Kubernetes

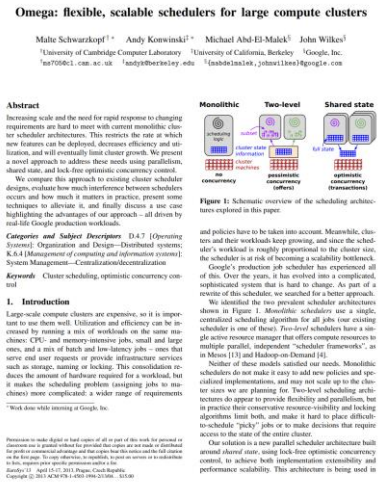
Christopher Meiklejohn

October 28, 2020

Why Kubernetes?

Now that we have a bunch of microservices, how do we deploy and manage them?

Kubernetes (k8s) is an open-source ***container orchestration framework*** that originated at Google.



Omega: flexible, scalable schedulers for large compute clusters

Malte Schwarzkopf[†] * Andy Konwinski[‡] * Michael Abd-El-Malek[§] John Wilkes[§]

[†]University of Cambridge Computer Laboratory [‡]University of California, Berkeley [§]Google, Inc.

[†]ms705@cl.cam.ac.uk [‡]andyk@berkeley.edu [§]{mabdelmalek,johnwilkes}@google.com

Abstract

Increasing scale and the need for rapid response to changing requirements are hard to meet with current monolithic cluster scheduler architectures. This restricts the rate at which new features can be deployed, decreases efficiency and utilization, and will eventually limit cluster growth. We present a novel approach to address these needs using parallelism, shared state, and lock-free optimistic concurrency control.

We compare this approach to existing cluster scheduler designs, evaluate how much interference between schedulers occurs and how much it matters in practice, present some techniques to alleviate it, and finally discuss a use case highlighting the advantages of our approach – all driven by real-life Google production workloads.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design—Distributed systems; K.6.4 [*Management of computing and information systems*]: System Management—Centralization/decentralization

Keywords Cluster scheduling, optimistic concurrency control

1. Introduction

Large-scale compute clusters are expensive, so it is important to use them well. Utilization and efficiency can be increased by running a mix of workloads on the same machines: CPU- and memory-intensive jobs, small and large ones, and a mix of batch and low-latency jobs – ones that serve end user requests or provide infrastructure services such as storage, naming or locking. This consolidation reduces the amount of hardware required for a workload, but it makes the scheduling problem (assigning jobs to machines) more complicated: a wider range of requirements

* Work done while interning at Google, Inc.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EuroSys'13 April 15-17, 2013, Prague, Czech Republic

Copyright (C) 2013 ACM 978-1-4503-1994-2/13/04 ... $15.00

Figure 1: Schematic overview of the scheduling architectures explored in this paper.

and policies have to be taken into account. Meanwhile, clusters and their workloads keep growing, and since the scheduler's workload is roughly proportional to the cluster size, the scheduler is at risk of becoming a scalability bottleneck.

Google's production job scheduler has experienced all of this. Over the years, it has evolved into a complicated, sophisticated system that is hard to change. As part of a rewrite of this scheduler, we searched for a better approach.

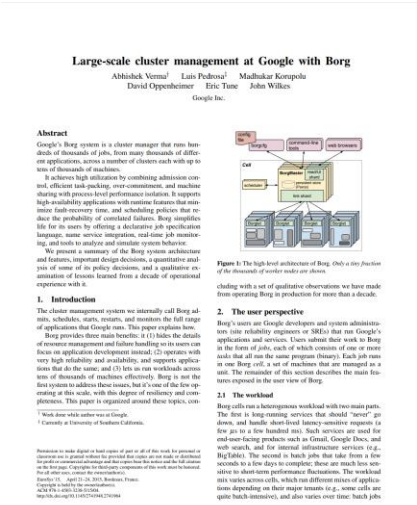
We identified the two prevalent scheduler architectures shown in Figure 1. *Monolithic schedulers* use a single, centralized scheduling algorithm for all jobs (our existing scheduler is one of these). *Two-level* schedulers have a single active resource manager that offers compute resources to multiple parallel, independent "scheduler frameworks", as in Mesos [13] and Hadoop-on-Demand [4].

