querying our data, Kibana has features we can use for anomaly detection. For instance, Figure 5-7 shows how we can drill down on a particular field of interest that Kibana has identified as being likely to influence an anomaly. Using this feature, we can pinpoint when anomalies happened and dig further into logs with this feature and time interval for more insights.

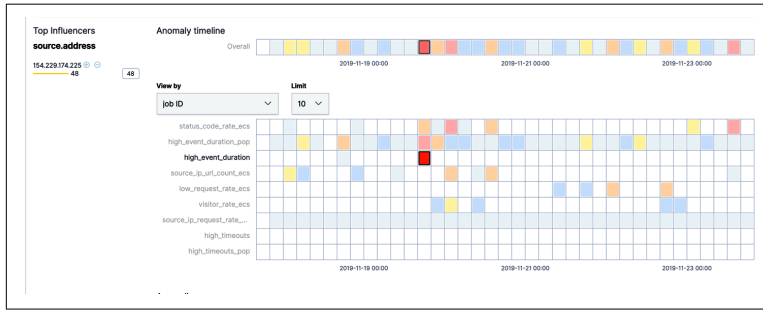


Figure 5-7. Kibana's machine learning feature can help us identify where anomalies are happening

We can also get a detailed view of which specific values of other fields influenced the field seeing the anomaly (Figure 5-8).

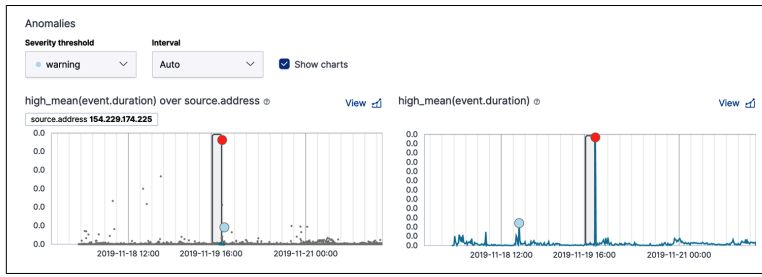


Figure 5-8. Kibana can draw visualizations of what was correlated with the anomaly

We can, of course, also directly query the data stored in Elasticsearch ourselves for other purposes. We could take this log data and add it to our favorite tool for more sophisticated analysis. Likewise, we could do post-hoc processing of the logs if we also store the raw data on other storage such as an object store. Logstash includes capabilities to perform such a "tee" function. We need only to add another `output` in the configuration discussed above.

Kubernetes Ideas

There are a variety of ways to implement such a stack on Kubernetes. The most important concern in this setup is likely to be getting a reliable implementation of the data engine (such as Elasticsearch) in place: due to its stateful and distributed nature, a sizable Elasticsearch cluster is unlikely to be a walk in the park to operate. Our options might include outsourcing maintenance for that piece to someone else, such as a cloud provider, but we also might need to deploy on premise, or we might need flexibility that out-of-the-box solutions do not provide. Let's review some ideas for deploying Elasticsearch (and Logstash and Kibana) on Kubernetes by looking at some of the resources created with Elasticsearch charts from Helm, an open source package manager for Kubernetes.

The best way to implement an Elasticsearch architecture is by using a `StatefulSet` for the master nodes and the data nodes of an administrated Elasticsearch cluster. We can see these with `kubectl get statefulsets`, and we can see that we have three master nodes

(to ensure high availability and reliability) as well as two data nodes to store the data and do the work assigned by the masters:

```
$ kubectl get statefulsets
NAME                              READY   AGE
logcluster-elasticsearch-data     2/2     14m
logcluster-elasticsearch-master   3/3     14m
```

Let's take a look under the hood at what these `StatefulSet` resources contain to get a feel for how this type of architecture is structured on Kubernetes with the `logcluster-elasticsearch-master` `StatefulSet`:

```
$ kubectl get \
  statefulset/logcluster-elasticsearch-master \
  -o yaml
```

