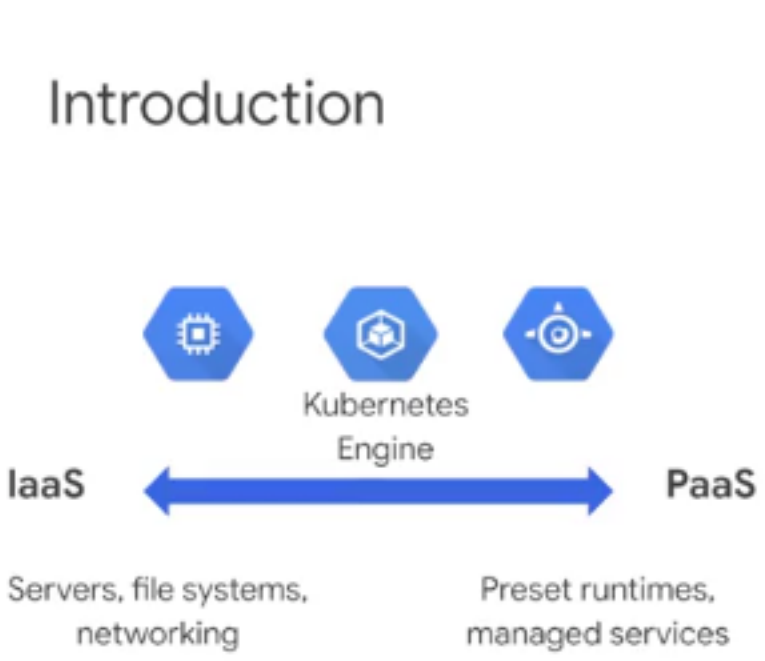
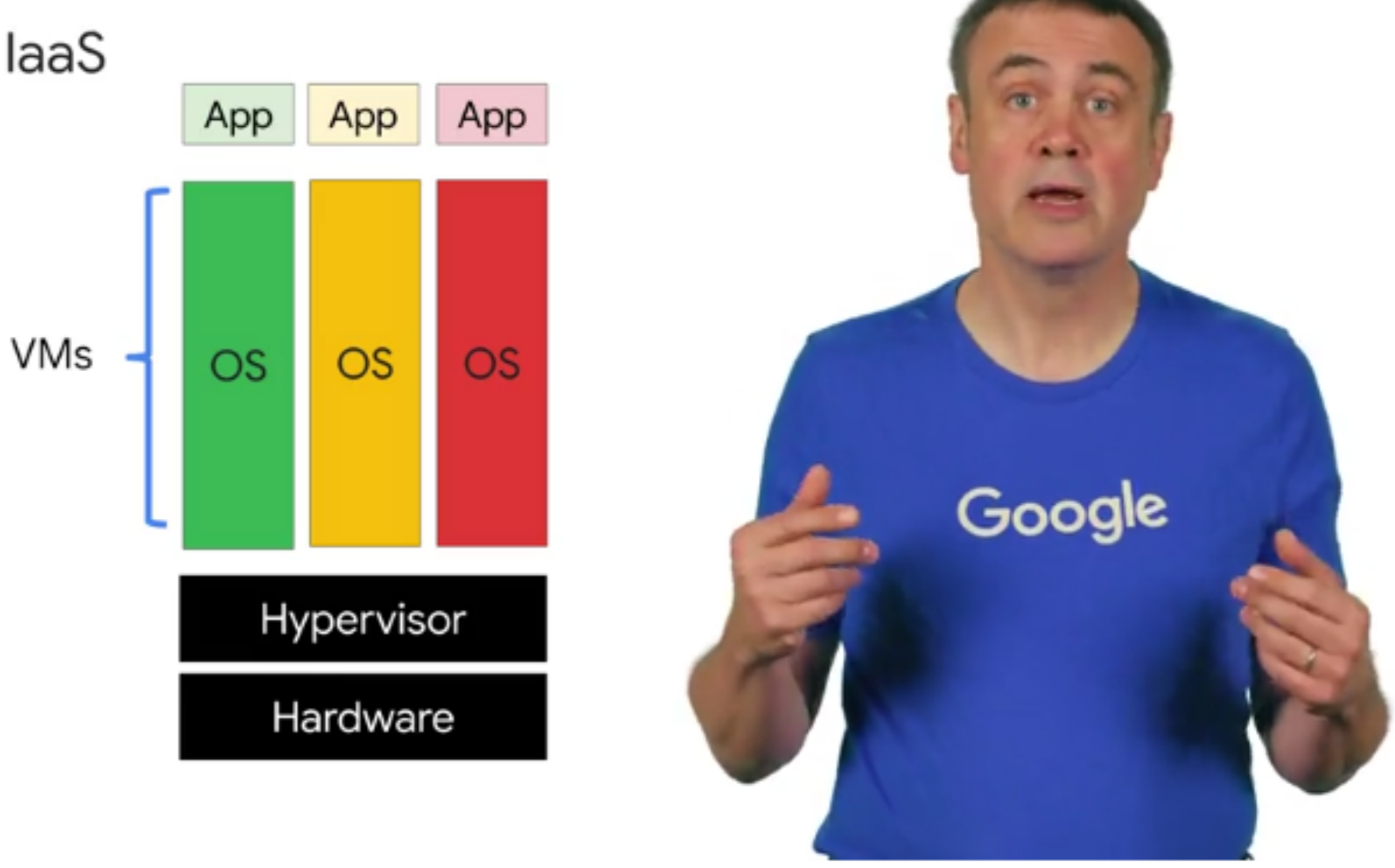


Now I'm going to introduce you to a service called Kubernetes Engine. It's like an Infrastructure as a Service offering in that it saves you infrastructure chores. It's also like a platform as a service offering, in that it was built with the needs of developers in mind.

First, I'll tell you about a way to package software called Containers. I'll describe why Containers are useful, and how to manage them in Kubernetes Engine. Let's begin by remembering that infrastructure as a service offering let you share compute resources with others by virtualizing the hardware.

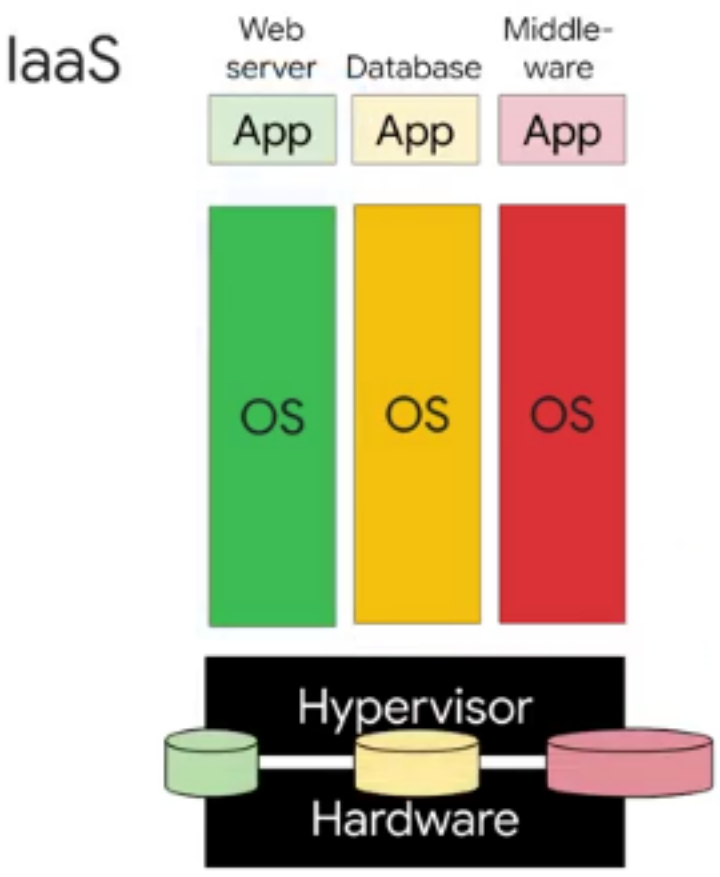
Each Virtual Machine has its own



Each Virtual Machine has its own instance of an operating system, your choice, and you can build and run applications on it with access to memory, file systems, networking interfaces, and the other attributes that physical computers also have. But flexibility comes with a cost.

In an environment like this, the smallest unit of compute is a Virtual Machine together with its application. The guest OS, that is the operating system maybe large, even gigabytes in size. It can take minutes to boot up. Often it's worth it.

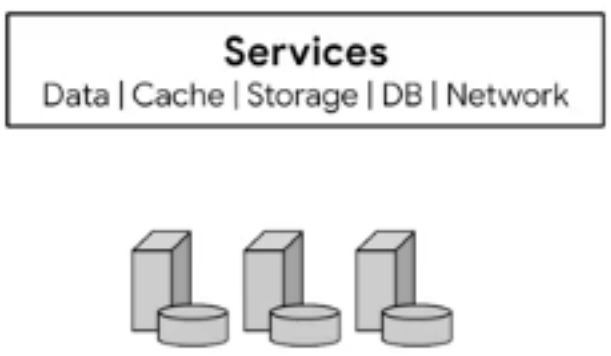
Virtual Machine are highly configurable, and you can install



Virtual Machine are highly configurable, and you can install and run your tools of choice. So you can configure the underlying system resources such as disks and networking, and you can install your own web server database or a middle ware. But suppose your application is a big success.

As demand for it increases, you have to scale out in units of an entire Virtual Machine with a guest operating system for each. That can mean your resource consumption grows faster than you like. Now, let's make a contrast with a Platform as a Service environment like App Engine.

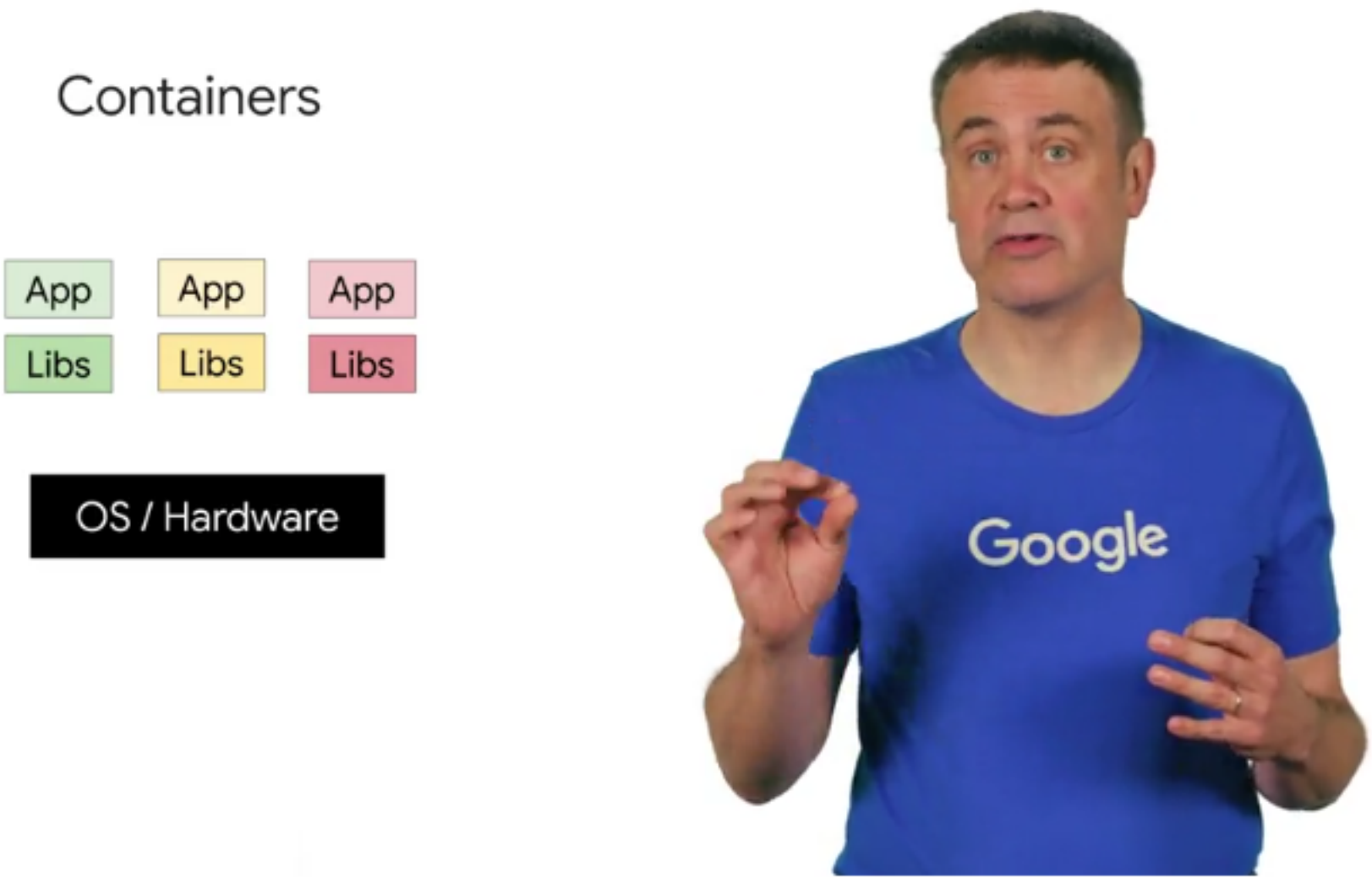
App Engine



From the perspective of someone deploying on App Engine, it feels very different. Instead of getting a blank Virtual Machine, you get access to a family of services that applications need.