Neither of these models satisfied our needs. Monolithic schedulers do not make it easy to add new policies and specialized implementations, and may not scale up to the cluster sizes we are planning for. Two-level scheduling architectures do appear to provide flexibility and parallelism, but in practice their conservative resource-visibility and locking algorithms limit both, and make it hard to place difficult-to-schedule "picky" jobs or to make decisions that require access to the state of the entire cluster.

Our solution is a new parallel scheduler architecture built around *shared state*, using lock-free optimistic concurrency control, to achieve both implementation extensibility and performance scalability. This architecture is being used in

351

EuroSys '13



Large-scale cluster management at Google with Borg

Abhishek Verma[†] Luis Pedrosa[‡] Madhukar Korupolu

David Oppenheimer Eric Tune John Wilkes

Google Inc.

Abstract

Google's Borg system is a cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines.

It achieves high utilization by combining admission control, efficient task-packing, over-commitment, and machine sharing with process-level performance isolation. It supports high-availability applications with runtime features that minimize fault-recovery time, and scheduling policies that reduce the probability of correlated failures. Borg simplifies life for its users by offering a declarative job specification language, name service integration, real-time job monitoring, and tools to analyze and simulate system behavior.

We present a summary of the Borg system architecture and features, important design decisions, a quantitative analysis of some of its policy decisions, and a qualitative examination of lessons learned from a decade of operational experience with it.

1. Introduction

The cluster management system we internally call Borg admits, schedules, starts, restarts, and monitors the full range of applications that Google runs. This paper explains how.

Borg provides three main benefits: it (1) hides the details of resource management and failure handling so its users can focus on application development instead; (2) operates with very high reliability and availability, and supports applications that do the same; and (3) lets us run workloads across tens of thousands of machines effectively. Borg is not the first system to address these issues, but it's one of the few operating at this scale, with this degree of resiliency and completeness. This paper is organized around these topics, con-

[†] Work done while author was at Google.

[‡] Currently at University of Southern California.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

EuroSys'15, April 21–24, 2015, Bordeaux, France.

Copyright is held by the owner/author(s).

ACM 978-1-4503-3238-5/15/04.

http://dx.doi.org/10.1145/2741948.2741964

Figure 1: The high-level architecture of Borg. *Only a tiny fraction of the thousands of worker nodes are shown.*

cluding with a set of qualitative observations we have made from operating Borg in production for more than a decade.

2. The user perspective

Borg's users are Google developers and system administrators (site reliability engineers or SREs) that run Google's applications and services. Users submit their work to Borg in the form of *jobs*, each of which consists of one or more *tasks* that all run the same program (binary). Each job runs in one Borg *cell*, a set of machines that are managed as a unit. The remainder of this section describes the main features exposed in the user view of Borg.

2.1 The workload

Borg cells run a heterogeneous workload with two main parts. The first is long-running services that should "never" go down, and handle short-lived latency-sensitive requests (a few µs to a few hundred ms). Such services are used for end-user-facing products such as Gmail, Google Docs, and web search, and for internal infrastructure services (e.g., BigTable). The second is batch jobs that take from a few seconds to a few days to complete; these are much less sensitive to short-term performance fluctuations. The workload mix varies across cells, which run different mixes of applications depending on their major tenants (e.g., some cells are quite batch-intensive), and also varies over time: batch jobs

