

Introduction to Kubernetes

WELCOME!



Allen Sanders
Senior Technology Instructor





Join Us in Making Learning Technology Easier **Develop Intelligence**

Our mission...

Over 16 years ago, we embarked on a journey to improve the world by making learning technology easy and accessible to everyone.



...impacts everyone daily.

And it's working. Today, we're known for delivering customized tech learning programs that drive innovation and transform organizations.

In fact, when you talk on the phone, watch a movie, connect with friends on social media, drive a car, fly on a plane, shop online, and order a latte with your mobile app, you are experiencing the impact of our solutions.

Over The Past Few Decades, We've Provided

 Over **62,300,000**
expert-led learning hours

In 2019 Alone, We Provided





Upskilling and Reskilling Offerings



Intimately customized learning experiences just for your teams.



Workshop

2-3 day upskilling experiences



Fast Track

5-day reskilling experiences



Learning Spike

1-day technology overviews



Target Topics

90-minute instructor-led micro-learnings



Hack-a-thon

Learn and build an MVP in 2-3 days

BACK END DEVELOPMENT

BIG DATA

CLOUD COMPUTING

DEVOPS

FRONT END DEVELOPMENT

MACHINE LEARNING

MOBILE APP DEVELOPMENT

SOFTWARE ENGINEERING

SYSTEM ADMINISTRATION



AND MANY OTHER TRENDING TECHNOLOGIES



World Class Practitioners



Develop
Intelligence

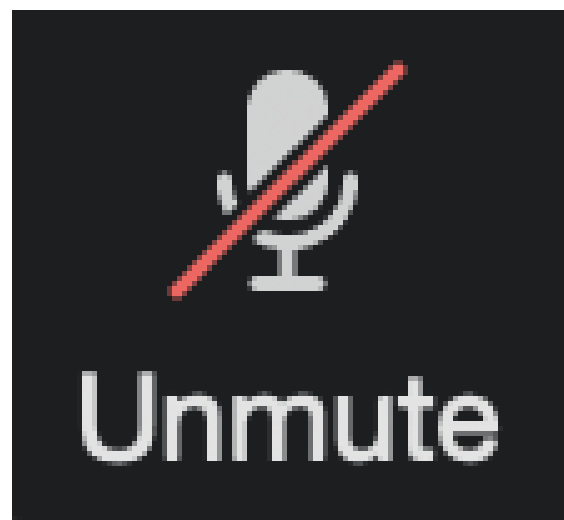




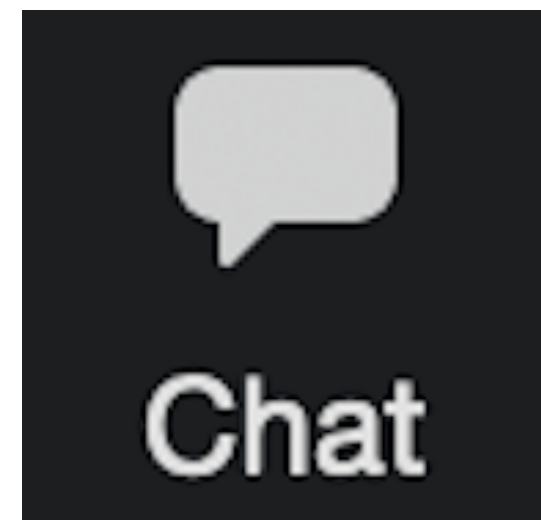
Virtual Training Expectations for You



Arrive on time / return on time



Mute unless speaking



Use chat or ask
questions verbally



Virtual Training Expectations for Me



I pledge to:

- Make this as interesting and interactive as possible
- Ask questions in order to stimulate discussion
- Use whatever resources I have at hand to explain the material
- Try my best to manage verbal responses so that everyone who wants to speak can do so
- Use an on-screen timer for breaks so you know when to be back



Prerequisites

- Working knowledge of 1 or more programming languages
- This course is designed for existing software developers seeking to learn more about Kubernetes and how to use it



Objectives

At the end of this course, you will be able to:

- Explain how containerization technologies like Docker integrate with Kubernetes
- Apply Kubernetes system architecture and the declarative approach to services, applications, storage, and networking
- Deploy an application on Kubernetes leveraging several of the k8s primitives available for container orchestration



Agenda

- Introduction to Kubernetes and container orchestration
- How do Docker and K8s fit together?
- Diving deeper into several Kubernetes primitives including Pods, Deployments, Services, ConfigMaps, Secrets, etc.
- Discussion of key areas of focus like configuration, observability, and persistence
- The architecture of a k8s cluster

Containerization as a Deployment Strategy





Why Containerization?

- Abstracts away the code implementation so you can deploy in a platform-agnostic manner, writing in the language of your choice
- Aligns strongly with the principles and practices of DevOps
- Helps leverage the power of the cloud
- Speeds up important non-coding activities (infrastructure spin-up, testing, CI/CD tasks, DevSecOps, code quality checks, etc.
- Helps breed consistency vs. “snowflake”



What Do We Mean by Application Hosting?



- The target infrastructure and runtime platform that will be employed for deployment and execution of an application or system
- Can include compute (CPU and server resources), storage, network, data and operating system



Application Hosting – An “Interesting” Example?

Here's an example of someone thinking “outside-of-the-box” when it comes to application hosting!

<https://mashable.com/article/pregnancy-test-doom/>



What Are the Hosting Options with Cloud?

- IaaS
- PaaS
- Serverless / FaaS
- SaaS
- Containers



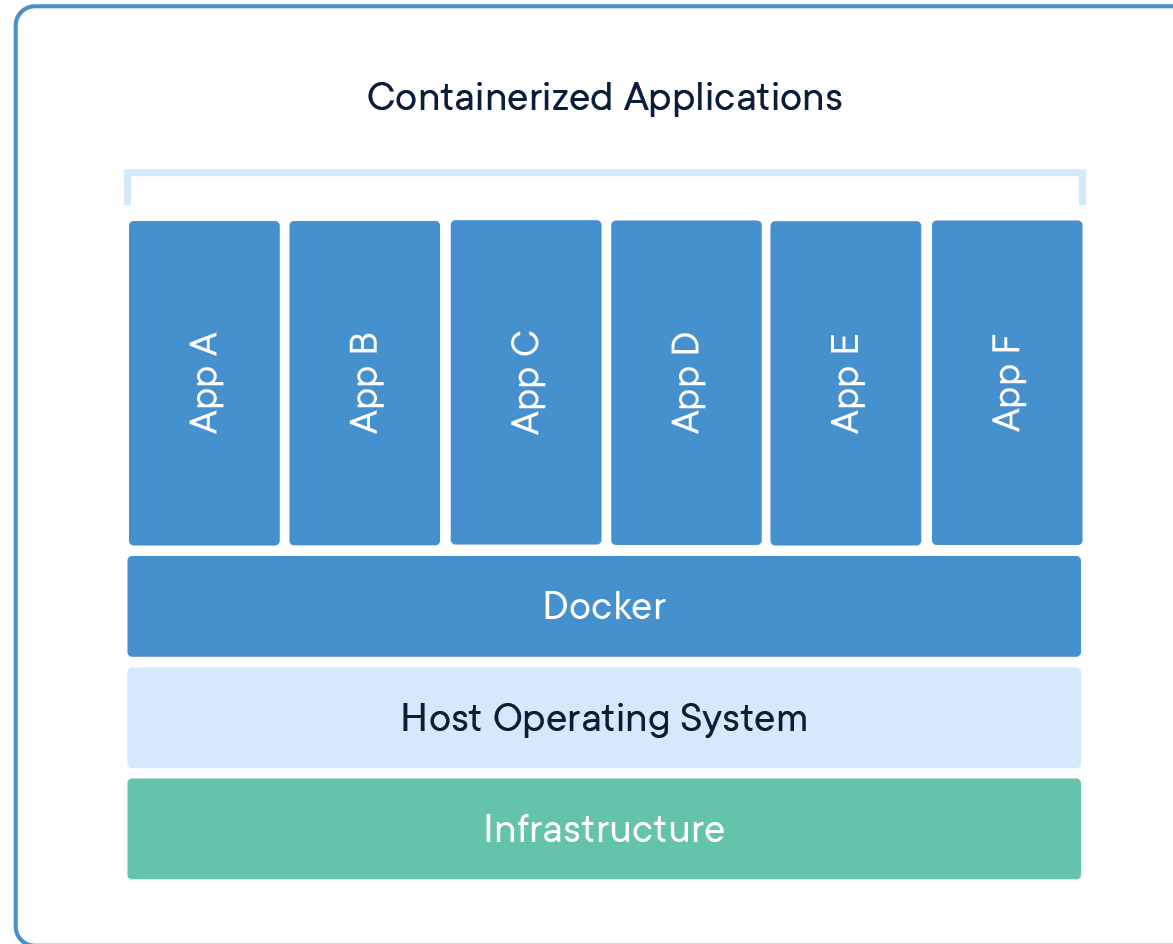
- Form of virtualization at the app packaging level (like virtual machines at the server level)
- Isolated from one another at the OS process layer (vs VM's which are isolated at the hardware abstraction layer)
- Images represent the packaging up of an application and its dependencies as a complete, deployable unit of execution (code, runtime and configuration)



- A platform (e.g., Docker) running on a system can be used to dynamically create containers (executable instances of the app) from the defined image
- Typically, much, much smaller than a VM which makes them lightweight, quickly deployable and quick to “boot up”
- An orchestration engine (e.g., Kubernetes) might be used to coordinate multiple instances of the same container (or a “pod” of containers) to enable the servicing of more concurrent requests (scalability)



Containers





What are Microservices?