So all you do is write your code and self-contained workloads that use these services and include any dependent libraries. As demand for your application increases, the platform scales your applications seamlessly and independently by workload and infrastructure.

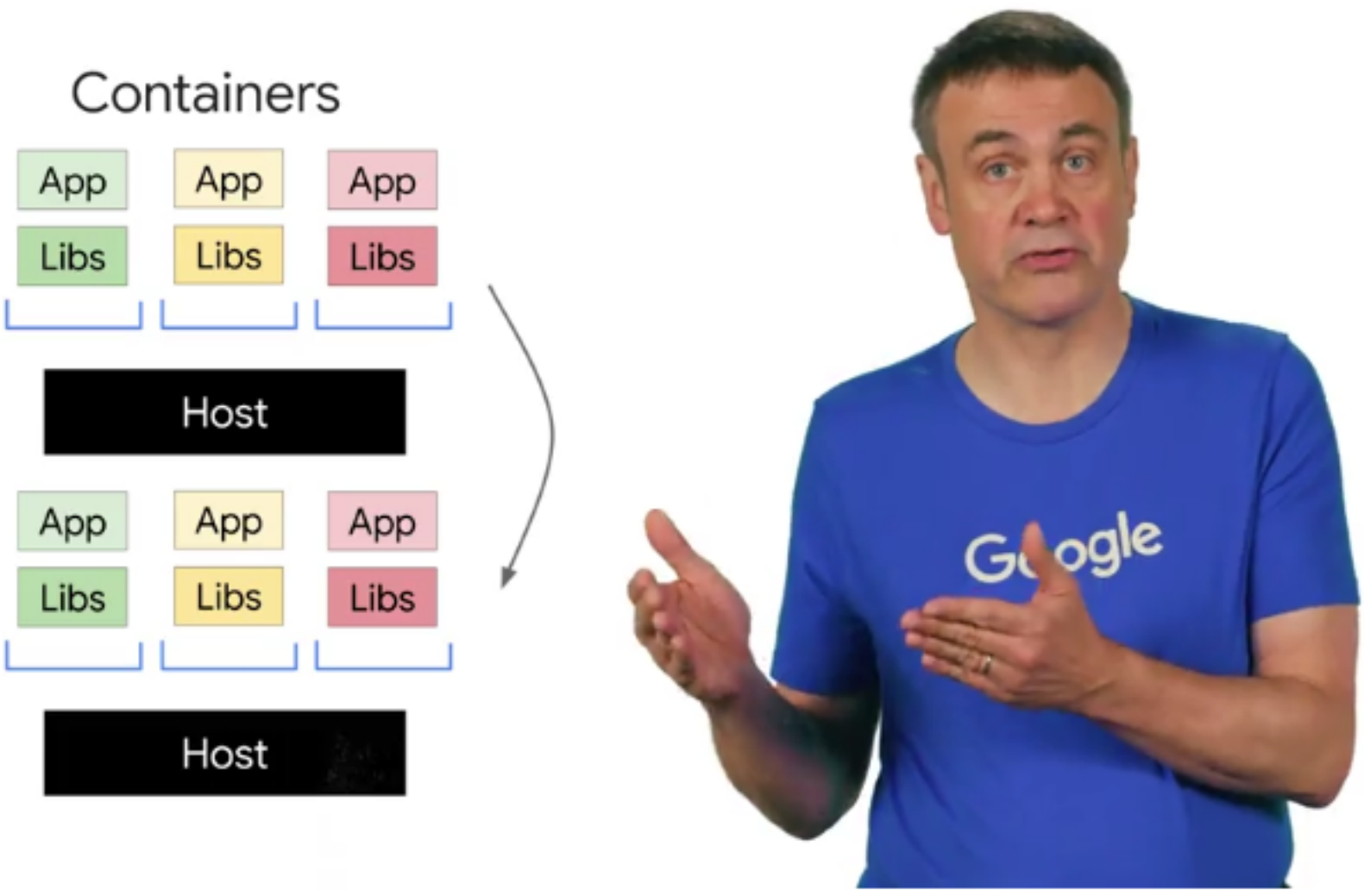
This scales rapidly, but you give up control of the underlying server



The idea of a Container is to give you the independent scalability of workloads like you get in a PaaS environment, and an abstraction layer of the operating system and hardware, like you get in an Infrastructure as a Service environment.

What do you get as an invisible box around your code and its dependencies with limited access to its own partition of the file system and hardware? Remember that in Windows, Linux, and other operating systems, a process is an instance of a running program.

A Container starts as quickly as a new



You can treat the operating system and hardware as a black box. So you can move your code from development, to staging, to production, or from your laptop to the Cloud without changing or rebuilding anything.

If you went to scale for example a web server, you can do so in seconds, and deploy dozens or hundreds of them depending on the size of your workload on a single host. Well, that's a simple example. Let's consider a more complicated case.

You'll likely want to build your applications using lots of Containers, each performing their own function,

across a group of hosts. The host can scale up and down, and start and stop Containers as demand for your application changes, or even as hosts fail and are replaced.

A tool that helps you do this well is Kubernetes. Kubernetes makes it easy to orchestrate many Containers on many hosts. Scale them, roll out new versions of them, and even roll back to the old version if things go wrong. First, I'll show you how you build and run containers.

The most common format for Container images is the one defined by the open source tool Docker. In my example, I'll use Docker to bundle an

The most common format for Container images is the one defined by the open source tool Docker. In my example, I'll use Docker to bundle an application and its dependencies into a Container. You could use a different tool.

For example, Google Cloud offers Cloud Build, a managed service for building Containers. It's up to you. Here is an example of some code you may have written. It's a Python web application, and it uses the very popular Flask framework.

Whenever a web browser talks to it by asking for its top-most document, it

app.py

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
  return "Hello World!\n"

@app.route("/version")
def version():
  return "Helloworld 1.0\n"

if __name__ == "__main__":
  app.run(host='0.0.0.0')
```

Whenever a web browser talks to it by asking for its top-most document, it replies "hello world". Or if the browser instead appends/version to the request, the application replies with its version. Great. So how do you deploy this application?

It needs a specific version of Python and a specific version of Flask, which we control using Python's requirements.txt file, together with its other dependencies too. So you use a Docker file to specify how your code gets packaged into a Container.

For example, Ubuntu is a popular distribution of Linux. Let's start there.

Dockerfile

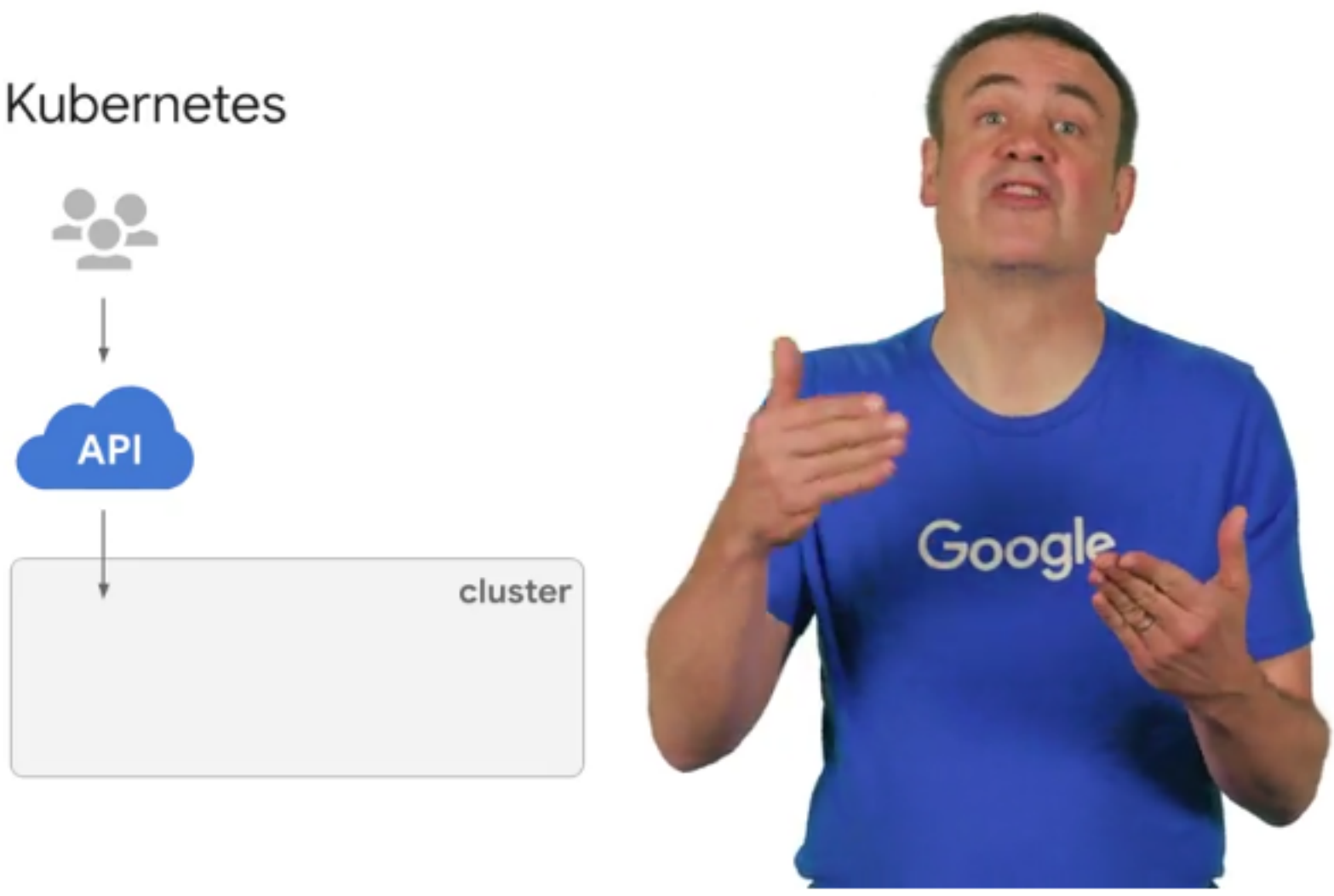
```
FROM ubuntu:18.10
RUN apt-get update -y && \
    apt-get install -y python3-pip python3-dev
COPY requirements.txt /app/requirements.txt
WORKDIR /app
RUN pip3 install -r requirements.txt
COPY . /app
ENTRYPOINT ["python3", "app.py"]
```

You can install Python the same

Build and run