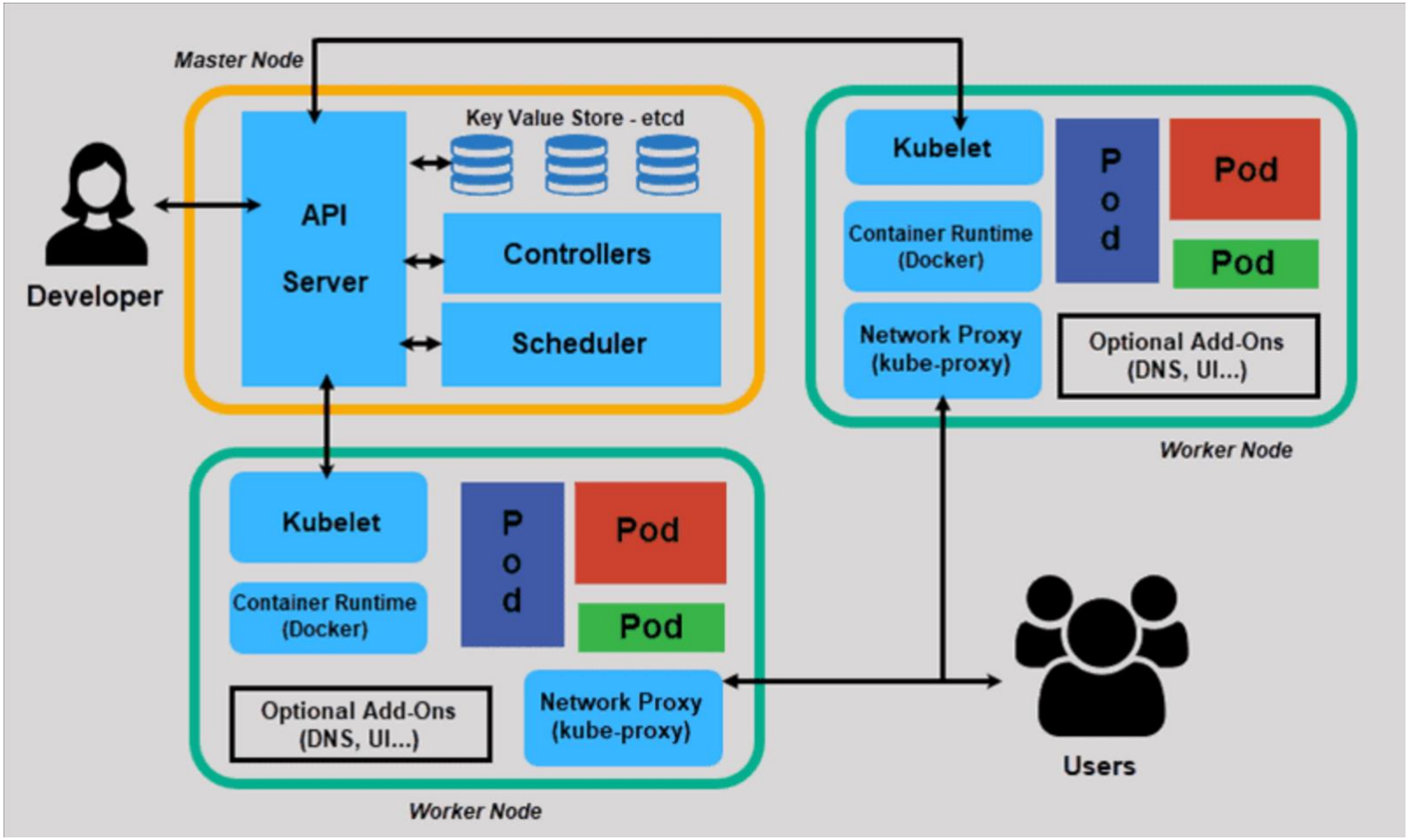
EuroSys '15

Google Kubernetes Engine
(Google Cloud Platform)

Azure Kubernetes Service
(Microsoft Azure)