- An architectural style in which a distributed application is created as a collection of services that are:
 - Highly maintainable and testable
 - Loosely coupled
 - Independently deployable (by extension, independently-scalable)
 - Domain centric and organized around business capabilities
- The Microservices architecture enables the continuous deployment and delivery of large, complex distributed applications

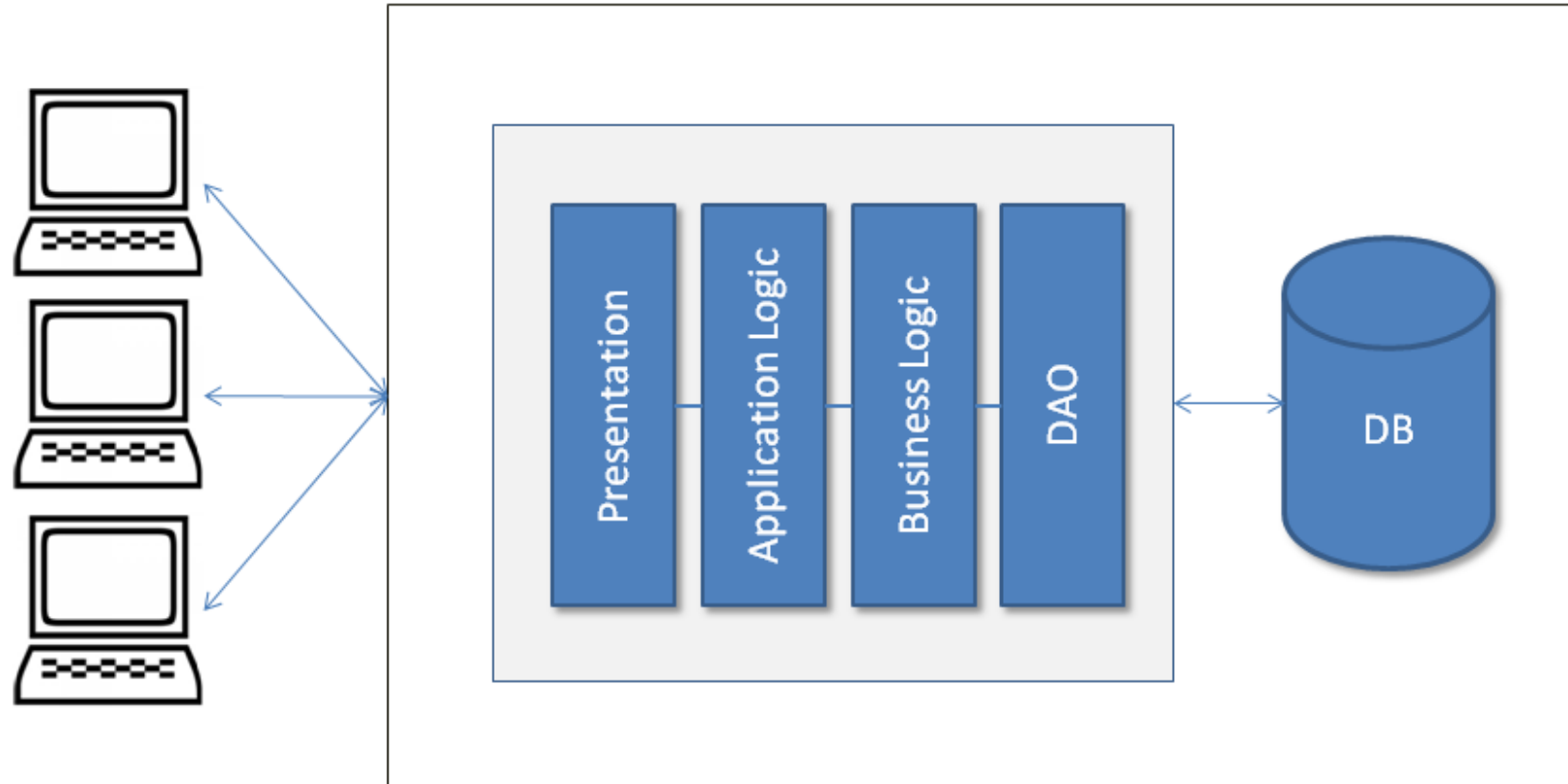


What is a Monolith?

- Typical enterprise application
- Large codebases
- Set technology stack
- Highly coupled elements
- Whole system affected by failures
- Scaling requires the duplication of the entire app
- Minor changes often require full rebuild



Monolithic Architecture Example

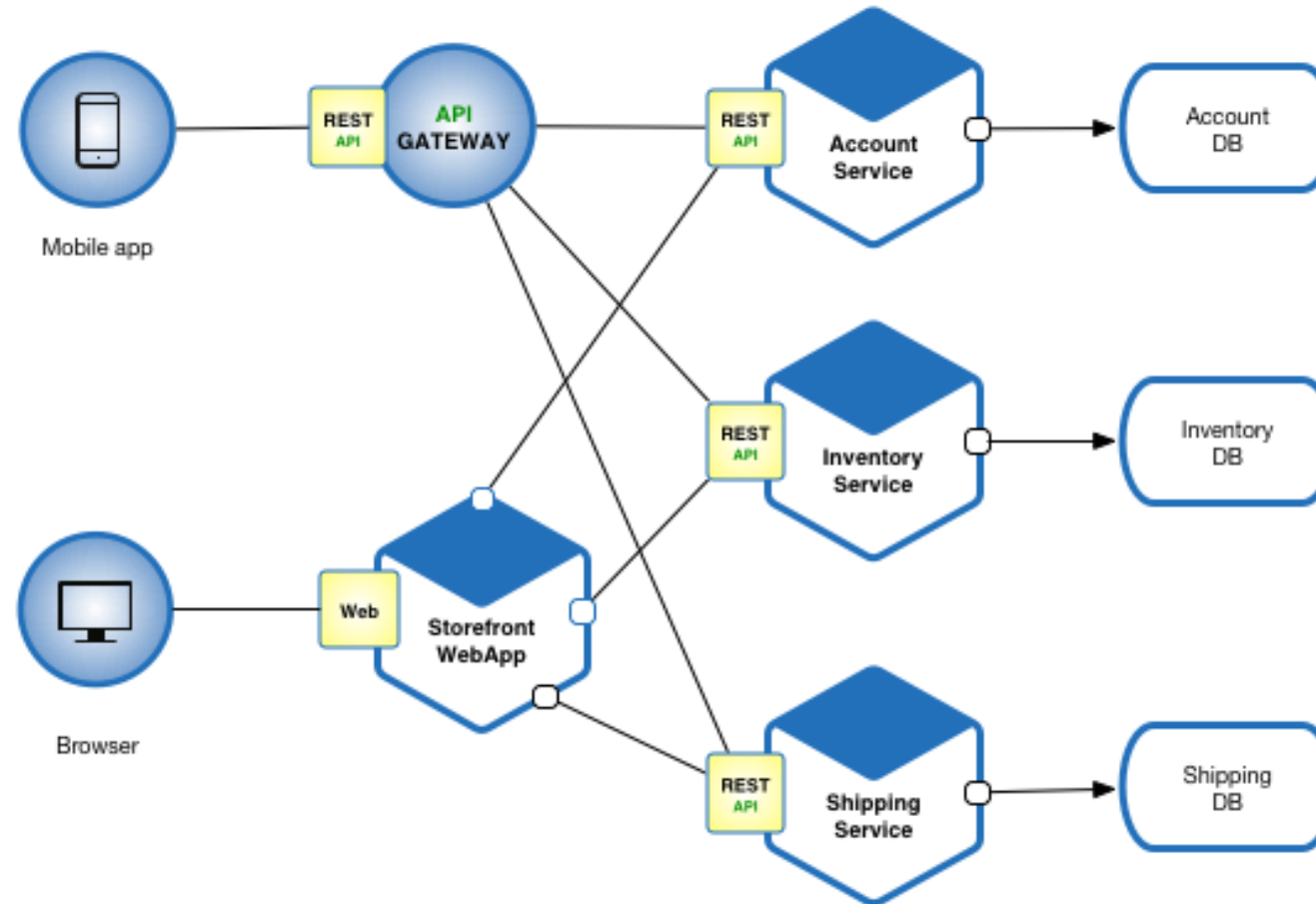




- We have moved towards a refinement of “what is a service?”
- Even in SOA, we can create a service that does more than one thing – using it to solve an overall business problem, rather than one part of the actions needed to solve that problem
 - This was moving us back towards monolithic
- We arrive at a point where we want individual, purposeful services that do one thing and do it well – the microservice



Example Microservices Architecture





Microservices – Benefits vs. Costs

Benefits:

- Enables work in parallel
- Promotes organization according to business domain
- Advantages from isolation
- Flexible in terms of deployment and scale
- Flexible in terms of technology



Microservices – Benefits vs. Costs

Costs:

- Requires a different way of thinking
- Complexity moves to the integration layer
- Organization needs to be able to support re-org according to business domain (instead of technology domain)
- With an increased reliance on the network, you may encounter latency and failures at the network layer
- Transactions must be handled differently (across service boundaries)



Microservices & Good Architecture

- With microservices, key concepts:
 - Cohesion (goal to increase)
 - Coupling (goal to decrease)

- SOLID principles & DDD (Domain Driven Design) lay the foundation for:
 - Clean, logical organization of components
 - Maintainability
 - Clear boundaries, encapsulation and separation of concerns in the components used to build out complex systems
 - Techniques that minimize coupling
 - Being “surgical” with our change



SOLID principles help us build testable code

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)



Single Responsibility Principle (SRP)