```
$> docker build -t py-server .
$> docker run -d py-server
```

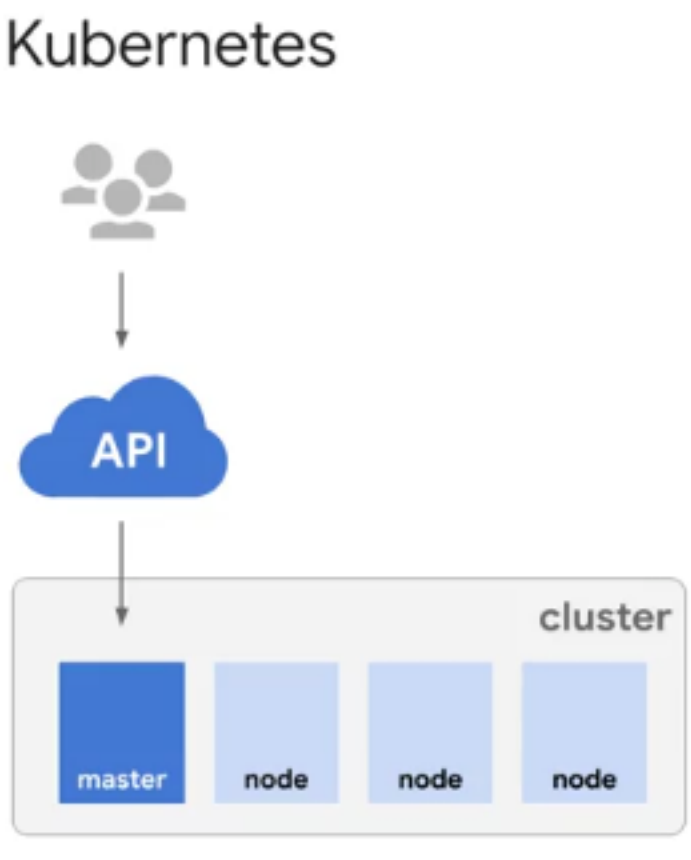
you'd probably upload the image



Kubernetes comes in. Kubernetes is an open-source orchestrator for containers so you can better manage and scale your applications.

Kubernetes offers an API that lets people, that is authorized people, not just anybody, control its operation through several utilities. Very soon we'll meet one of those utilities, the kubectl command. Kubernetes lets you deploy containers on a set of nodes called a cluster. What's a cluster?

It's a set of master components that control the system as a whole and



It's a set of master components that control the system as a whole and a set of nodes that run containers. In Kubernetes, a node represents a computing instance. In Google Cloud, nodes are virtual machines running in Compute Engine.

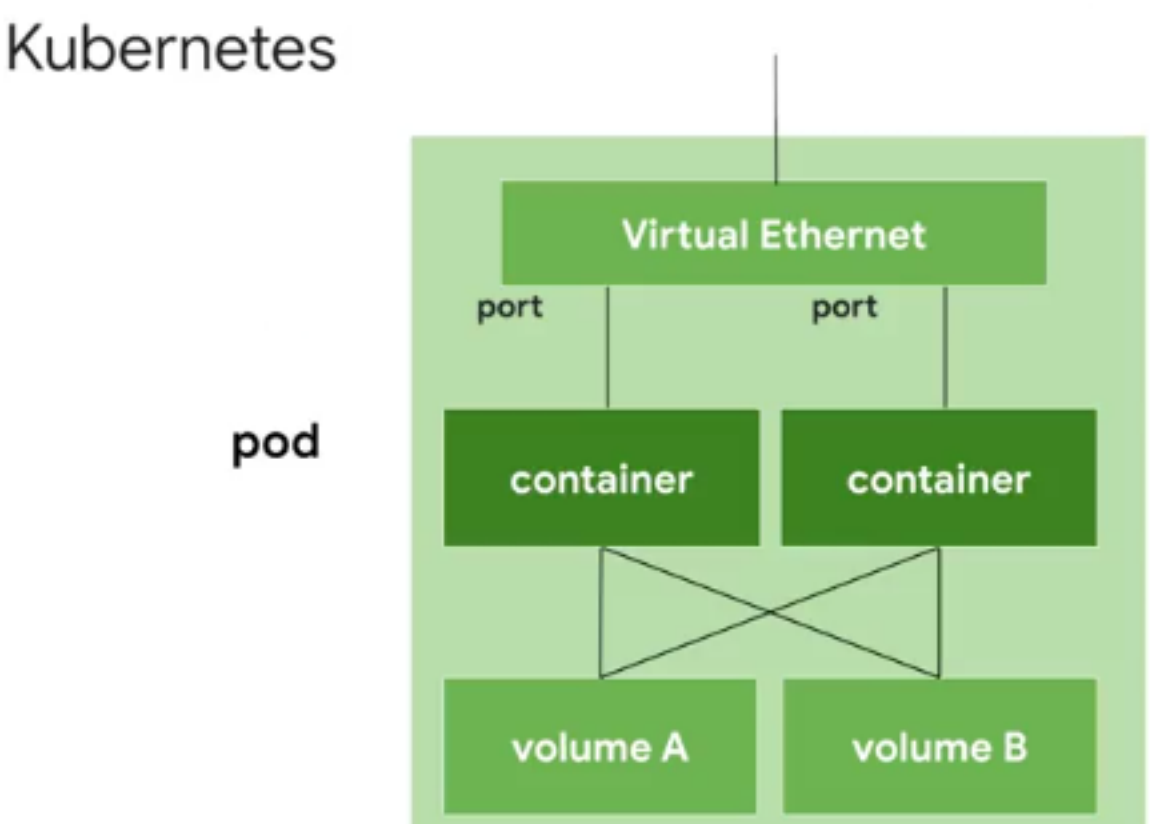
To use Kubernetes, you can describe a set of applications and how they should interact with each other, and Kubernetes figures out how to make that happen. Kubernetes makes it easy to run containerized applications like the one we built in the last lesson, but how do you get a Kubernetes cluster?

You can always build one yourself

You can always build one yourself on your own hardware, or in any environment that provides virtual machines, but that's work. And if you've built it yourself, you have to maintain it. That's even more toil.

Because that effort is not always a valuable use of your time, Google Cloud provides Kubernetes Engine, which is Kubernetes as a managed service in the cloud. You can create a Kubernetes cluster with Kubernetes Engine using the GCP console or the g-cloud command that's provided by the Cloud SDK.