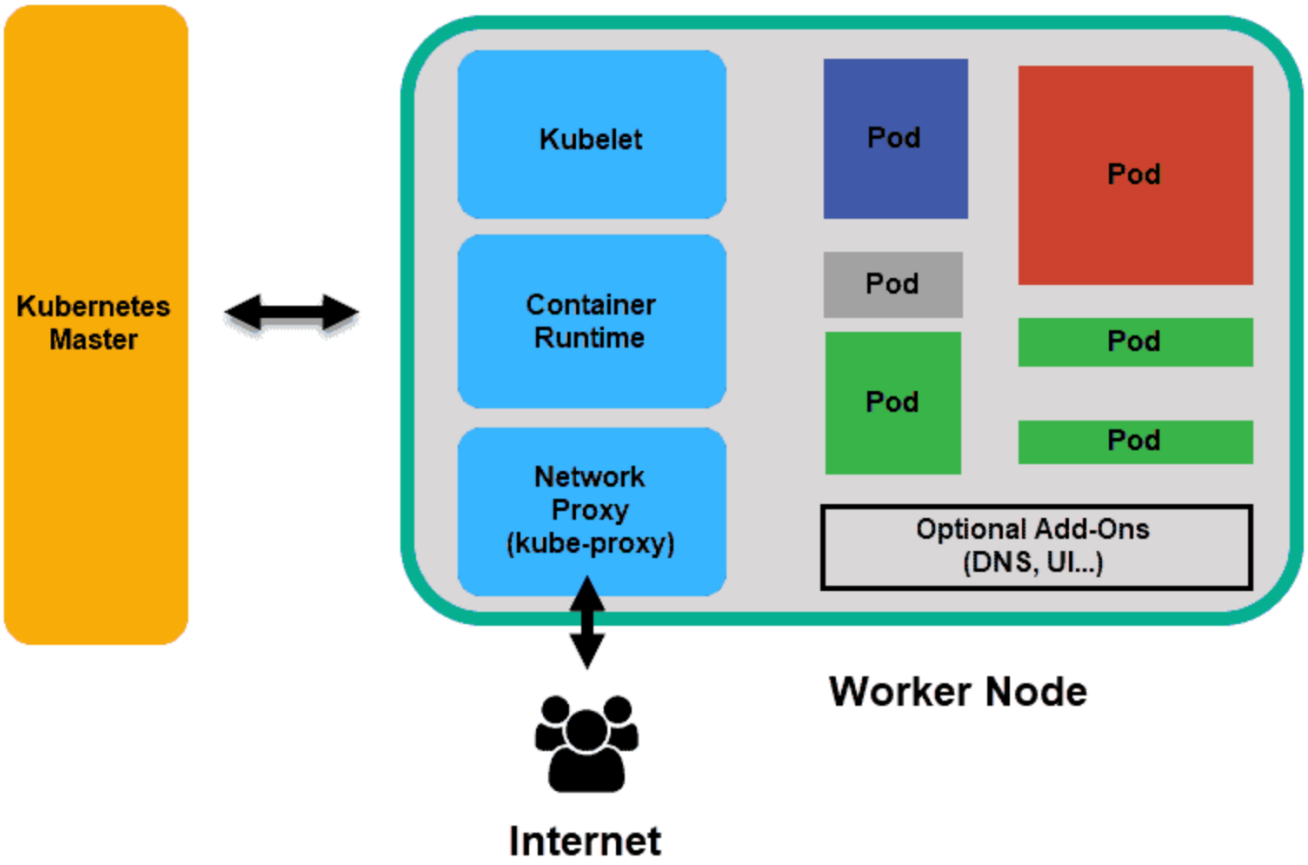
Elastic Kubernetes Service
(Amazon Web Services)

Architecture



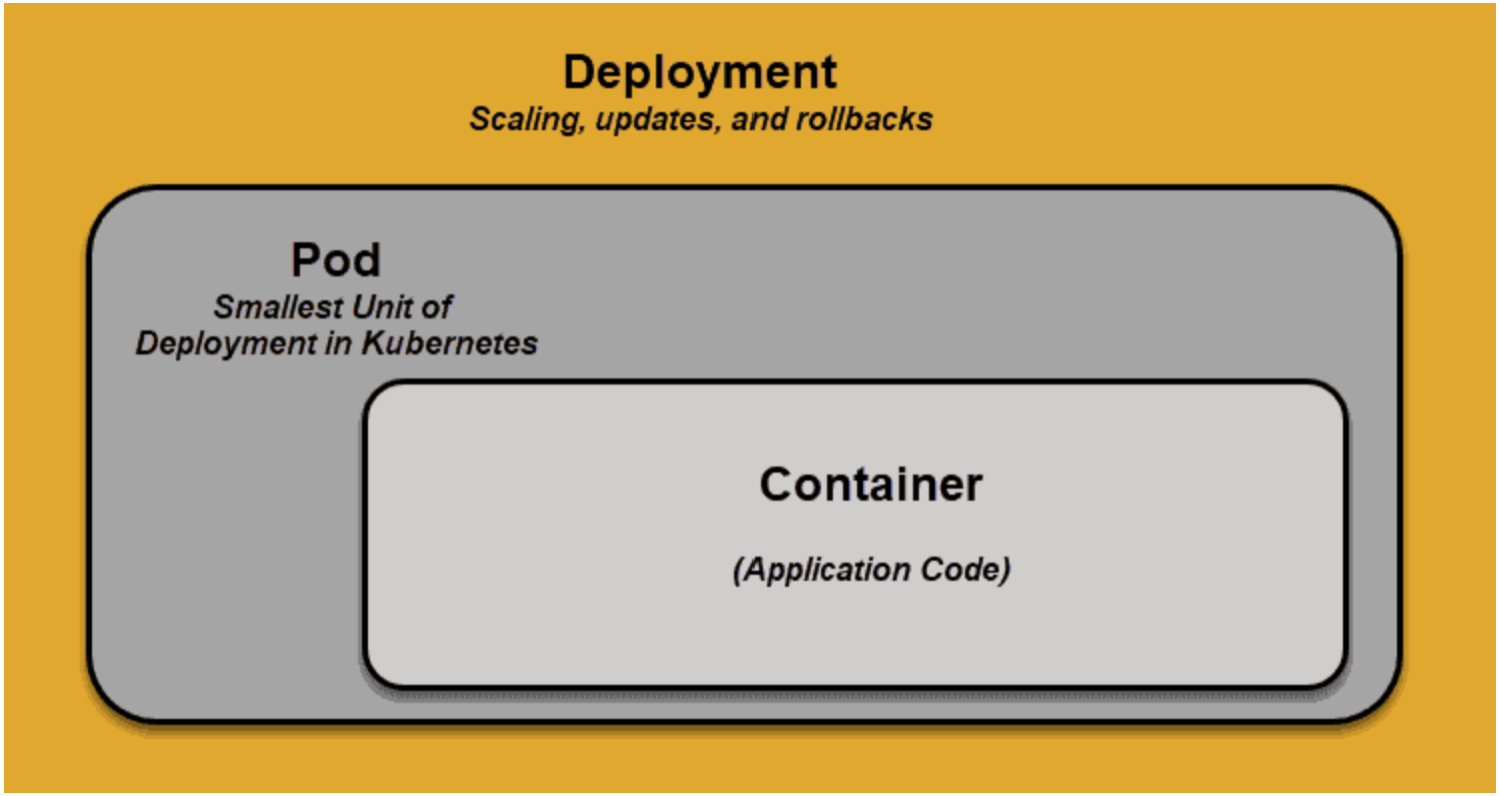
reference, https://phoenixnap.com/kb/understanding-kubernetes-architecture-diagrams