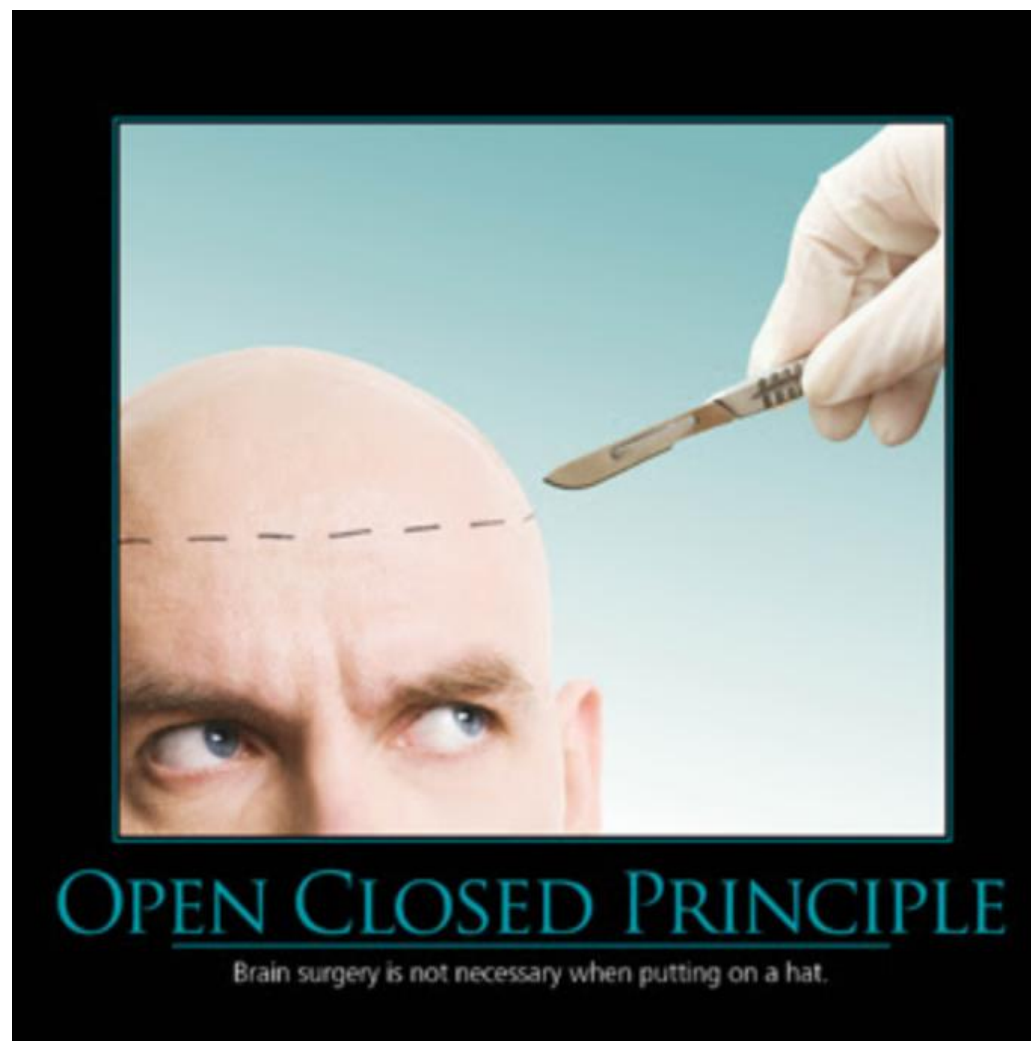


SINGLE RESPONSIBILITY PRINCIPLE

Every object should have a single responsibility, and all its services should be narrowly aligned with that responsibility.

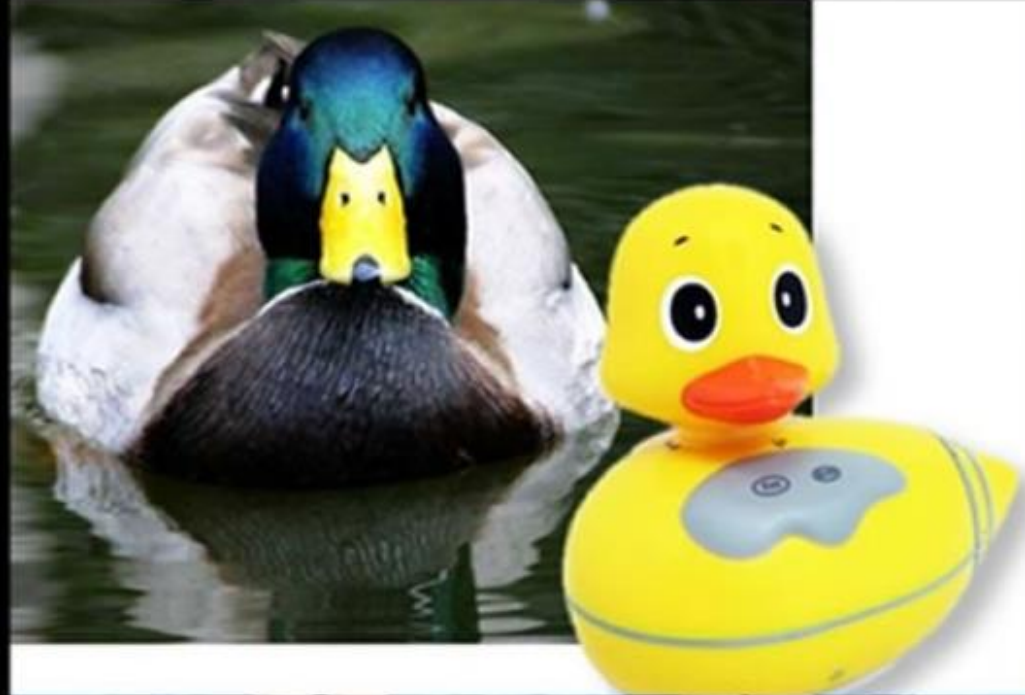


Open-Closed Principle (OCP)





Liskov Substitution Principle (LSP)



Liskov Substitution Principle

If it looks like a duck and quacks like a duck but needs batteries, you probably have the wrong abstraction.



Interface Segregation Principle (ISP)



INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?



Dependency Inversion Principle (DIP)



DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?



Microservices & Containers

- Microservices – with their smaller size, independently-deployable and independently-scalable profile, and encapsulated business domain boundary – are a great fit for containers
- Using Kubernetes, sophisticated systems of integrated microservices can be built, tested and deployed
- Leveraging the scheduling and scalability benefits of Kubernetes can help an organization target scaling across a complex workflow in very granular ways
- This helps with cost management as you can toggle individual parts of the system for optimized performance



Microservices & the Cloud

- Microservices have broad support across multiple Cloud providers
- One option includes standing up VM's (IaaS) and installing / managing a Kubernetes cluster on those machines or
- Another option includes leveraging a managed service (PaaS) provided by the CSP
- Microservices are a great option for the Cloud because the elastic scalability provided by the Cloud infrastructure can directly support the independent scalability needed with a microservices architecture



- Applications that can
 - Efficiently scale
 - Are flexible
 - Are high performance
- These apps are powered by multiple services, working in concert
- Each service is small and has a single focus
- We use a lightweight communication mechanism to coordinate the overall system
- This ends up giving us a technology agnostic API



Stateless Applications and Scaling

- If state is held in your application, scaling is very hard. Why?
- Where can we get state info if it's not held in the application?
- If we build to resist failure, scaling is hard. Why?
- How can we build to embrace failure?



Modern Approaches to Application Development



- Leverage the cloud
- Abstract away as much of the non-code elements as possible
- Use the tools that arise from DevOps (automation, feedback, etc.)
- Integrate testing and code quality checks early (“Shift Left”)



Design Approaches to Modern Applications



- Decompose when possible
- Embrace Failure
- Separate the language from the function of the code



Infrastructure as Code (IaC)

- Declarative vs. Imperative coding
- Drawbacks of declarative infrastructure
- Impact of IaC on the dev process, on testing, and on deployment

Quick Refresh on Docker





What is Docker?

- Built around the concept of images & containers
- Also, supports composition of a set of containers to be deployed together
- For example, application code + network components + database components



What is Docker?

- Utilizes principles of “immutable infrastructure”
- Complete application environments torn down and recreated as needed
- Helps to minimize infrastructure “drift” and environment inconsistencies



Setting Up a Docker Dev Environment

- Tools:
 - Docker Desktop
 - Code editor/IDE that plays nice with containers (e.g., VS Code)



The Dockerfile

- Tells Docker what to do in creating an image for your application
- The commands are all things you could do from the CLI
- Used by the docker “build” command
- Docker build uses this file and a “context” – a set of files at a specified location – to make your image



Dockerfile example

- The following creates an image for building/running Java app in container
- See <https://github.com/KernelGamut32/dockerlab-repo-sample> for sample

```
Dockerfile X
Dockerfile > ...
1  # Grabs OpenJDK image upon which the new image will be based
2  FROM openjdk:17
3
4  # Creates a new target folder in image
5  RUN mkdir /usr/src/JavaDemoApp
6
7  # Copies current directory contents to newly created folder
8  COPY . /usr/src/JavaDemoApp
9
10 # Switches working directory in image to app folder
11 WORKDIR /usr/src/JavaDemoApp
12
13 # Compiles/builds Java app
14 RUN javac JavaDemo.java
15
16 # Executes new Java app
17 CMD ["java", "JavaDemo"]
18 |
```




Docker Images

- Represent templates defining an application environment
- New instances of the application can be created from the image
- These instances are called containers



Docker Images

- Images are defined via a Dockerfile definition
- Support layers for building up the environment in stages
- Fully defines the application, including all components required to support



Docker Images

- Those components can include:
 - Runtime
 - Development framework
 - Source code
 - Executable instructions for container startup



Docker Images

- Start with a base that gives your app a place to live
 - Needed OS/runtimes/dB server applications, etc.
- Examples:
 - nginx
 - Node
 - MySQL
 - Apache HTTP Server
 - IIS with .NET Runtimes



- Centralized registry for image storage & sharing
- Can signup for an account – user accounts offer both free and pro versions
- Also, supports organizations for grouping of multiple team members



Docker Hub

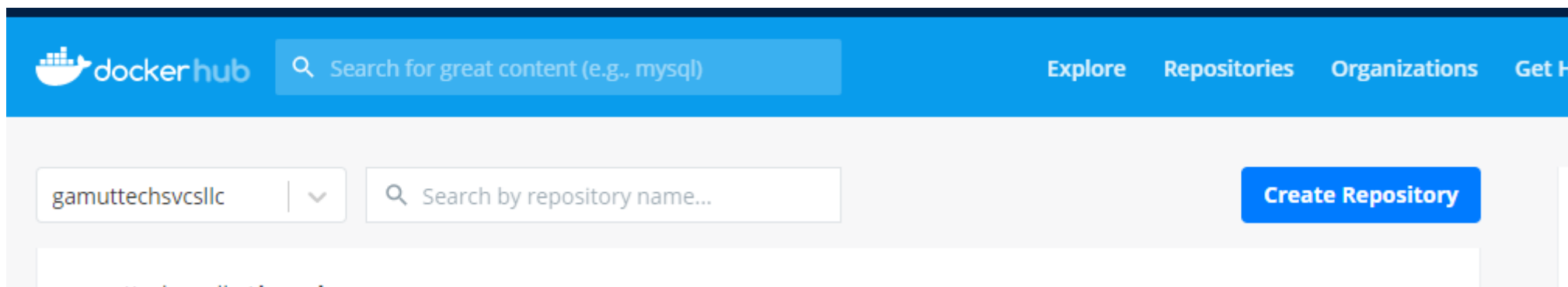
- Accessible at <https://hub.docker.com>
- Search feature enables search for image by technology or keyword
- Image detail displays available tags and image variants