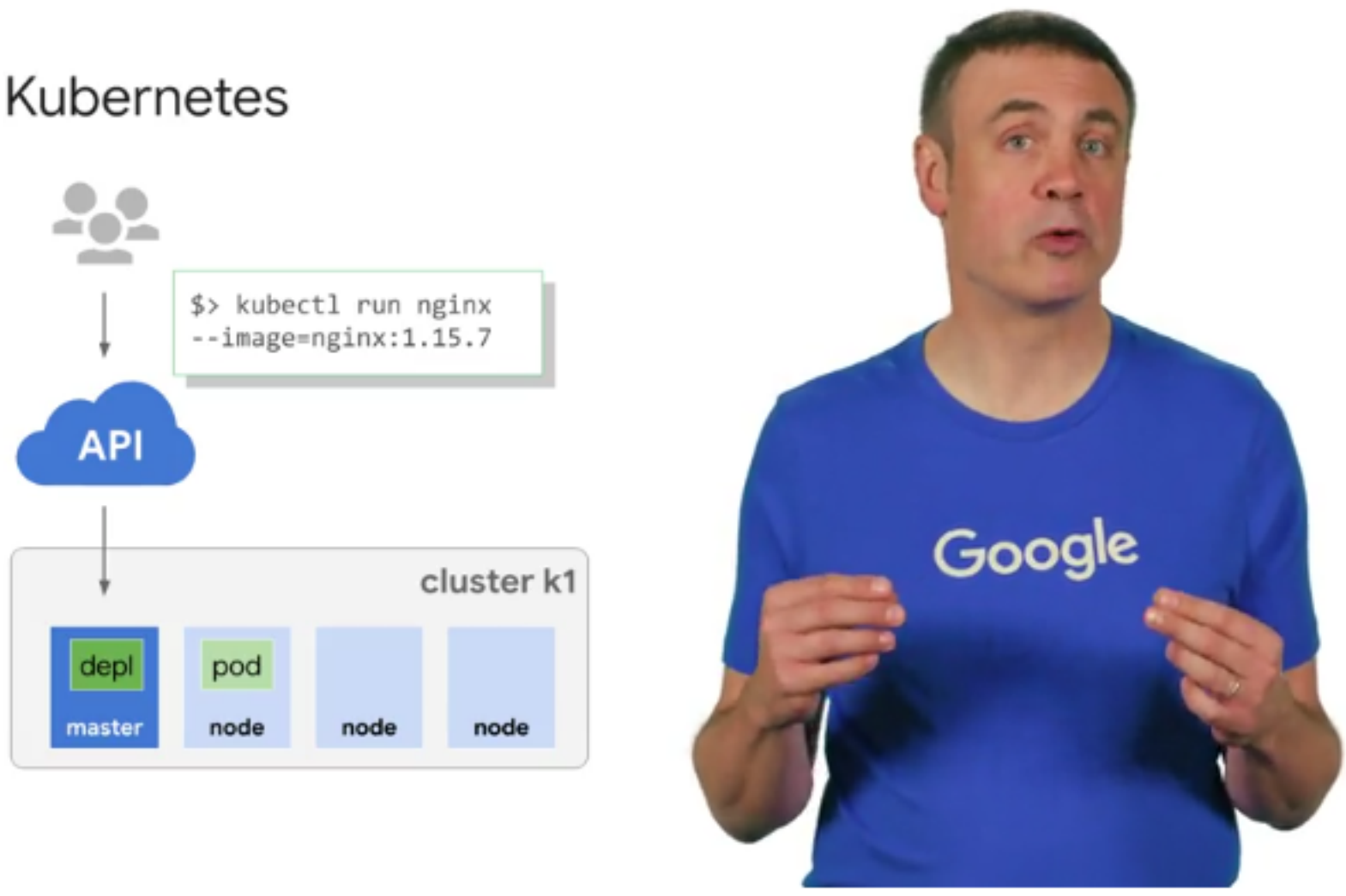
GKE clusters can be customized,



GCP console. Whenever Kubernetes deploys a container or a set of related containers, it does so inside an abstraction called a pod.

A pod is the smallest deployable unit in Kubernetes. Think of a pod as if it were a running process on your cluster. It could be one component of your application or even an entire application. It's common to have only one container per pod.

But if you have multiple containers with a hard dependency, you can package them into a single pod. They'll automatically share networking and they can have disk storage



They'll automatically share networking and they can have disk storage volumes in common. Each pod in Kubernetes gets a unique IP address and set of ports for your containers.

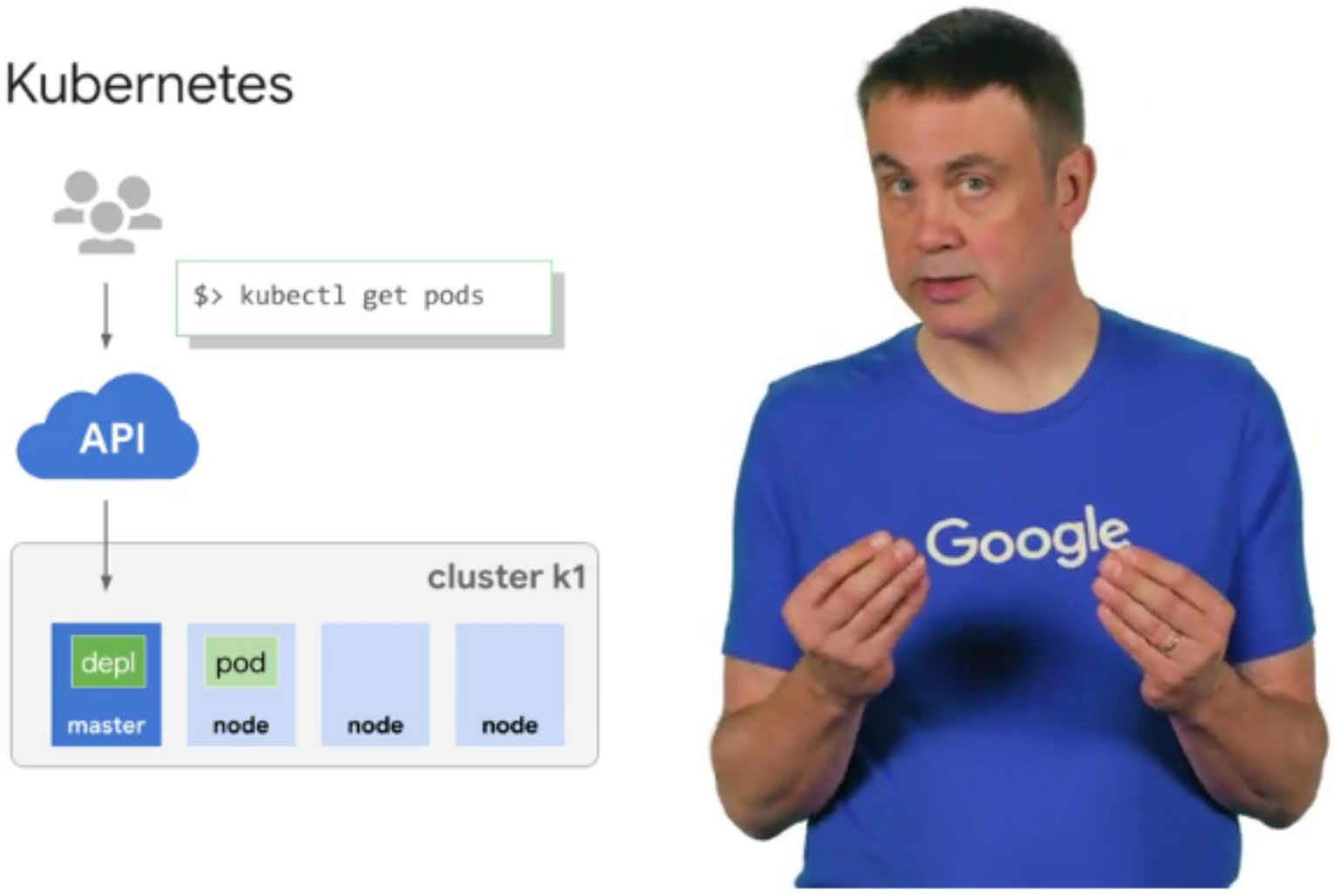
Because containers inside a pod can communicate with each other using the localhost network interface, they don't know or care which nodes they're deployed on. One way to run a container in a pod in Kubernetes is to use the kubectl run command.

We'll learn a better way later in this module, but this gets you started quickly. Running the kubectl run command starts a deployment with

quickly. Running the kubectl run command starts a deployment with a container running a pod. In this example, the container running inside the pod is an image of the popular nginx open source web server.

The kubectl command is smart enough to fetch an image of nginx of the version we request from a container registry. So what is a deployment? A deployment represents a group of replicas of the same pod. It keeps your pods running even if a node on which some of them run on fails.

You can use a deployment to contain a component of your application or