Worker Nodes



reference, https://phoenixnap.com/kb/understanding-kubernetes-architecture-diagrams

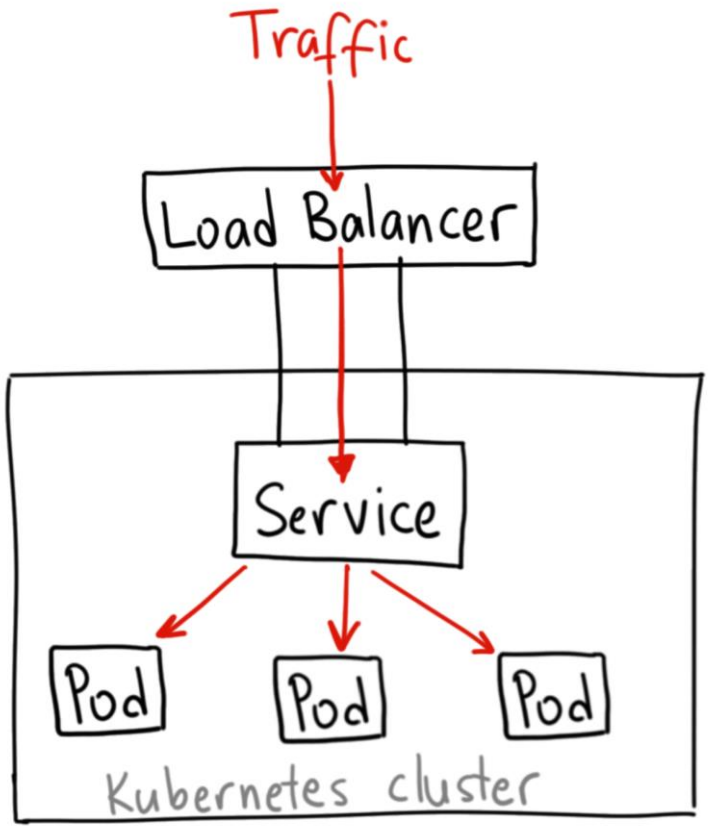
Deployments, Pods, Containers



reference, https://phoenixnap.com/kb/understanding-kubernetes-architecture-diagrams