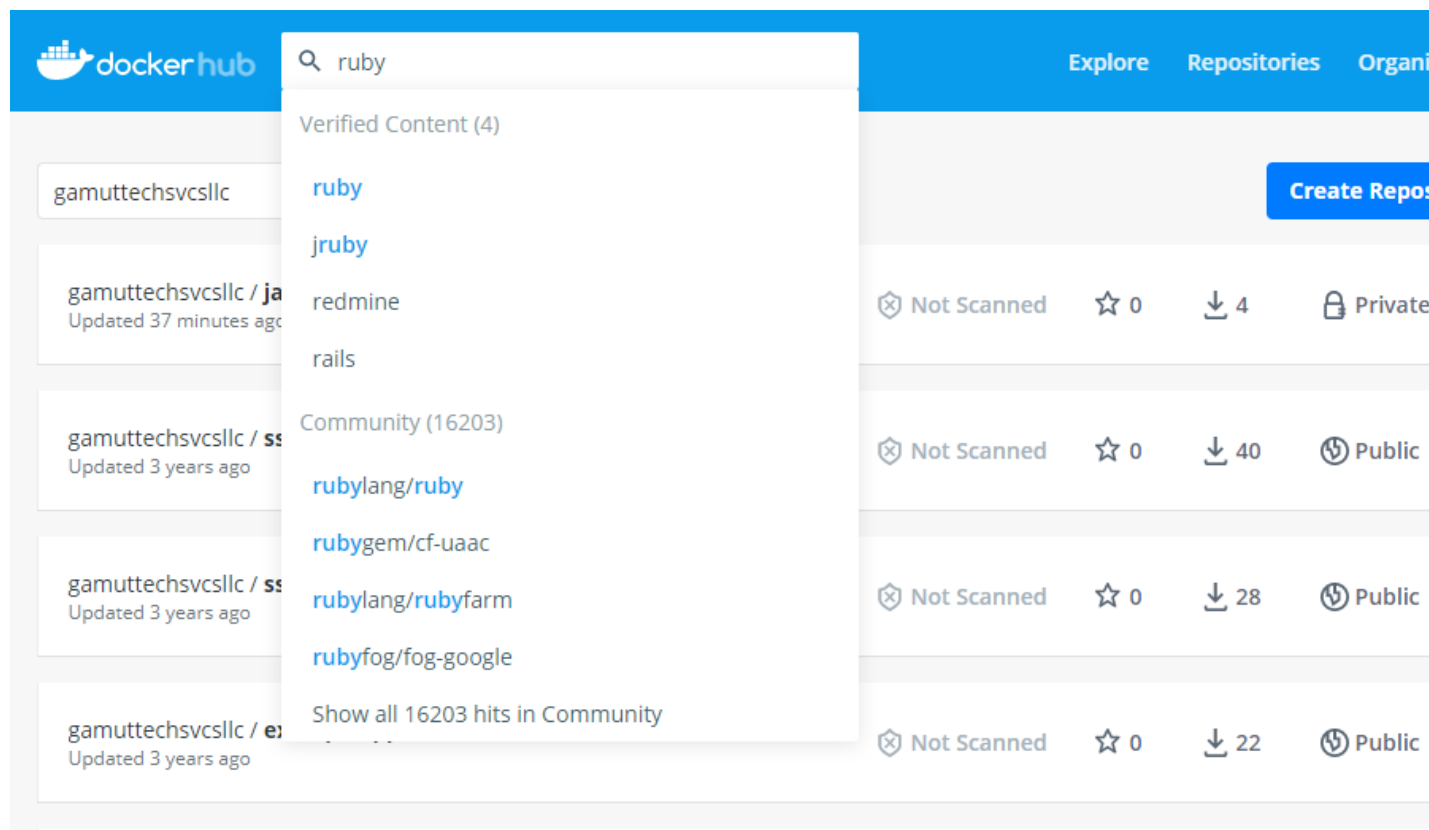


- May also include examples of usage
- Pro account supports image scanning for security vulnerabilities
- Can be useful for image reuse and image sharing across a dev team



- There are other registry types, including private registries










The screenshot shows the Docker Hub search interface. The search bar at the top contains the text "ruby". Below the search bar, a dropdown menu displays search results. The results are categorized into "Verified Content (4)" and "Community (16203)". The "Verified Content" section lists four items: "ruby", "jruby", "redmine", and "rails". The "Community" section lists four items: "rubylang/ruby", "rubygem/cf-uaac", "rubylang/rubyfarm", and "rubyfog/fog-google". At the bottom of the dropdown, there is a link to "Show all 16203 hits in Community". The background of the screenshot shows the Docker Hub homepage with a list of repositories on the left and a table of repository details on the right. The table includes columns for repository name, status (Not Scanned), stars, downloads, and visibility (Private or Public).

Repository	Status	Stars	Downloads	Visibility
gamuttechsvcsllc / ja	Not Scanned	0	4	Private
gamuttechsvcsllc / ss	Not Scanned	0	40	Public
gamuttechsvcsllc / ss	Not Scanned	0	28	Public
gamuttechsvcsllc / es	Not Scanned	0	22	Public




 dockerhub


[Explore](#) [Repositories](#) [Organizations](#) [Get Help](#) [gamuttechsvcsllc](#) 


 Docker  Containers  Plugins

Filters

Images

☐ Verified Publisher 


☐ Official Images 
Official Images Published By Docker

Categories 

☐ Analytics

☐ Application Frameworks

1 - 25 of 16,207 results for **ruby**. [Clear search](#)



ruby

Updated a day ago

Ruby is a dynamic, reflective, object-oriented, general-purpose, open-source programming language.

Container

Linux

386

x86-64

ARM 64


IBM Z

mips64le

ARM

PowerPC 64 LE

Programming Languages


OFFICIAL IMAGE 

10M+

2.0K

Downloads

Stars

OFFICIAL IMAGE 



ruby ☆

[Docker Official Images](#)

Ruby is a dynamic, reflective, object-oriented, general-purpose, open-source programming language.

↓ 100M+

Container

Linux

PowerPC 64 LE

386

x86-64

ARM 64

IBM Z

mips64le

ARM

Programming Languages

Official Image

Description

Reviews

Tags

Copy and paste to pull this image

```
docker pull ruby
```



[View Available Tags](#)



How to use this image

Create a `Dockerfile` in your Ruby app project

```
FROM ruby:2.5

# throw errors if Gemfile has been modified since Gemfile.lock
RUN bundle config --global frozen 1

WORKDIR /usr/src/app

COPY Gemfile Gemfile.lock ./
RUN bundle install

COPY . .

CMD ["./your-daemon-or-script.rb"]
```

Put this file in the root of your app, next to the `Gemfile`.

You can then build and run the Ruby image:

```
$ docker build -t my-ruby-app .
$ docker run -it --name my-running-script my-ruby-app
```

Generate a `Gemfile.lock`



- To build the image from Dockerfile use *docker build*
- *docker build -t <tag name> <path to Dockerfile>*
- For example, *docker build -t java-demo .*
- Builds image from Dockerfile in current folder (.) with tag name “java-demo”



Building The Image

- For tag name, can include optional detail:
 - Docker ID in Docker Hub for eventual push to image registry
 - Version identifier for tag – defaults to “latest” if excluded
- For example, *`docker build -t <docker ID>/<tag name>:<version> .`*



Application Bootstrapping with Docker and k8s



- Kubernetes provides a hosting environment for containerized applications
- Once you have a Docker image, you can work entirely within Kubernetes to deploy your app



Open Container Initiative (OCI)

- The OCI is an open governance structure for the express purpose of creating open industry standards around container formats and runtimes
- Docker started it
- They have two specs: runtime-spec and image-spec
- This deals with containers in the abstract