container registry. So what is a deployment? A deployment represents a group of replicas of the same pod. It keeps your pods running even if a node on which some of them run on fails.

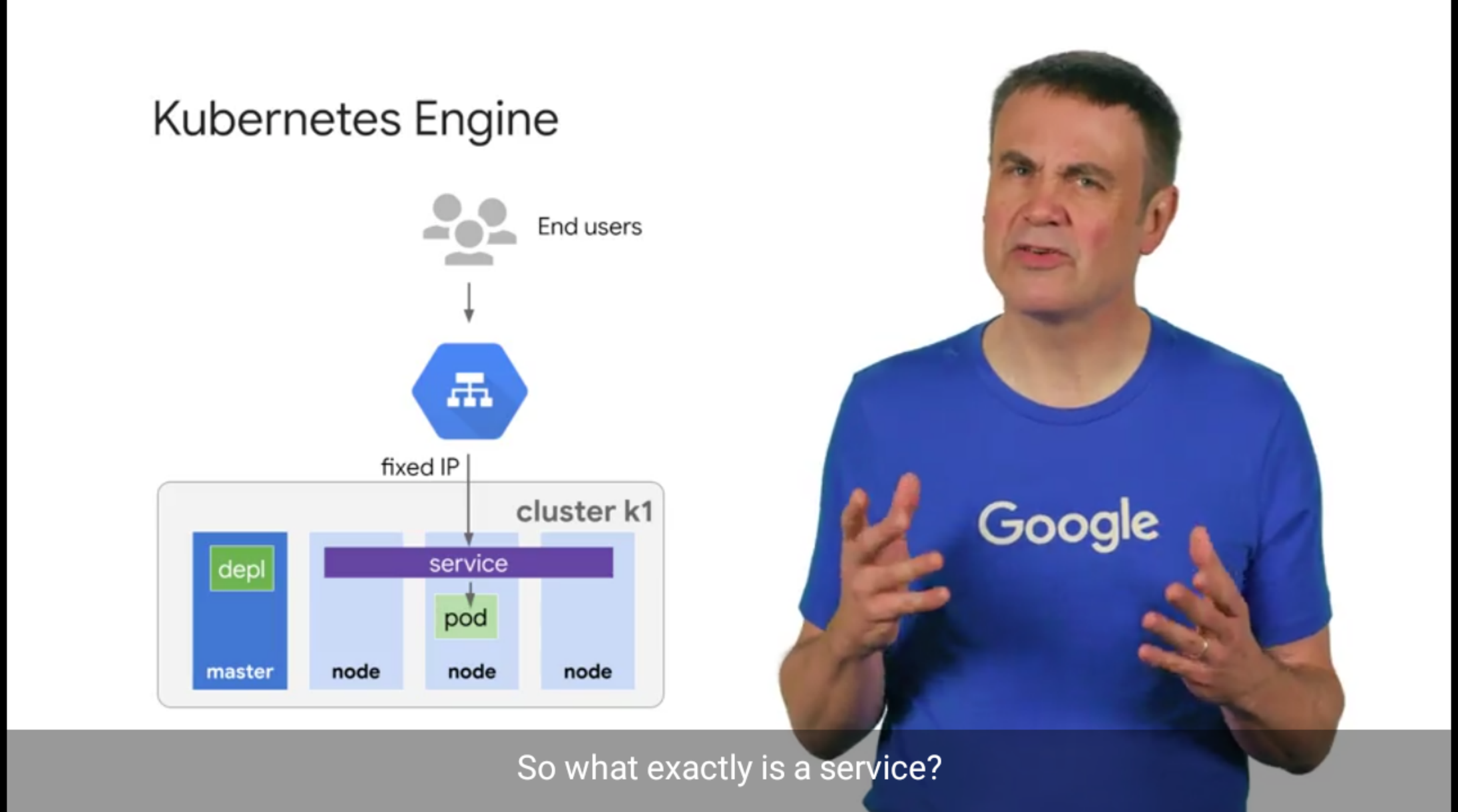
You can use a deployment to contain a component of your application or even the entire application. In this case, it's the nginx web server. To see the running nginx pods, run the command kubectl get pods.

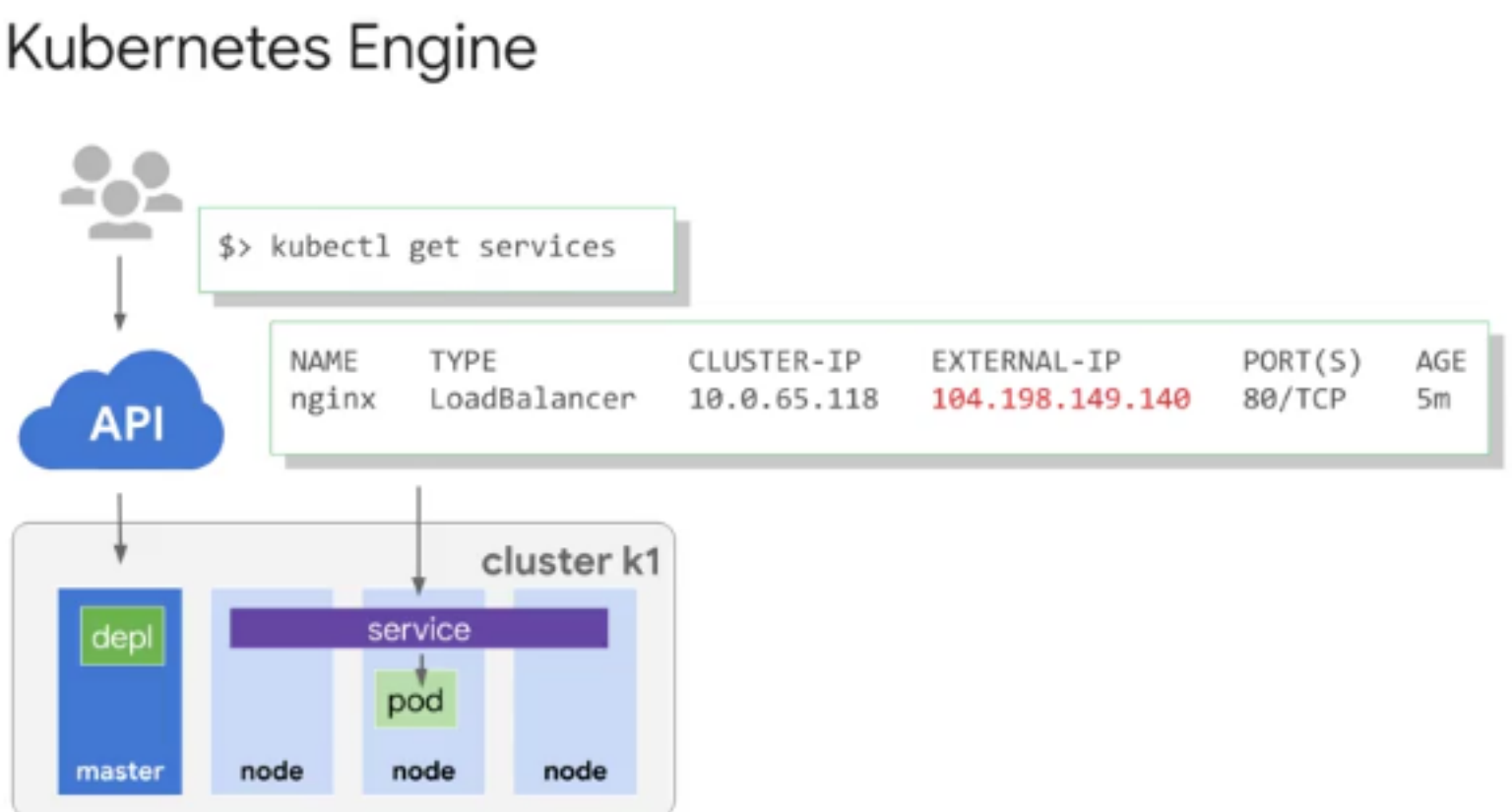
By default, pods in a deployment or only accessible inside your cluster, but what if you want people on the Internet to be able to access the content in your nginx web server?

To make the pods in your deployment publicly available, you can connect a load balancer to it by running the kubectl expose command. Kubernetes then creates a service with a fixed IP address for your pods. A service is the fundamental way Kubernetes represents load balancing.

To be specific, you requested Kubernetes to attach an external load balancer with a public IP address to your service so that others outside the cluster can access it. In GKE, this kind of load balancer is created as a network load balancer.

This is one of the managed load bal-

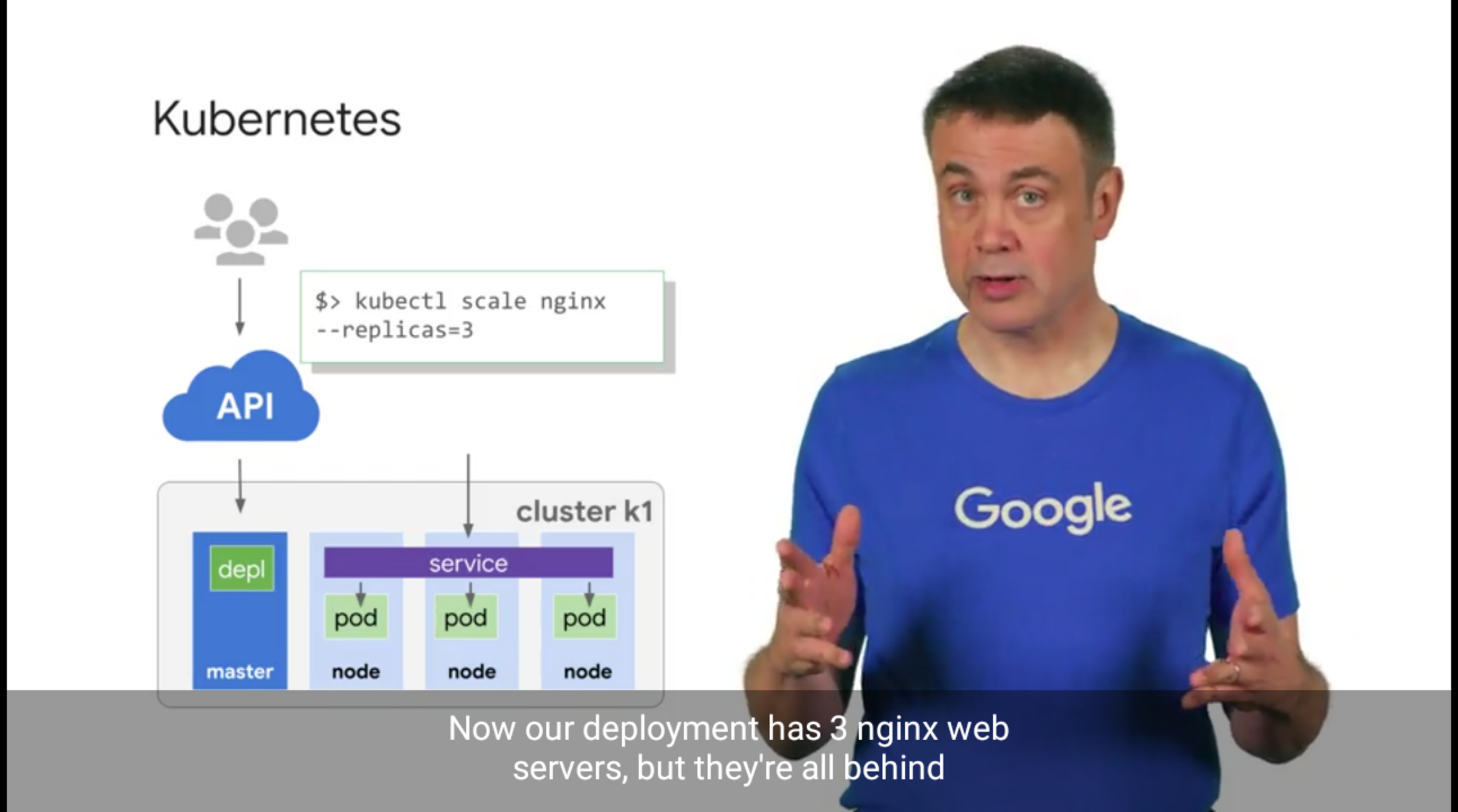

Kubernetes Engine
End users
fixed IP
cluster k1
depl
service
pod
master
node
node
node
Google
So what exactly is a service?



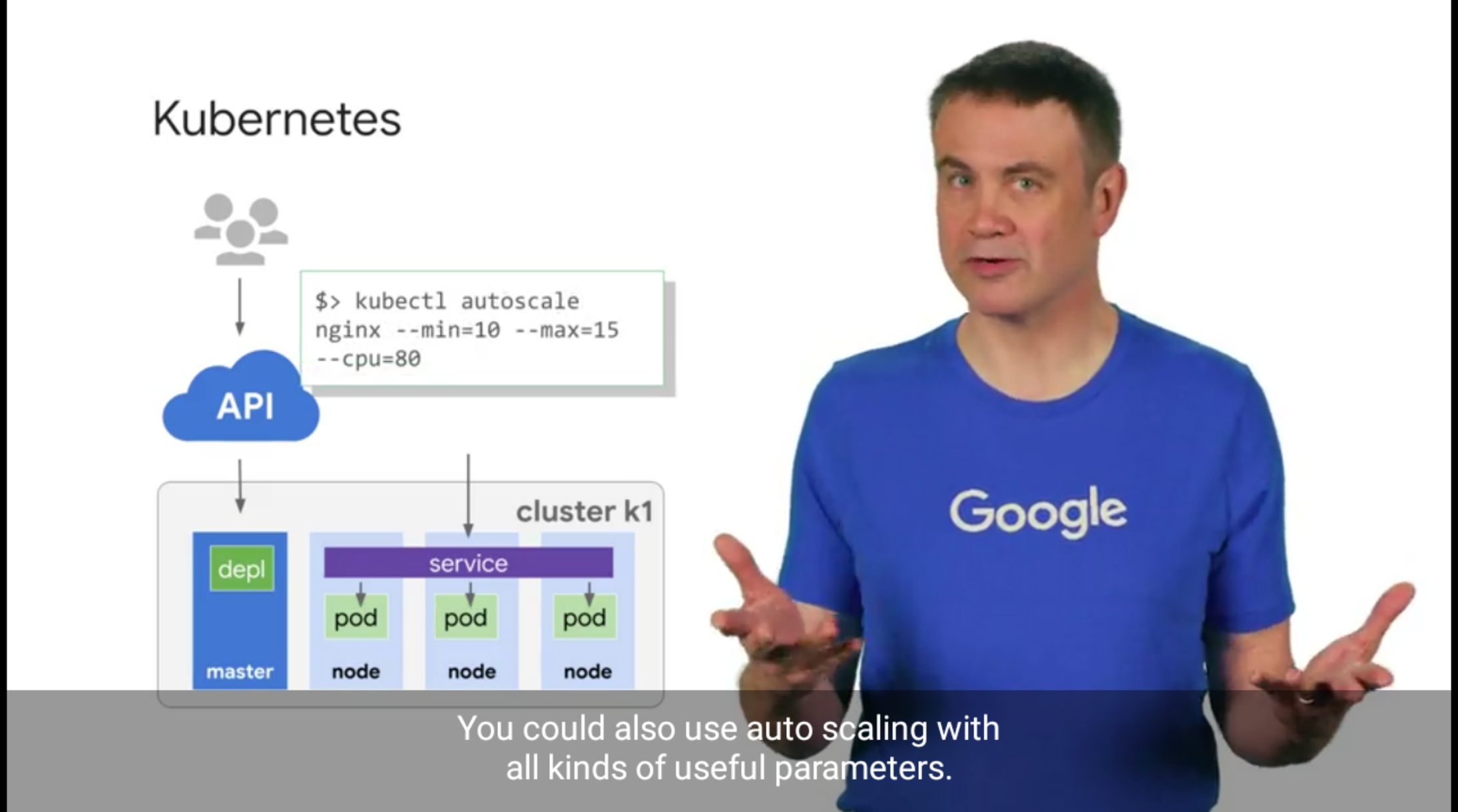
The kubectl get services command shows you your service's public IP address. Clients can use this address to hit the nginx container remotely. What if you need more power? To scale a deployment, run the kubectl scale command.