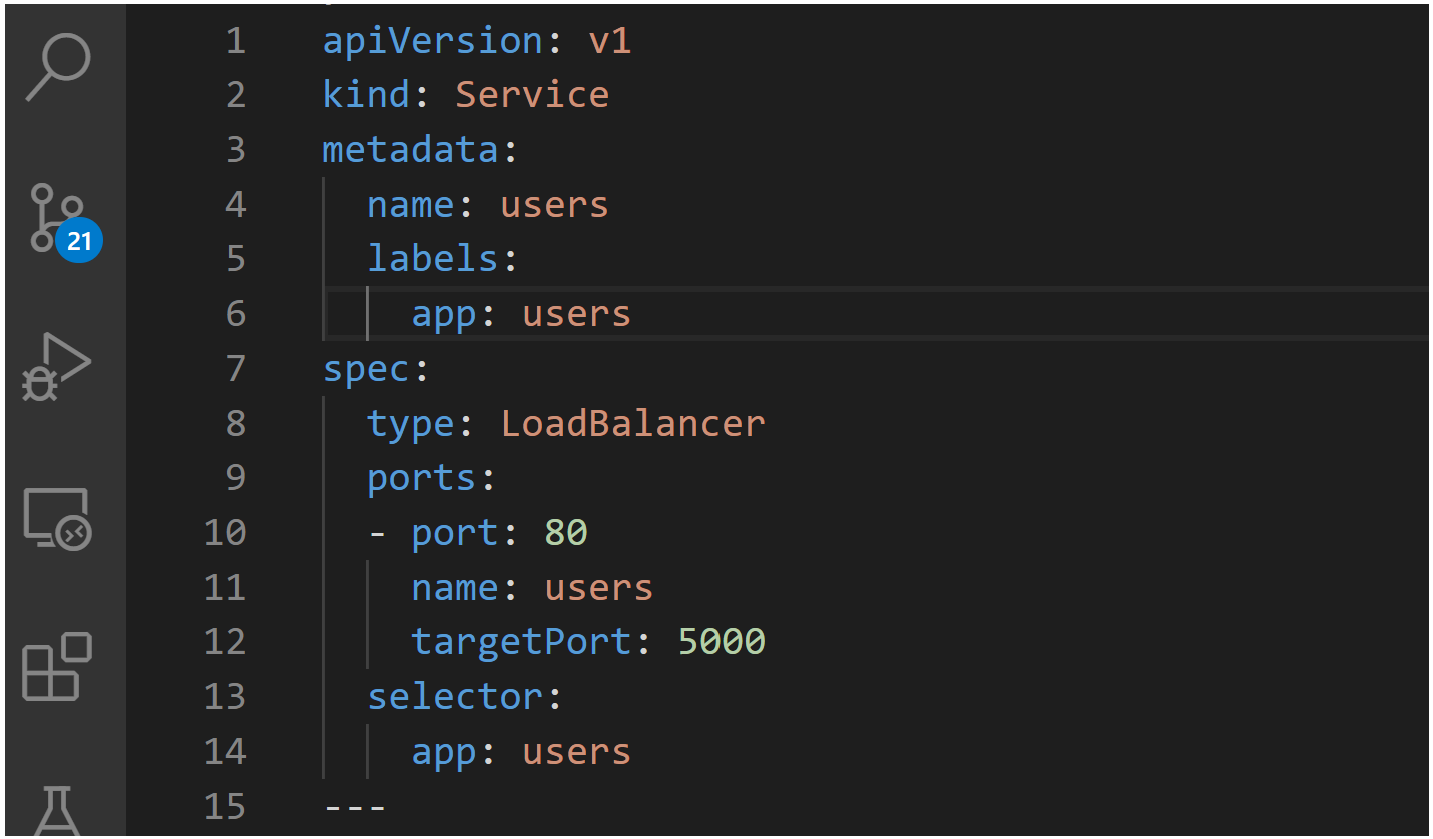
Deployments

Services: LoadBalancer Example



reference, https://medium.com/avmconsulting-blog/external-ip-service-type-for-kubernetes-ec2073ef5442

Service




```yaml
apiVersion: v1
kind: Service
metadata:
  name: users
  labels:
    app: users
spec:
  type: LoadBalancer
  ports:
  - port: 80
    name: users
    targetPort: 5000
  selector:
    app: users
---
```

Useful Commands

```
# Display active pods.
$ kubectl get pods

# Display active deployments
$ kubectl get deployments

# Display active services.
$ kubectl get services

# Display details of a service
$ kubectl describe service SERVICE_NAME

# Create the services defined in File.yaml
$ kubectl create -f File.yaml

# Get help!
$ kubectl help
```