Competing Container Runtimes

- [rkt](#) from CoreOS
- [Mesos](#) from Apache
- [LXC](#) Linux containers



Set up Local k8s Dev Environment



Activate Kubernetes now, via Docker Desktop

Use kubectl from command line to see k8s config data and confirm setup

Optional: Install Docker/k8s extensions/plugins for your IDE

Kubernetes and Container Orchestration





Kubernetes (k8s) Overview

- Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services

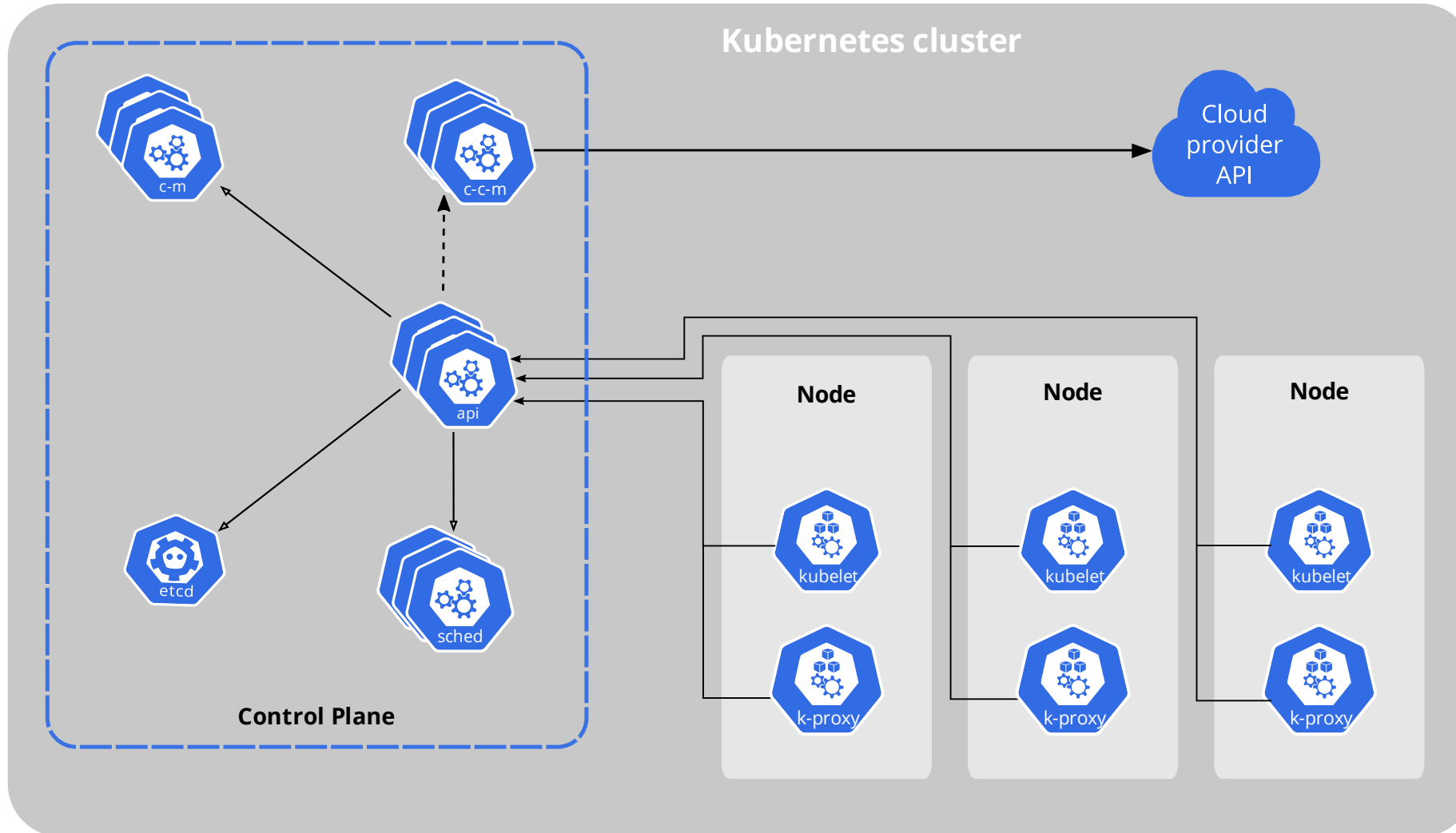


What is an Orchestrator and Why Do We Need It?

- What would it look like to manually control the containers needed for your application as your app scales or containers fail?
- “Orchestration” is the execution of a defined workflow
- We can use an orchestrator to start and stop containers automatically based on your set rules
- What does this open up for you?



Architecture of k8s System





Core Components of k8s

- When you deploy Kubernetes, you get a cluster.
- A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node.
- The worker node(s) host the Pods that are the components of the application workload.
- The control plane manages the worker nodes and the Pods in the cluster.



k8s Resource/Manifest

- Kubernetes manifests are text files, usually written in YAML
- They are used to create, modify and delete Kubernetes resources



Kubernetes Architecture

- Kubernetes automates and monitors the lifecycle of a stateless application
- It can scale the app up or down and ensure it keeps running
- Kubernetes cluster consists of computers called nodes
- The basic unit of work in kubernetes is called a pod, which is a group of one or more containers
- Kubernetes can be divided into planes
 - Control plane - the actual pods that the kubernetes core components runs on, exposing the api, etc...
 - Data Plane - everything else meaning the nodes and pods which the application runs on



Kubernetes Architecture

- In short, control plane is what monitors the cluster, schedules the work, makes the changes, etc...
- The nodes are what actually do the real work, report back to the master, and watch for changes



Kubernetes Control Plane & Data Plane

- Kubernetes is actually a collection of pods implementing the k8s api and the cluster orchestration logic (AKA the Kubernetes control plane)
- The application/data plane is everything else, meaning all the nodes that host all the pods that the application runs on
- The *controllers* of the control plane implement control loops that repeatedly compare the desired state of the cluster to its actual state
- When the state of the cluster changes, controllers take action to bring it back inline with desired state



Declarative and Desired State

- Kubernetes supports a declarative model – we can send the api a yaml file in a declarative form
- Desired state means that we specify in the yaml file our desired state and the cluster takes responsibility to make sure it will happen
- We describe the desired state using a yaml or json file that serves as a record of intent, but we do not specify how to get there (this is kubernetes responsibility to get us there)
- Things could change or go wrong over the lifetime of the cluster (node failing, etc...), Kubernetes is responsible to always make sure that the desired state is kept intact
- Kubernetes control plane controllers are always running in a loop and checking that the actual state of the cluster matches the desired state, so that if any error occurs they kick in and rectify the cluster



Reconciling state

- Watch for the `spec` fields in the YAML files
- The *spec* describes *what we want the thing to be*
- Kubernetes will *reconcile* the current state with the spec (technically, this is done by a number of *controllers*)
- When we want to change a resource, we update the *spec*, and reapply
- Kubernetes will then *converge* that resource



k8s Pods: The Basic Building Block

- A pod represents a set of running containers in your cluster
- Worker nodes host pods
- The control plane manages the worker nodes and the pods inside them



k8s Pods: The Basic Building Block

- Pod is the basic execution and scaling unit in Kubernetes
- Kubernetes runs containers but always inside pods
- It is like a sandbox in which containers run (abstraction)
- A pod is a group of containers:
 - Running together (on the same node)
 - Sharing resources (RAM, CPU; but, also, network, volumes, etc...)
- A Pod models an application-specific “logical host”



- Pods are considered to be relatively ephemeral (rather than durable) entities
- Pods do not hold state, so if a pod crashes, Kubernetes will replace it with another
- Multiple instances of the same pod are called replicas