Now our deployment has 3 nginx web servers, but they're all behind the service and they're all available through one fixed IP address. You could also use auto scaling with all kinds of useful parameters. For example, here's how to auto scale a deployment based on CPU usage.

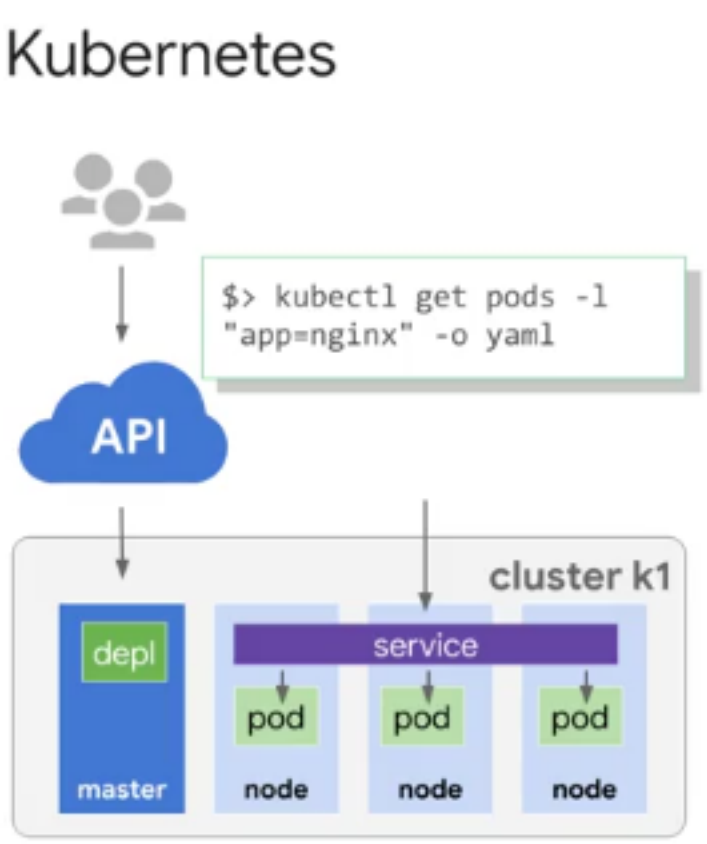
In the command shown, you specify



Now our deployment has 3 nginx web
servers, but they're all behind


Kubernetes
$> kubectl autoscale
nginx --min=10 --max=15
--cpu=80
API
cluster k1
depl
master
service
pod
node
pod
node
pod
node
Google
You could also use auto scaling with
all kinds of useful parameters.

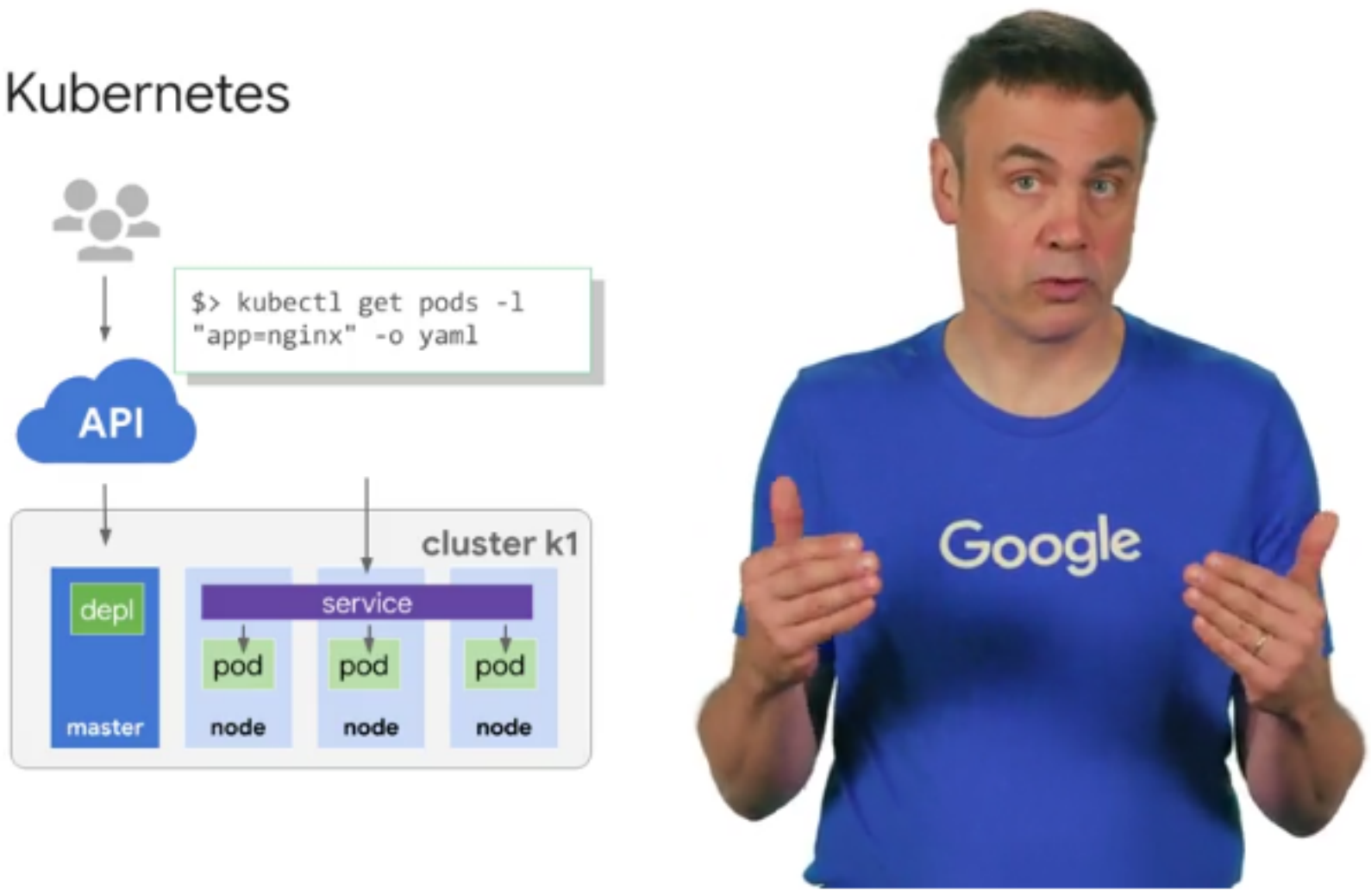

Kubernetes
$> kubectl get pods -l "app=nginx" -o yaml
API
cluster k1
depl
master
service
pod
node
pod
node
pod
node
Google
the real strength of Kubernetes comes
when you work in a declarative way.



Instead of issuing commands, you provide a configuration file that tells Kubernetes what you want your desired state to look like and Kubernetes figures out how to do it. These configuration files then become your management tools.

To make a change, edit the file and then present the changed version to Kubernetes. The command on the slide is one way we could get a starting point for one of these files based on the work we've already done. That command's output would look something like this.

These files are intimidating the first

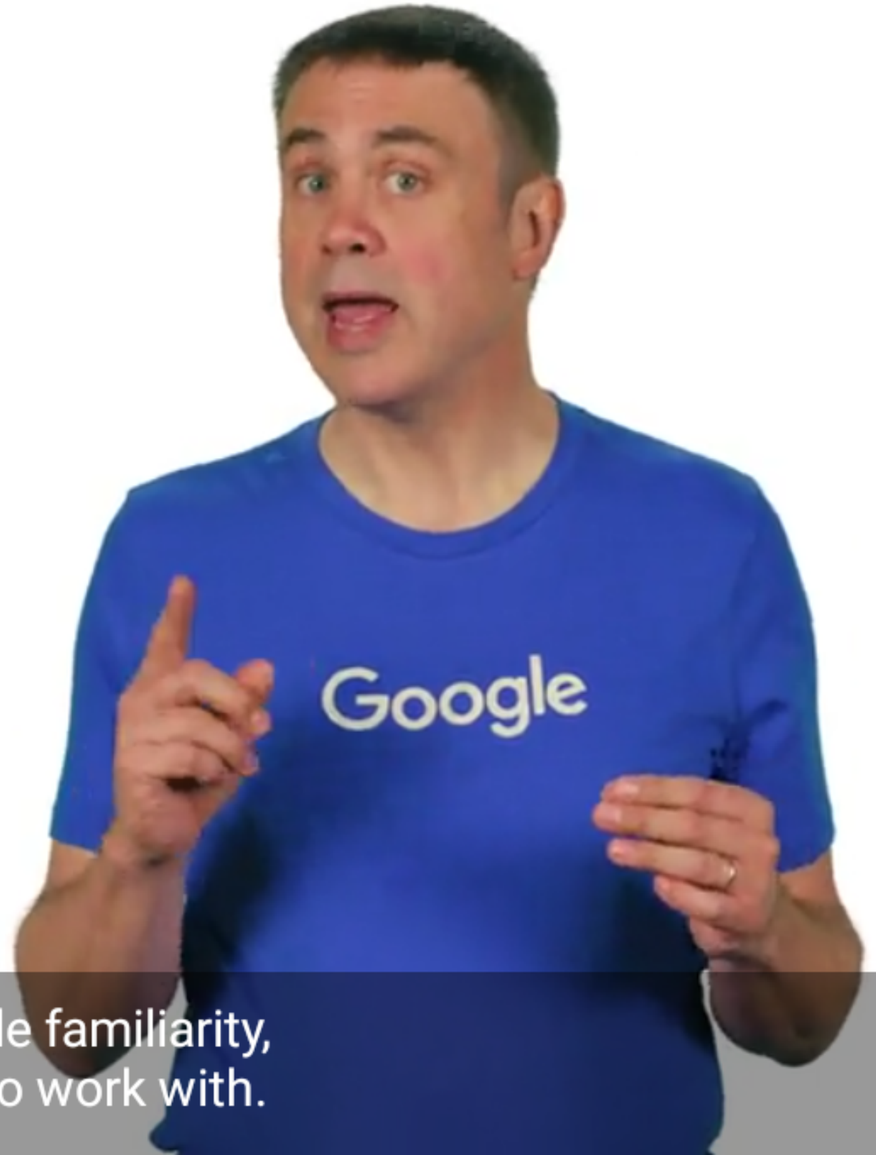


These files are intimidating the first time you see them because they're long and they contain syntax you don't yet understand. But with a little familiarity, they're easy to work with. And you can save them in a version control system to keep track of the changes you made to your infrastructure.

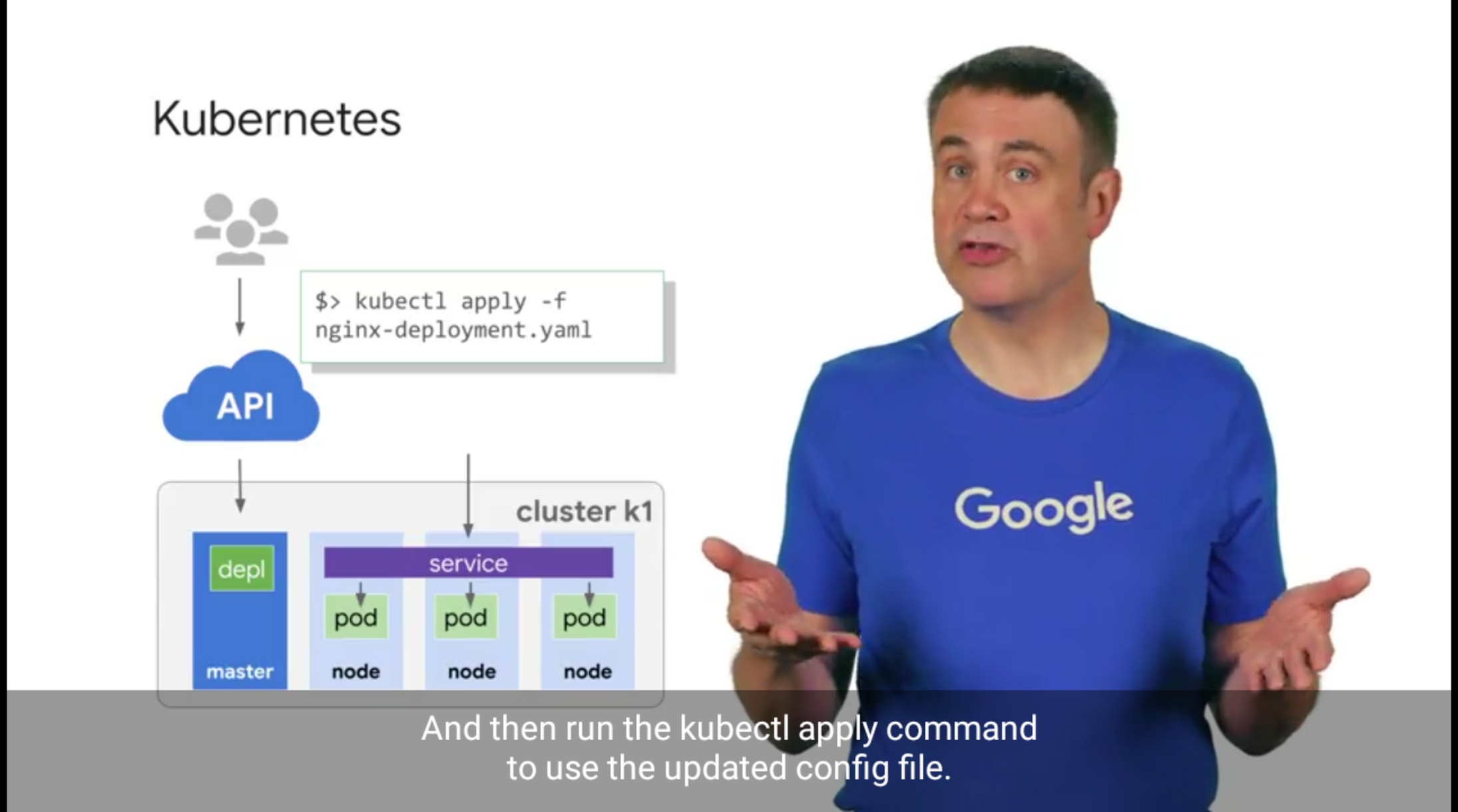