Nested within the outermost `spec` key defining the actual guts of the `StatefulSet`, we see some properties that define metadata about the set as well as properties to help the Kubernetes scheduler understand how to run this app. We want three replicas in the `StatefulSet`, and we express a strong preference not to be scheduled on the same nodes as pods with the labels `app=elasticsearch`, `component=master`, or `release=logcluster`. This ensures that the other pods in the `StatefulSet` in question for this Elasticsearch cluster do not get scheduled on the same node (assuming that there are other nodes on the network for them to be scheduled on):

```
podManagementPolicy: OrderedReady
replicas: 3
template:
  spec:
    affinity:
      podAntiAffinity:
        preferredDuringSchedulingIgnoredDuringExecution:
        - podAffinityTerm:
            labelSelector:
              matchLabels:
                app: elasticsearch
                component: master
                release: logcluster
            topologyKey: kubernetes.io/hostname
          weight: 1
```

The actual `containers` definition for the master pods is of interest. We can see in its definition that we can tune options such as JVM heap size and resource claims, as we are also dynamically allocating

storage for the parts of the filesystem Elasticsearch expects to be persisted across pod runs:

```
    - env:
      - name: NODE_DATA
        value: "false"
      - name: DISCOVERY_SERVICE
        value: logcluster-elasticsearch-discovery
      - name: PROCESSORS
        valueFrom:
          resourceFieldRef:
            divisor: "0"
            resource: limits.cpu
      - name: ES_JAVA_OPTS ❶
        value: '-Djava.net.preferIPv4Stack=true -Xms512m -Xmx512m  '
      - name: MINIMUM_MASTER_NODES
        value: "2"
    image: docker.elastic.co/elasticsearch/elasticsearch-oss:6.8.2
    imagePullPolicy: IfNotPresent
    name: elasticsearch
    ports:
    - containerPort: 9300
      name: transport
      protocol: TCP
    readinessProbe:
      failureThreshold: 3
      httpGet:
        path: /_cluster/health?local=true
        port: 9200
        scheme: HTTP
      initialDelaySeconds: 5
      periodSeconds: 10
      successThreshold: 1
      timeoutSeconds: 1
    resources: ❷
      limits:
        cpu: "1"
      requests:
        cpu: 25m
        memory: 512Mi
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    volumeMounts: ❸
    - mountPath: /usr/share/elasticsearch/data
      name: data
    - mountPath: /usr/share/elasticsearch/config/elasticsearch.yml
      name: config
      subPath: elasticsearch.yml
```

❶ The end user of this image can fine-tune JVM options using this environment variable parameter.

❷ This particular Elasticsearch pod definition doesn't request much memory or CPU; a production setup would likely need more.

❸ These volume definitions are key. A `volumeClaimTemplates` key within the `StatefulSet` definition helps users define where the backing storage for Elasticsearch to use will be mounted.