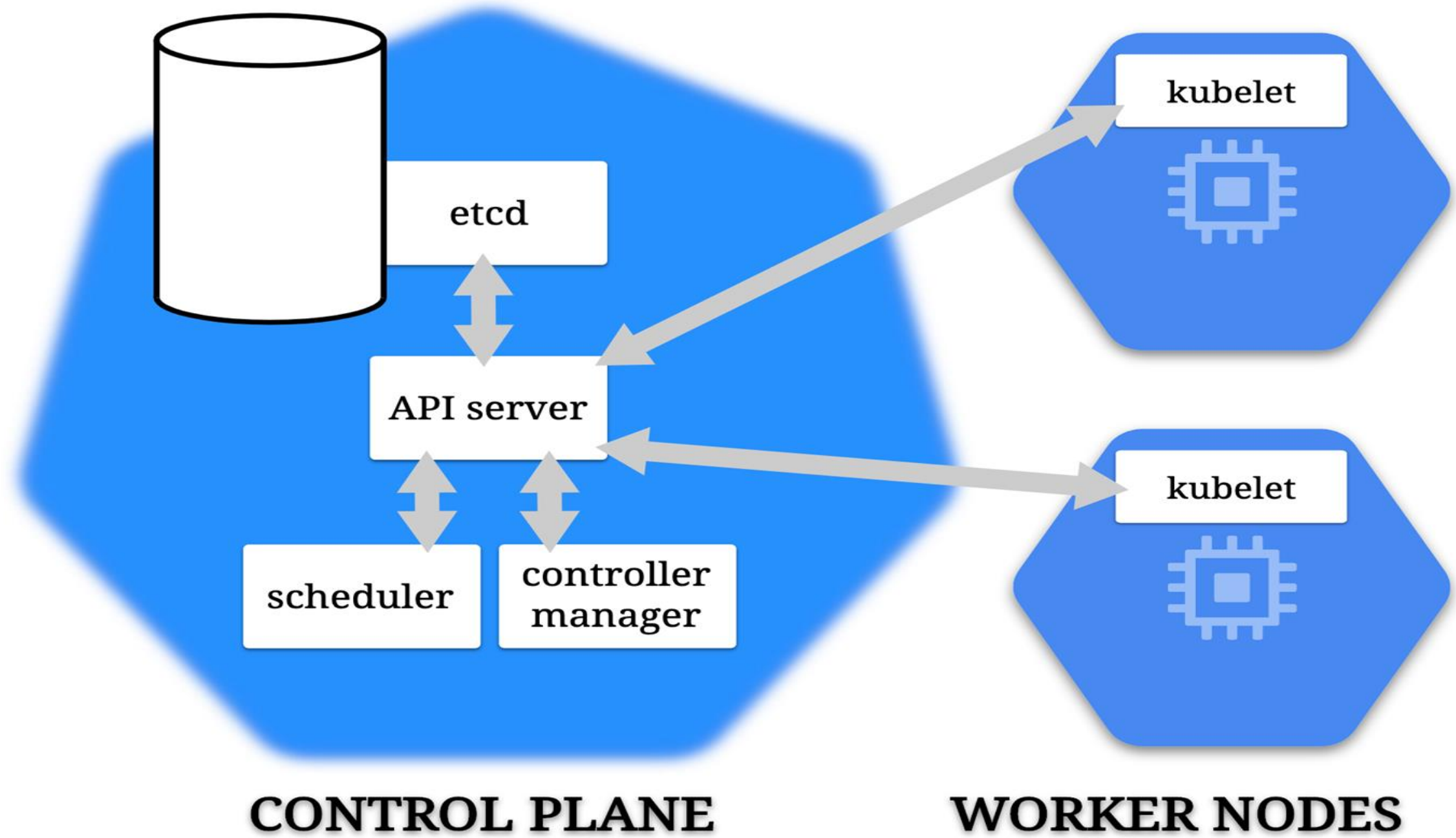


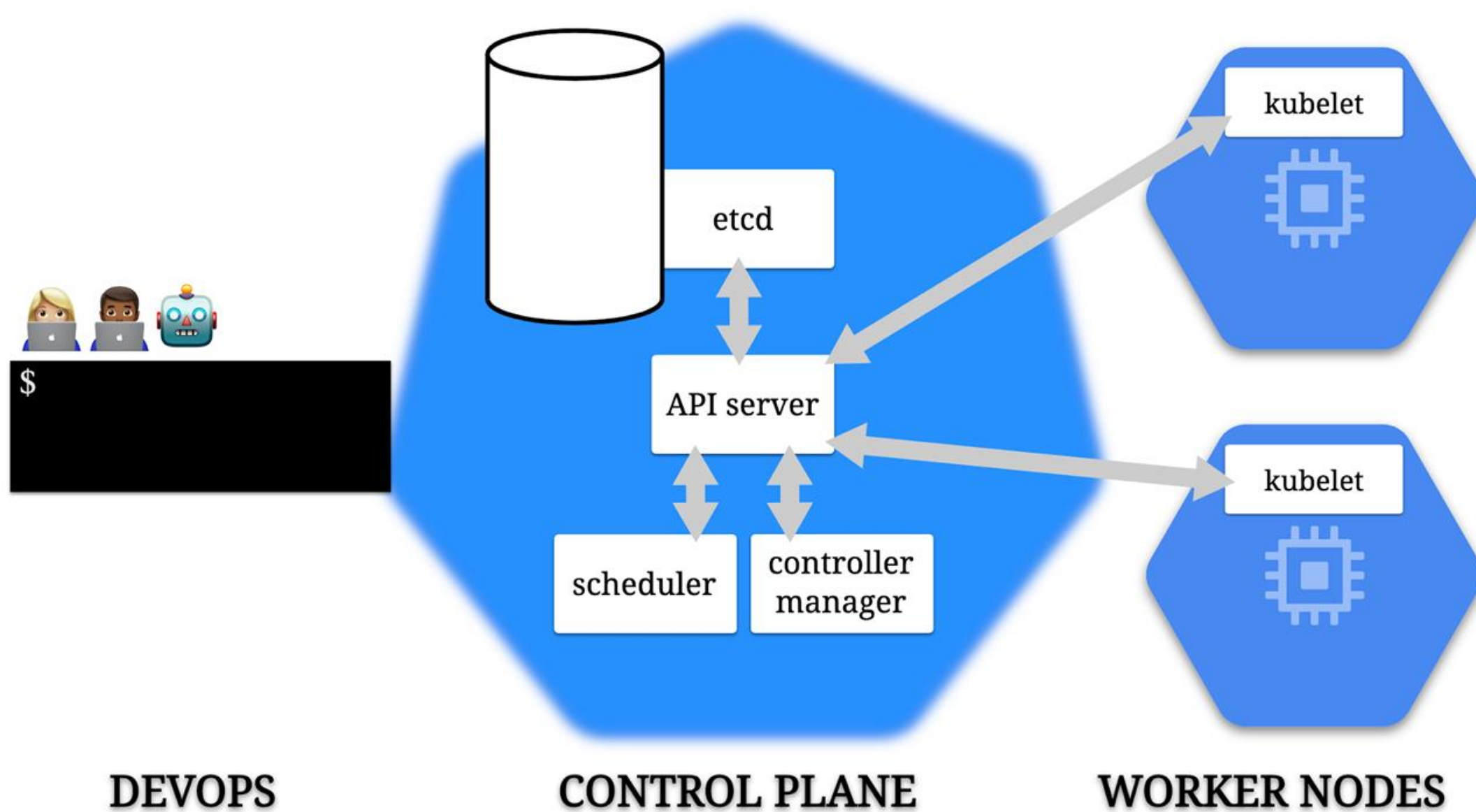
- A *Deployment* controller provides declarative updates for Pods and ReplicaSets
- The Deployment Object gives us a better way of handling the scaling of pods
- The advantage of using Deployment versus using a replicaset is having rolling updates support for the pod container versions out-of-the-box

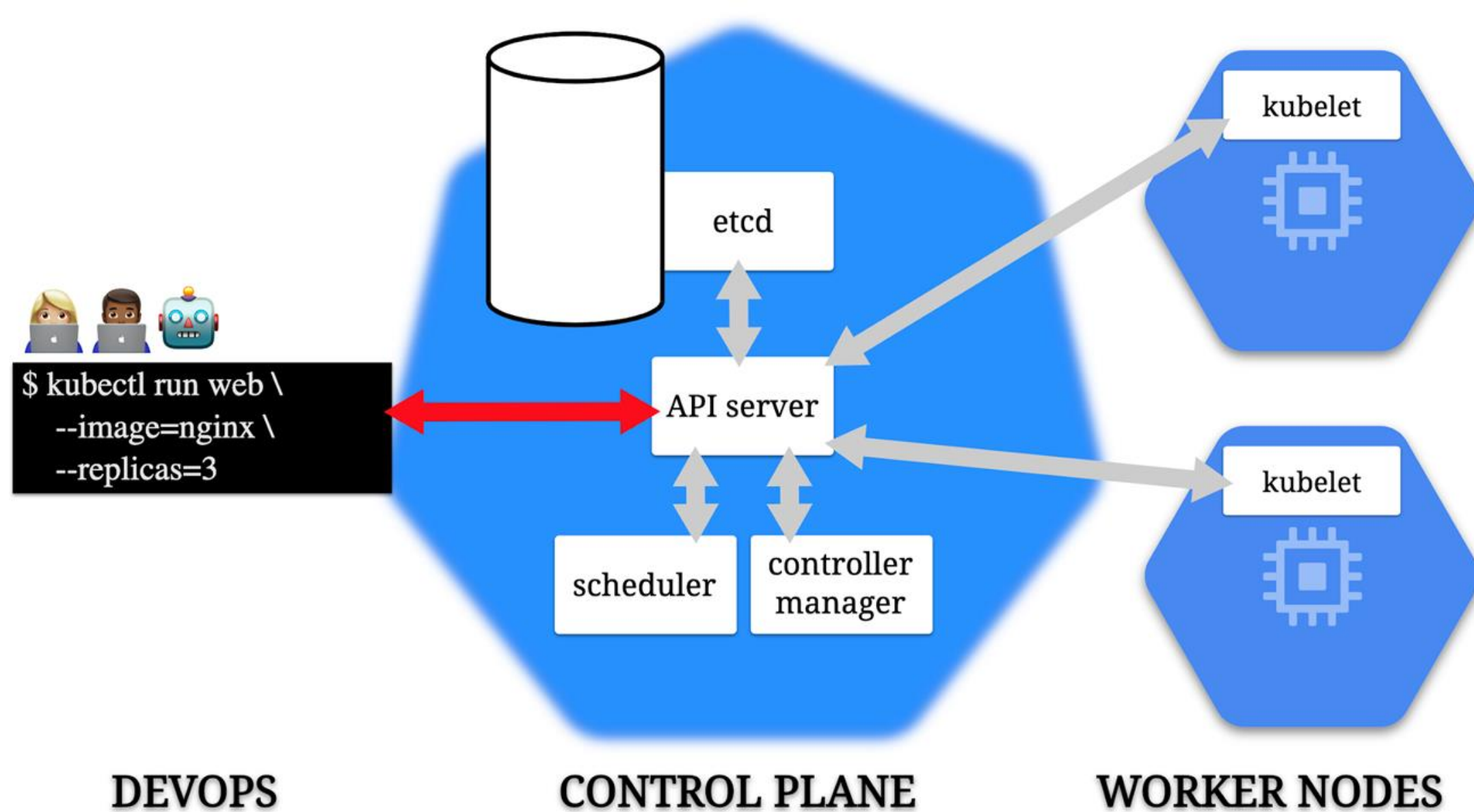


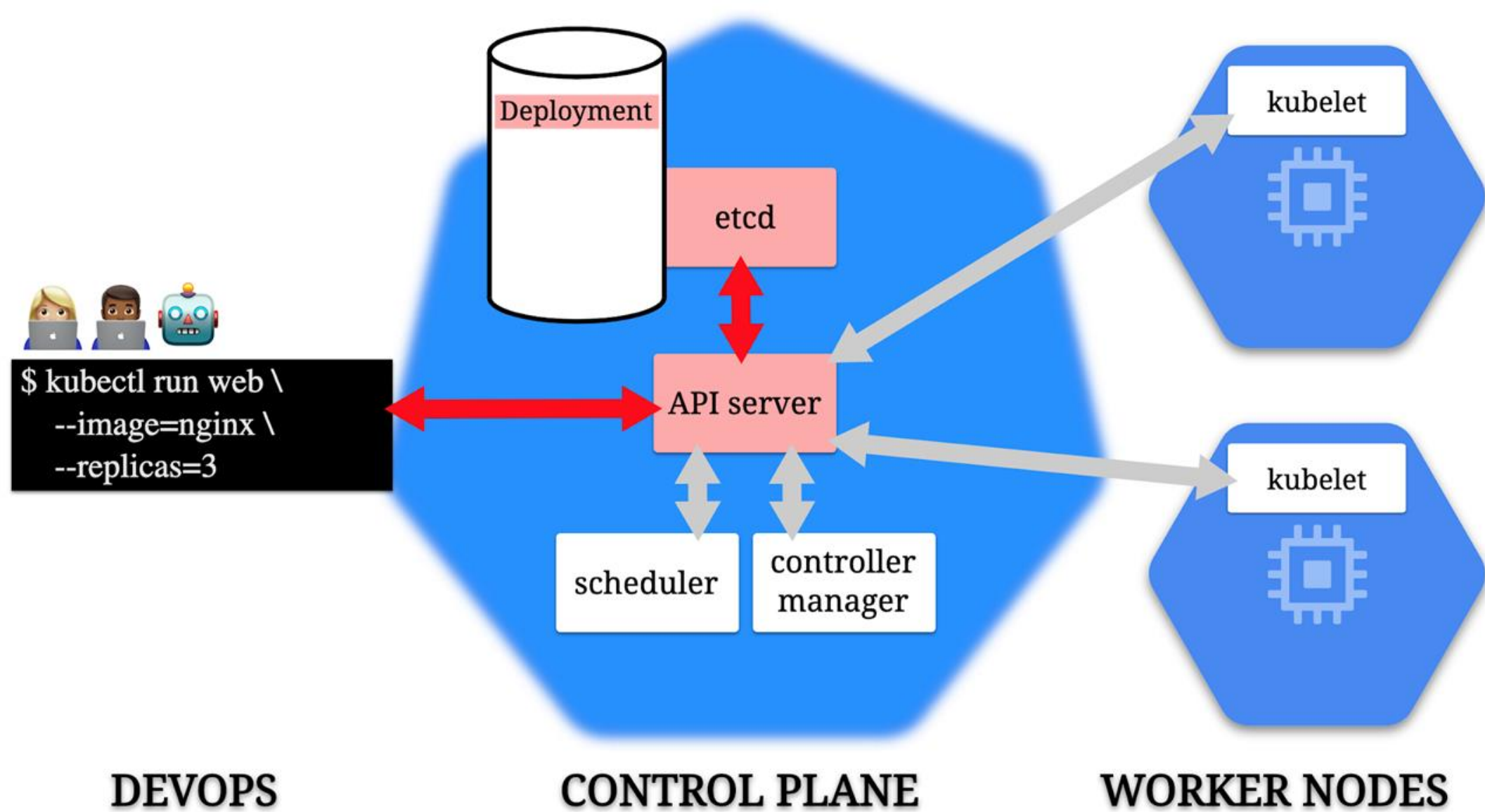
Deployments

- Every time the application code changes, a new version of the application container is built, and then there is a need to update the Deployment manifest with the new version and tell K8s to apply the changes
- K8s will then handle the rolling-out of this newer version, terminating pods with the old version as it spins up the new pods with the updated container
- This means that at some point we will have multiple versions of the same application running at the same time