In this case, the deployment configuration file declares that you want 3 replicas of your nginx pod. It defines a selector field, so your deployment knows how to group specific pods as replicas. It works because all of those specific pods share a label. Their app is tagged as nginx.

Kubernetes

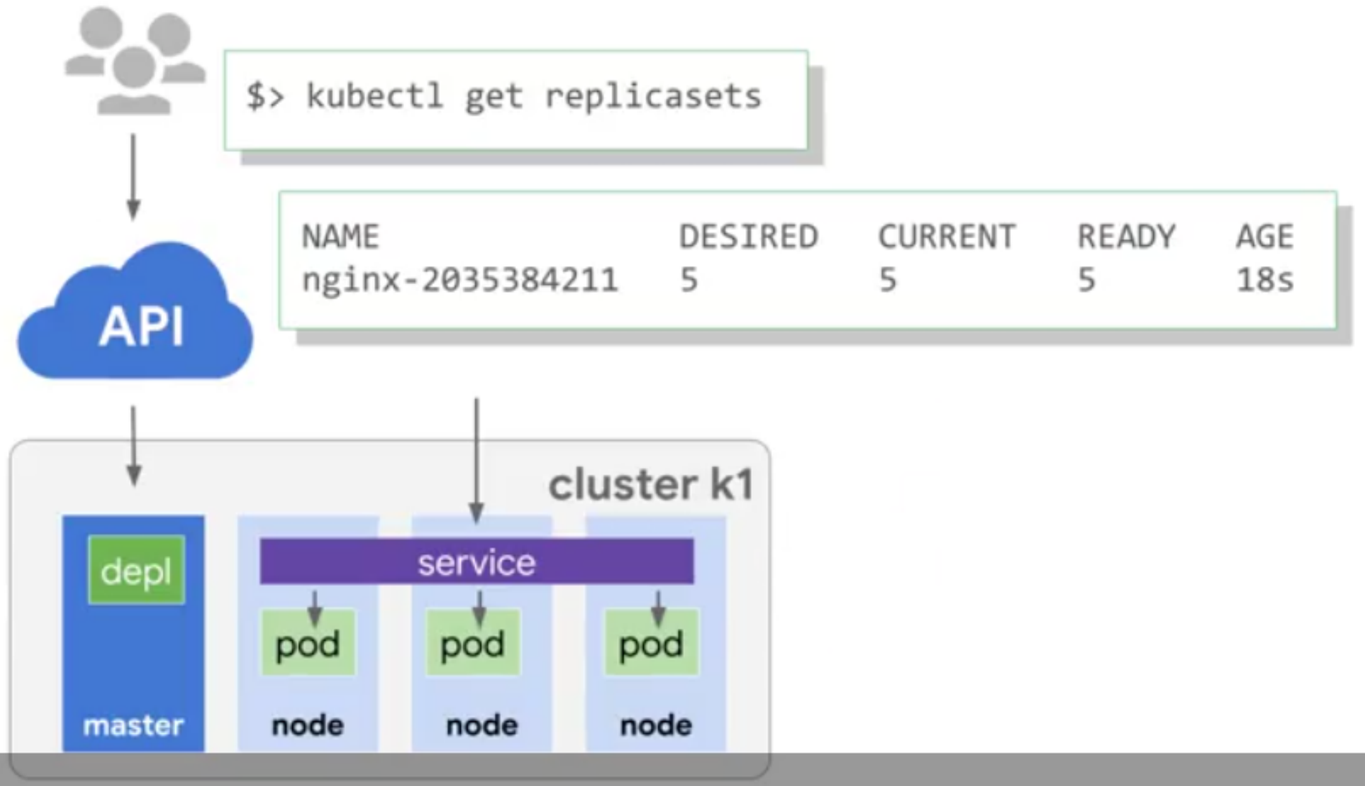
```
apiVersion: v1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.15.7
        ports:
        - containerPort: 80
```



But with a little familiarity,
they're easy to work with.

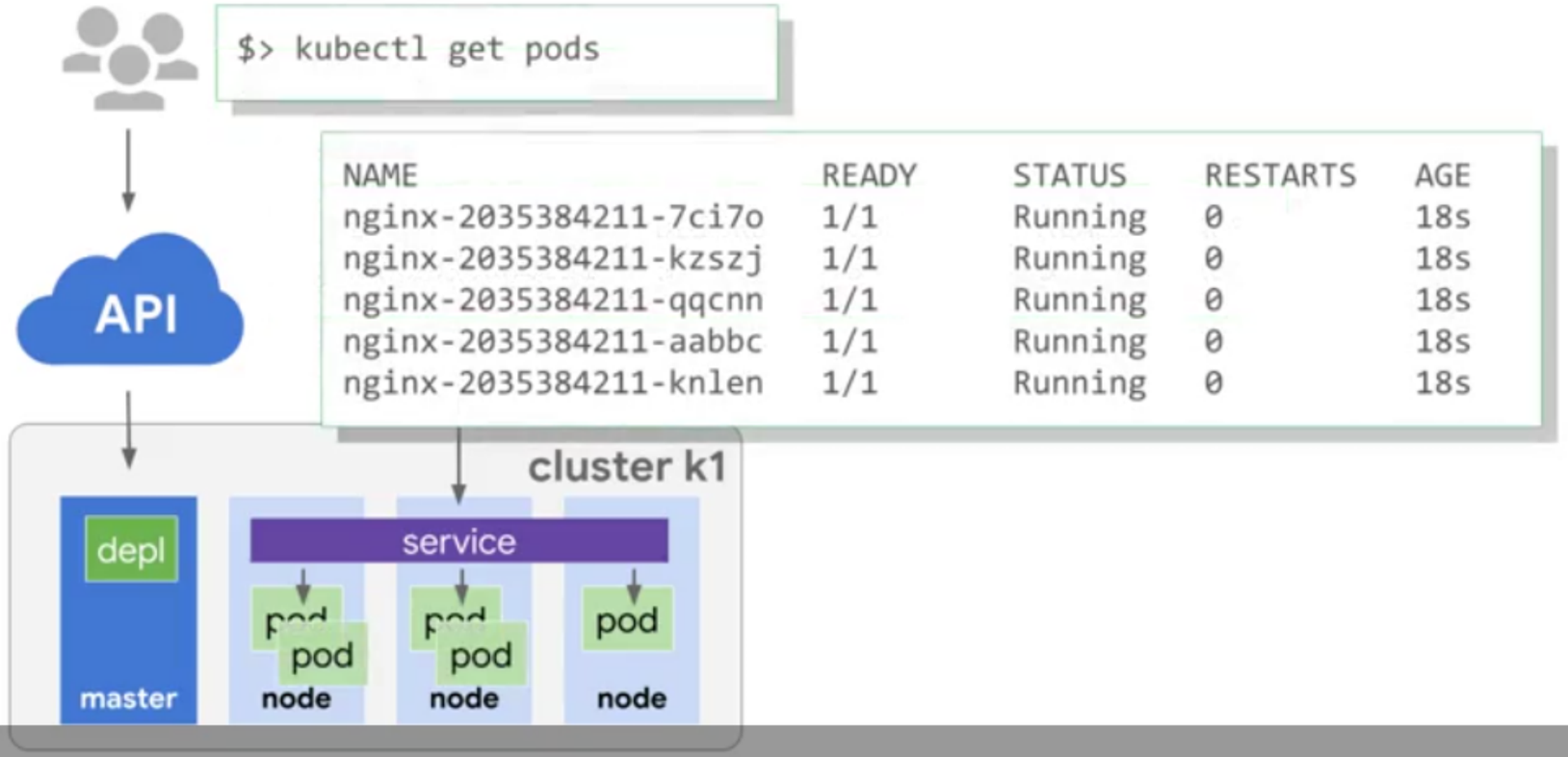

Kubernetes
$> kubectl apply -f
nginx-deployment.yaml
API
cluster k1
depl
master
service
pod
node
pod
node
pod
node
Google
And then run the kubectl apply command
to use the updated config file.

Kubernetes



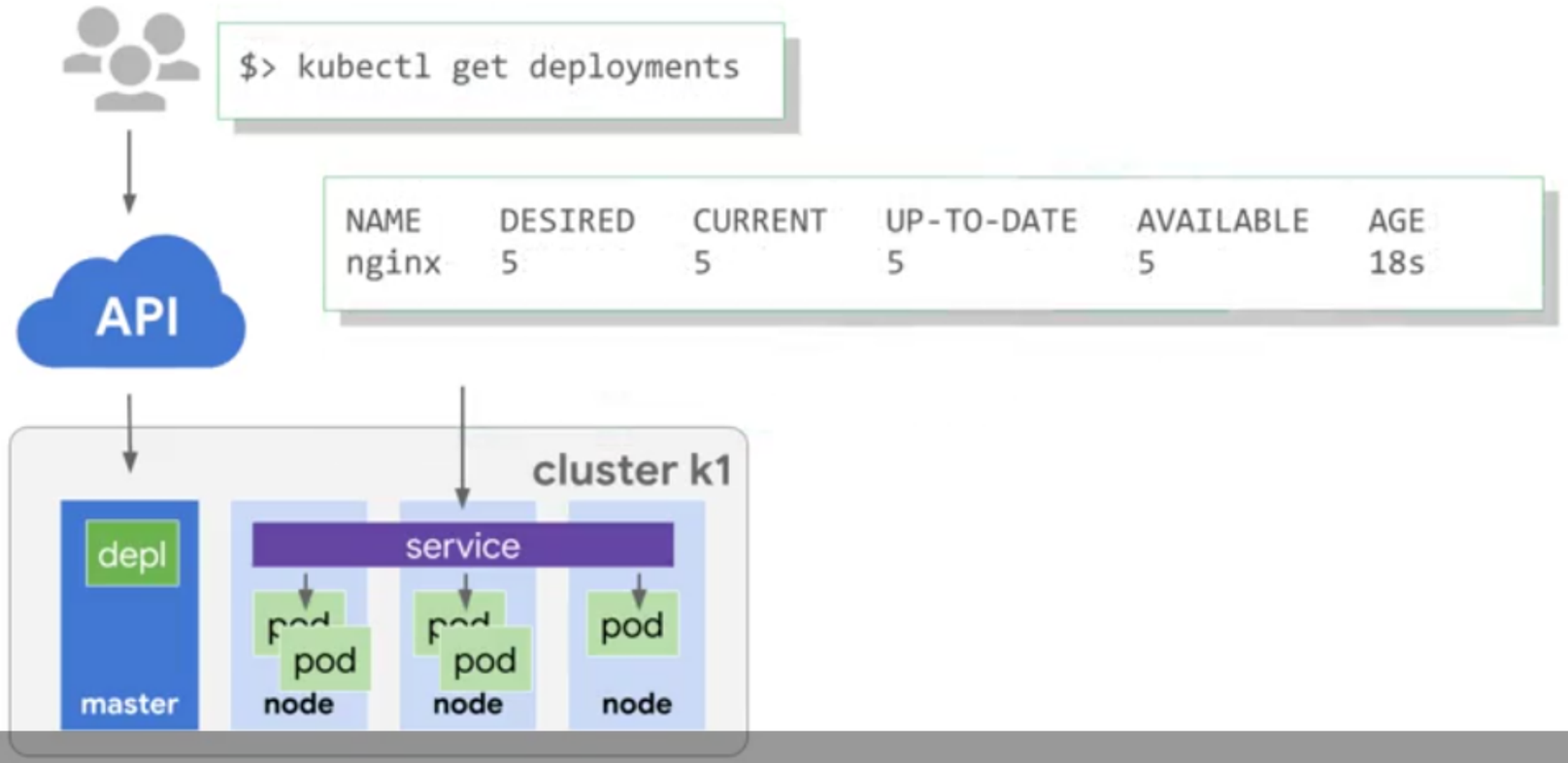
Now use the kubectl get replicasets
command to view your replicas and

Kubernetes



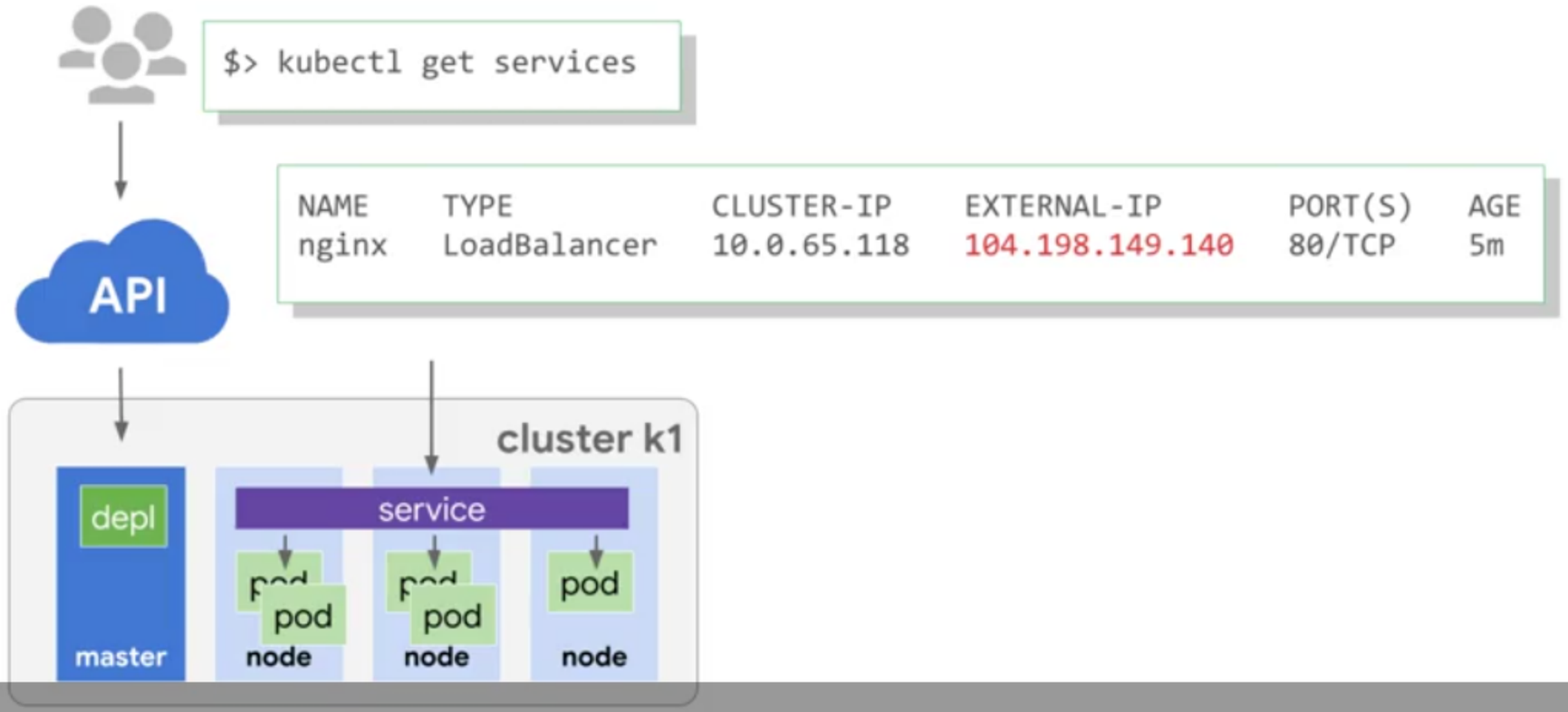
Then use the kubectl get pods command
to watch the pods come online.

Kubernetes

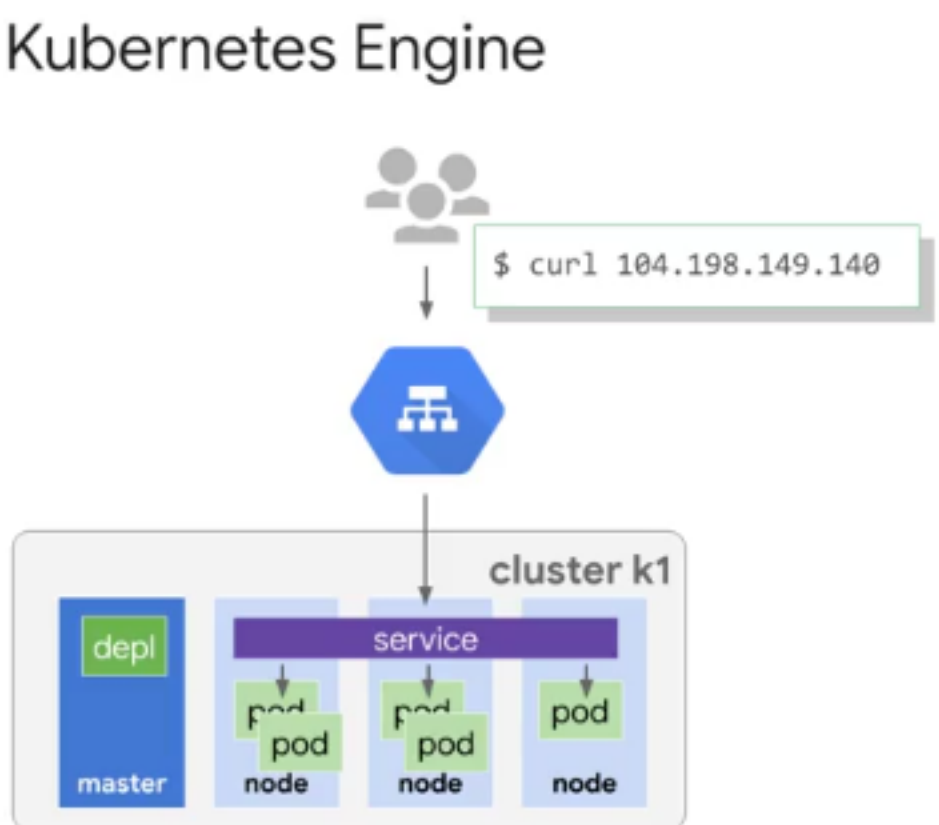


Finally, let's check the deployments
to make sure the proper number of

Kubernetes



And clients can still hit your endpoint,
just like before.



service is unaffected. Now you have 5 copies of your nginx pod running in GKE, and you have a single service that's proxying the traffic to all 5 pods.

This technique allows you to share the load and scale your service in Kubernetes. Remember the Python application you containerized in the previous lesson? You could have substituted it in place of nginx and used all the same tools to deploy and scale it too.

The last question we will answer is, what happens when you want to update the version of your application? You will definitely want to update your container and get the new code out in front of your users as

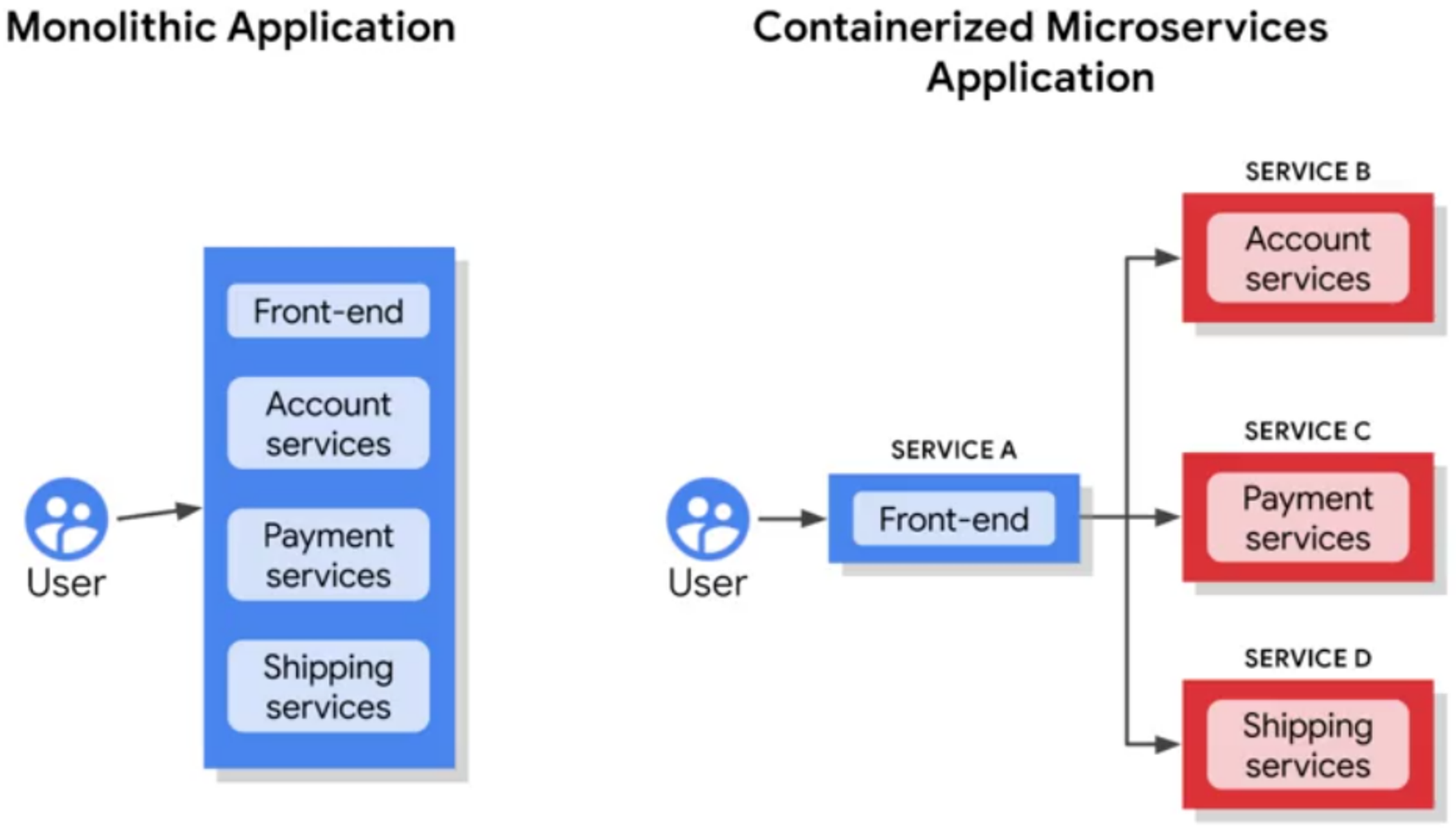
Kubernetes