The `volumeMounts` define where in the container the `PersistentVolume` will be mounted. In Kubernetes, when resources have a need for external storage, such as a block storage device mounted at a certain location like `/usr/share/elasticsearch/data`, they can request a "claim" to storage using a `PersistentVolumeClaim` (which is what having `volumeClaimTemplates` in the `StatefulSet` creates). Either the `PersistentVolumeClaim` will be matched up with an existing storage device that Kubernetes knows it can use (known as a `PersistentVolume`), or a new `PersistentVolume` will be created (so-called dynamic provisioning of a `PersistentVolume`). For instance, creating a `PersistentVolumeClaim` to Ceph block storage in YAML might look like this:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: claim1
spec:
  storageClassName: cephfs
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
```

Because of this use of `PersistentVolume`, we can claim storage that lives beyond the lifetime of any individual pod (which will eventually be rolled) and maintains state over time.

Logstash could be deployed as a `Deployment` with an associated `Service` allowing processes across the cluster to connect to it and send data, and Kibana could also be its own `Deployment`, accessed using port forwarding from `kubectl` or normal Kubernetes ingress.

GPU Computing: Inference at the Edge

Many intelligent applications need to use special hardware such as GPUs to effectively train and classify. Convolutional neural networks, for instance, have a structure that is so "deep" that it's only practical to optimize their loss function on GPUs in a highly parallel fashion. Likewise, GPUs might be needed for efficient predictions once the models are trained. To complicate things further, we might also need to deploy this specialized hardware at the "edge"—not in the cloud, but physically near the use case they are intended for.

For instance, self-driving cars relying on neural networks will need embedded GPUs and models because they need to act and respond rapidly. Such applications cannot handle the network latency of connecting to a predictive service running in the cloud or a remote data center. Deploying such a use case has a lot of challenging facets. For one thing, the correct drivers and hardware support have to be installed on the target edge nodes, and they are likely to be difficult to update once deployed. Getting telemetry about how things are performing and troubleshooting issues is going to be difficult since the computing is happening on-site at the edge and is not necessarily available on the public Internet.

Kubernetes supports a variety of ways to help with these issues. In the next section, we'll take a look at architecture for a quality control application that classifies manufactured products as defective or nondefective.

Implementation

As you can see in Figure 5-9, our implementation will have edge nodes that have special hardware attached (GPUs). These will power a TensorFlow model that classifies photos of products as they come off the assembly line. If a product is not above a certain acceptable threshold of confidence according to the model's notion of quality, a human operator can be alerted to inspect or remove the product. If we can power these predictions with this type of edge computing, we can greatly increase the efficiency of quality control in our manufacturing.

Our edge nodes are connected to an upstream Kubernetes cluster that can run remotely and is responsible for scheduling on these special types of workloads as requested. Connection to an upstream

Kubernetes cluster on a public or private network can be secured using TLS, ensuring that only the correct downstreams can connect. We can't structure the deployment haphazardly due to the need to ensure we are running within the proper edge computing location and with the proper hardware, so there are specific ways of deploying such an architecture on Kubernetes that we need to consider.

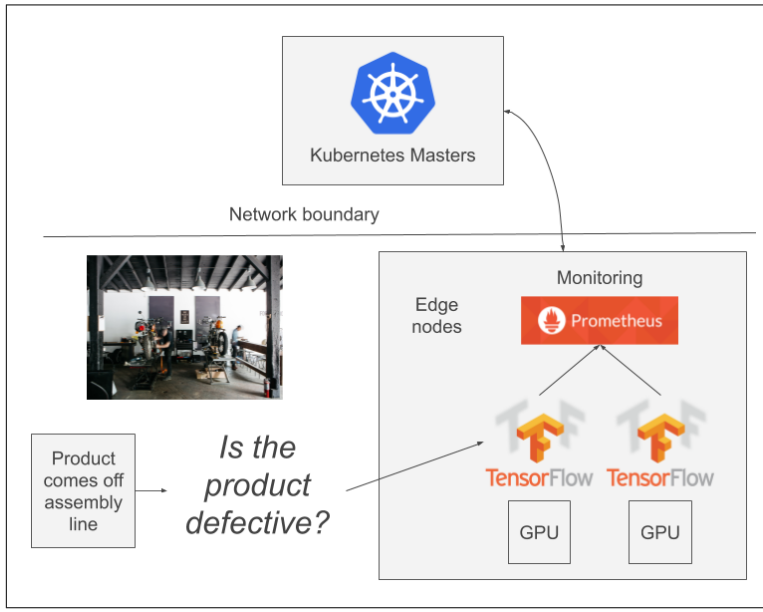


Figure 5-9. Kubernetes has the flexibility to take advantage of special hardware and deploy selectively to environments

Kubernetes Ideas

Nvidia, a popular manufacturer of GPUs, has a GPU Operator that is perfect for addressing many concerns out of the box that we would otherwise have to worry about ourselves. Operators in Kubernetes allow users to pass off programs to other users that mimic some of the actions an expert human operator might take for specific use cases when managing a cluster. For instance, operators exist that help users administrate things like databases, which need a lot of attention in the form of backups, tuning, and monitoring.

The Nvidia GPU Operator source code can help us understand what's going on behind the scenes. For starters, it installs Node Feature Discovery (NFD), a Kubernetes plug-in that will detect specific

attributes of interest that nodes have available and advertise them for scheduling using node labels. Hence, we can tell Kubernetes to schedule selectively based upon which features are available and have some freedom from managing that part manually as cluster administrators, which could be a big headache. The list of features supported by NFD has huge utility because it is so comprehensive, offering an opportunity for users to schedule based on kernel modules available, special CPU features, special devices, and more. In this case, PCIe device IDs are the feature and label of interest.

With NFD installed, the Nvidia operator can help us schedule pods based on where GPU resources are available. The Nvidia operator also takes care of things system administrators would typically have to do, such as installing the correct device drivers as a kernel module on each host. The operator state machine follows the same model of the Kubernetes control loop, ensuring that all parts of the cluster are converged to the desired state—that is, with each relevant node having the correct version of the kernel drivers installed for the attached Nvidia device.

The Nvidia operator will help us manage the GPU components, but what if we have several edge locations under our wing in the overarching Kubernetes cluster? For that, we can ensure correct scheduling across various locations using node labels. Adding a label to a given Kubernetes node is simple:

```
$ kubectl label node edgenode-name edgeLocation=factory1
node/edgenode-name labeled
```

This can be referenced later as a constraint on scheduling our containers to ensure they are deployed to the correct edge location.

For instance, the configuration for running a single pod that is scheduled using the Nvidia operator would look like the following. Since we set the `edgeLocation` label to `factory1` above, we match that in this pod definition's `nodeSelector` key. This ensures the resources are scheduled according to what we need. We could even extend the concept to ensure we get the specific GPU type we want:

```
apiVersion: v1
kind: Pod
metadata:
  name: qa-classifier
  labels:
    app: qa-classifier
spec:
  securityContext:
    fsGroup: 0
  containers:
  - name: qa-classifier
    image: quay.io/ourcompany/qa-classifier:v0.2.3
    resources:
      limits:
        nvidia.com/gpu: 1
    ports:
    - containerPort: 5000
      name: qa-classifier
  nodeSelector:
    edgeLocation: factory1
    offeredGPU: nvidia-tesla-p100
```

The Nvidia GPU Operator also includes instructions for scraping GPU metrics with Prometheus. This allows us to monitor what's going on with our GPUs, giving us a peek at what would otherwise be obtuse performance. The classification APIs could also emit Prometheus statistics defining their latency, classification performance, and the number of errors encountered. Their logs could meanwhile be forwarded using a `DaemonSet` running on every node to a remote or on-site logging cluster, such as our previous ELK example we've mentioned several times throughout the text.

The options for deploying an edge computing setup using GPUs on Kubernetes, and platforms like OpenShift, look crisp and enticing. Kubernetes is likely to power the next wave of intelligent applications running at the edge as well as in the data center proper.

About the Authors

Kyle Bader is a principal solutions architect on the storage solutions team at Red Hat. Kyle lends his design and operational skills with Ceph to help companies be successful with massive-scale data systems.

Sherard Griffin is responsible for the Open Data Hub, a community-driven reference architecture for building an AI-as-a-service platform on OpenShift. Sherard also leads the deployment of Open Data Hub in Red Hat's internal data center where data scientists across the company run machine learning initiatives. Additionally, he works with hardware and software partners to build out an ecosystem of AI technologies optimized to run on OpenShift as certified operators.

With over 20 years of experience in the IT industry, over a decade in the storage industry, and deep experience in containers and hybrid cloud, **Pete Brey** provides great insight into today's data analytics challenges and how to best solve those challenges using proven solutions that work.

Daniel Riek is a senior director for the AI Center of Excellence at Red Hat and is a technologist with 20 years of experience in the open source business, global product management, engineering management, and product strategy. He brings solid experience in a global leadership role at a S&P 500 company to sales and solution architect positions, IT consulting, management of software development teams across VC financed and publicly traded companies.

Nathan LeClaire is a Go programmer and author living in San Francisco, CA. He has explored his passion for developer tools and open source working at startups such as Docker and Honeycomb.