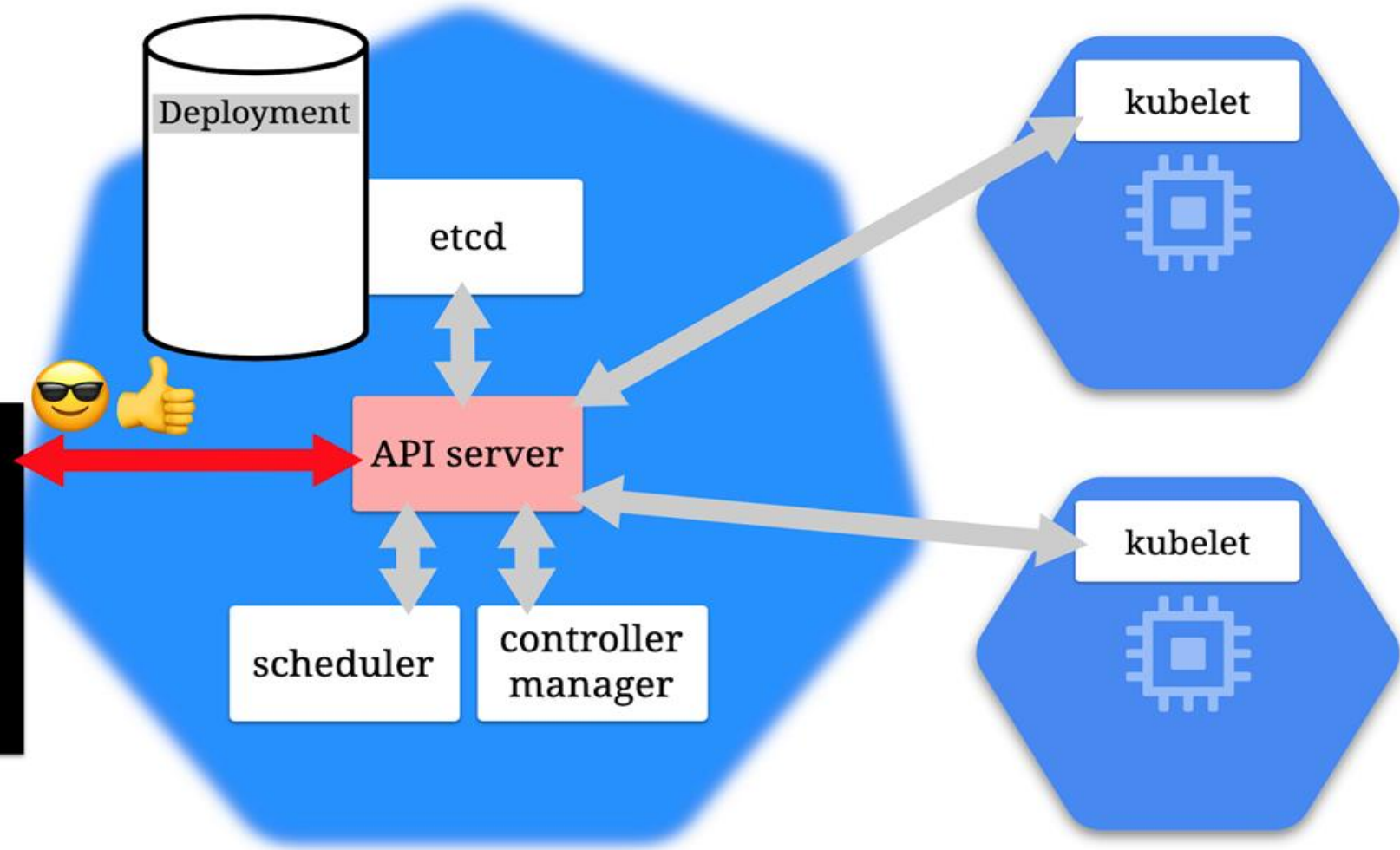






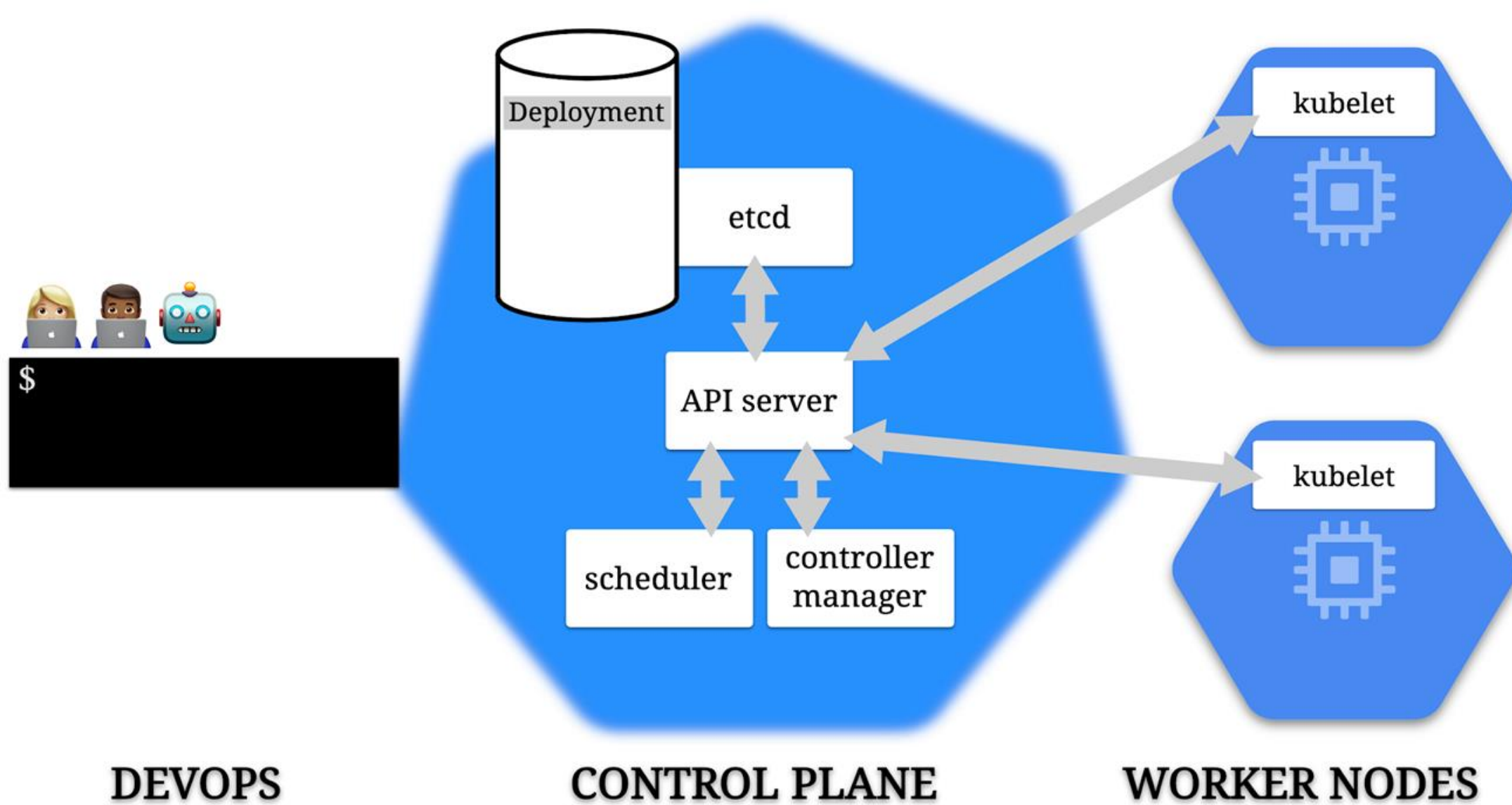
```
$ kubectl run web \
  --image=nginx \
  --replicas=3
...
deployment.apps/web
created
$
```