```
spec:
  # ...
  replicas: 5
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
    type: RollingUpdate
  # ...
```

soon as possible, but it could be risky to roll out all those changes at once.

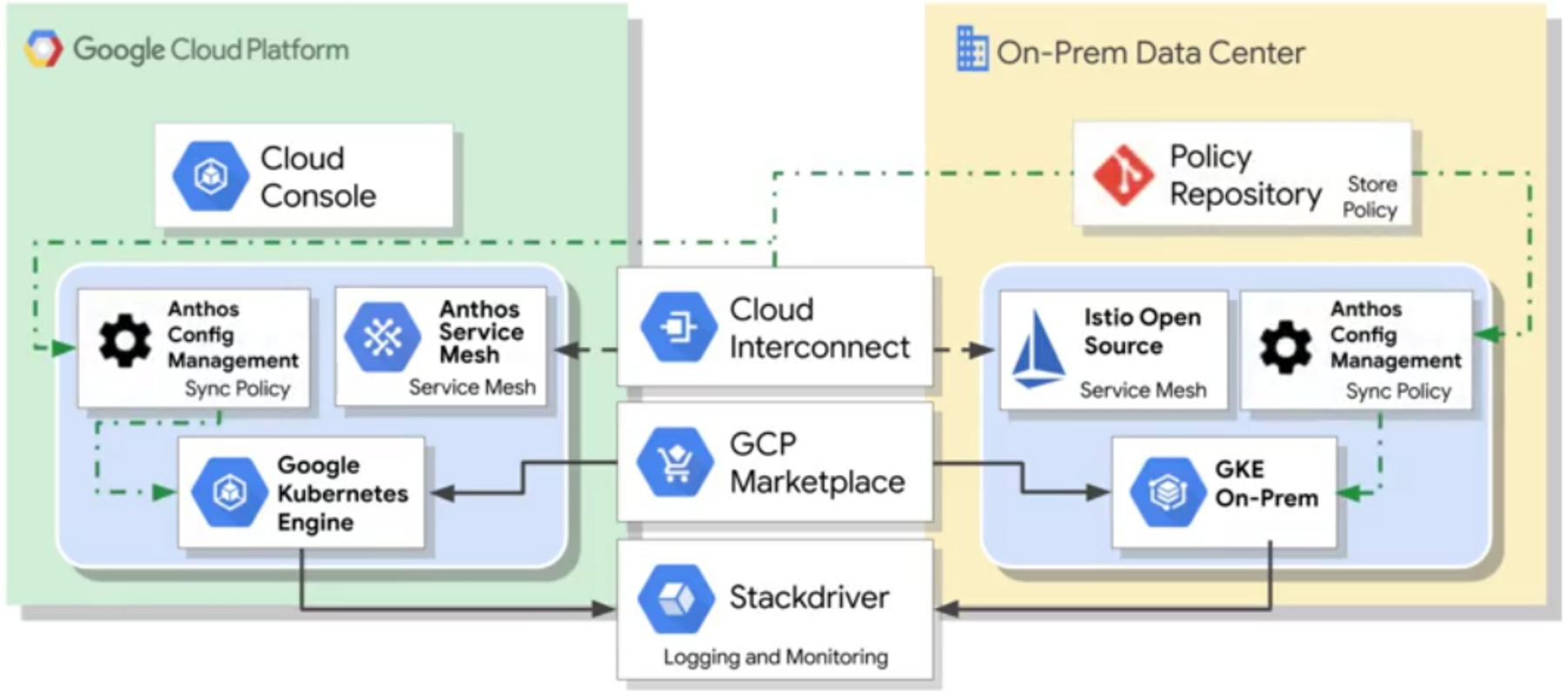
You do not want your users to experience downtime while your application rebuilds and redeploys. That's why one attribute of a deployment is its update strategy. Here's an example, a rolling update.

When you choose a rolling update for a deployment and then give it a new version of the software that it manages, Kubernetes will create pods of the new version one-by-one, waiting for each new version pod to become available before destroying one of the old version pods.



these workloads down into microservices,
so they can be more easily maintained and

Configuration Manager is the single source of truth



Lab

Getting Started with Kubernetes Engine

Brian Rice

There are a lot of features in Kubernetes and GKE we haven't even touched on such as configuring health checks, setting session affinity, managing different rollout strategies, and deploying pods across regions for high availability. But for now that's enough.

Let's get some hands-on experience with building and running containerized applications, orchestrating and scaling them on a cluster. Finally, deploying them using rollouts. Lets see how to do this in a demo, and then you'll practice it in a lab exercise.