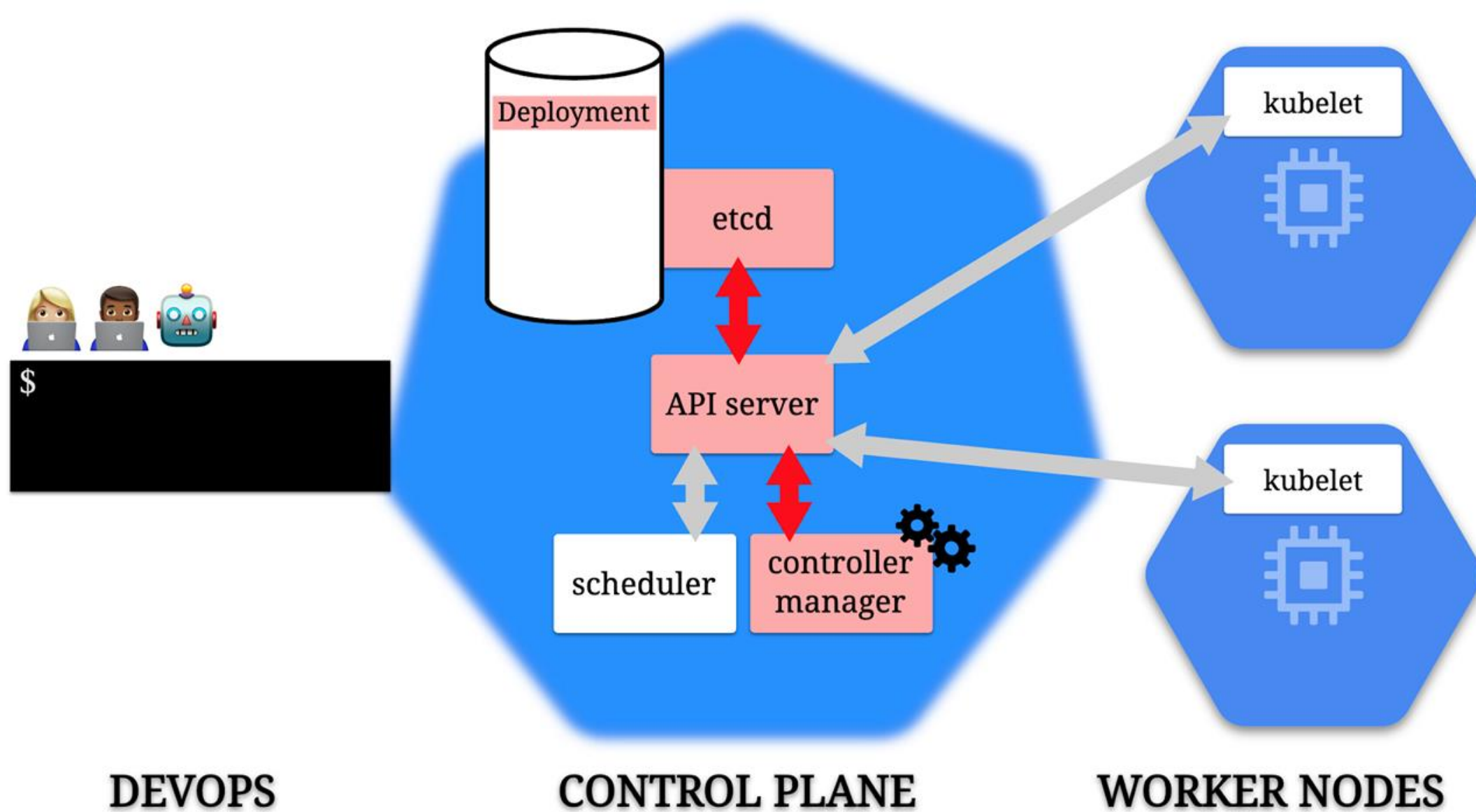
DEVOPS

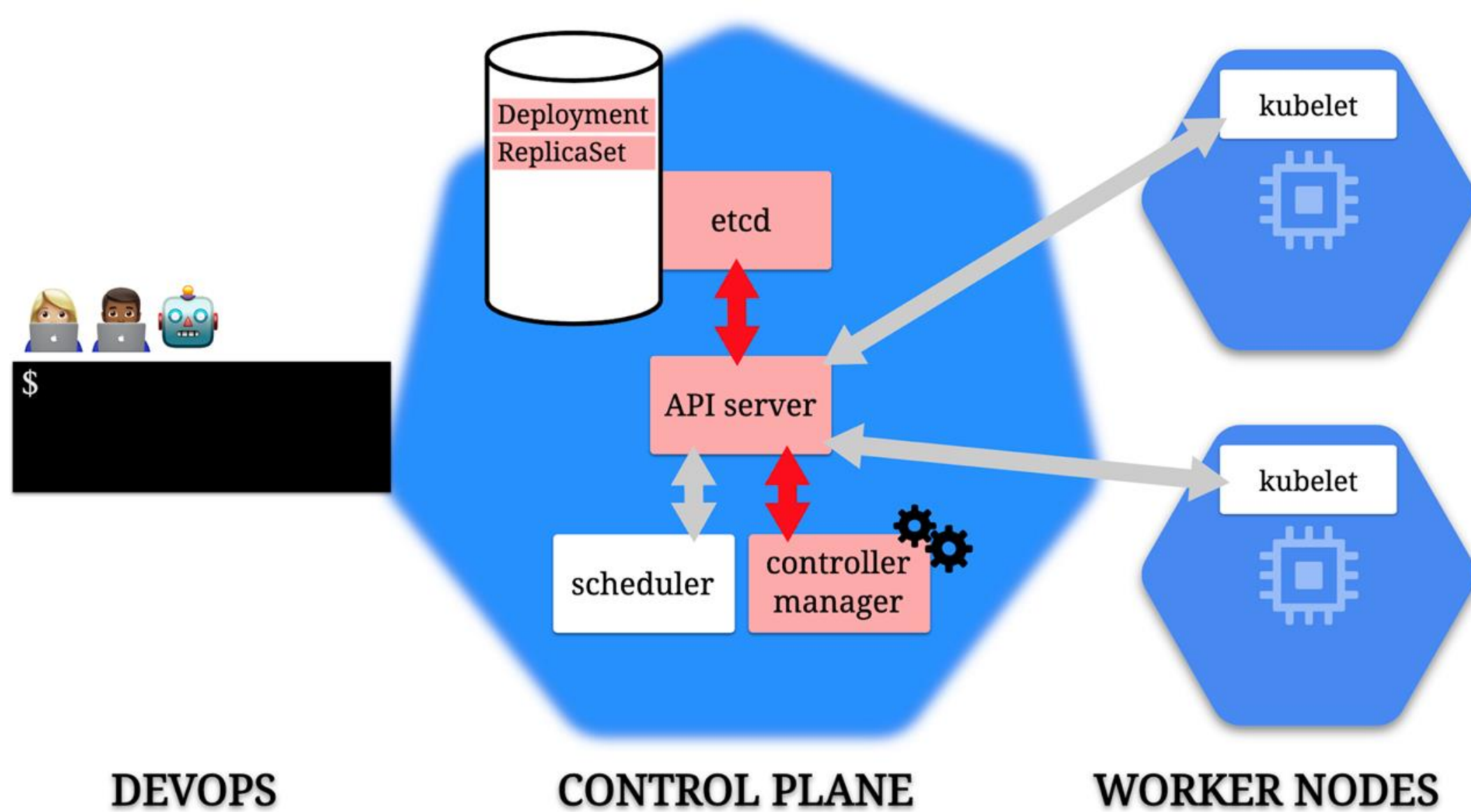


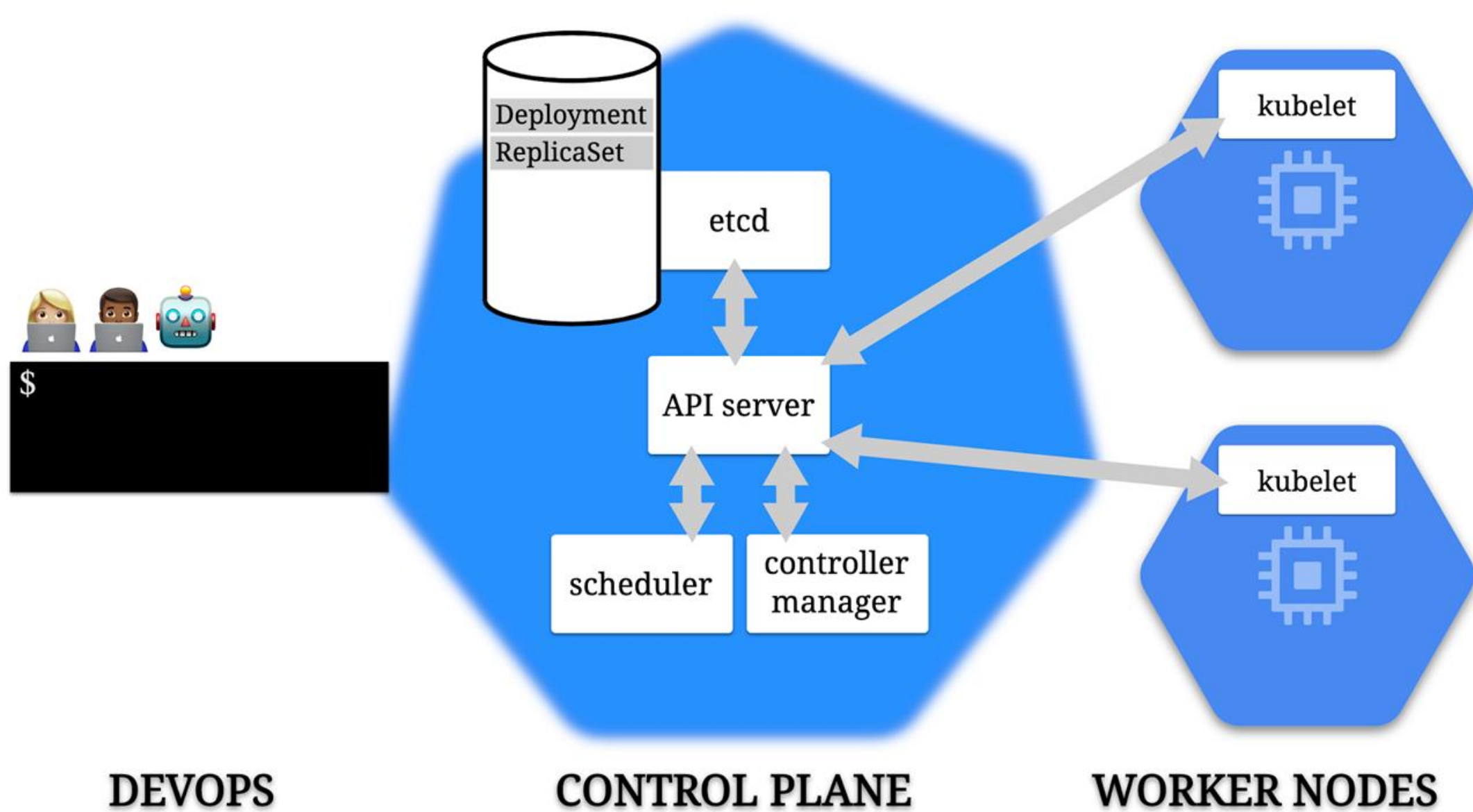
CONTROL PLANE

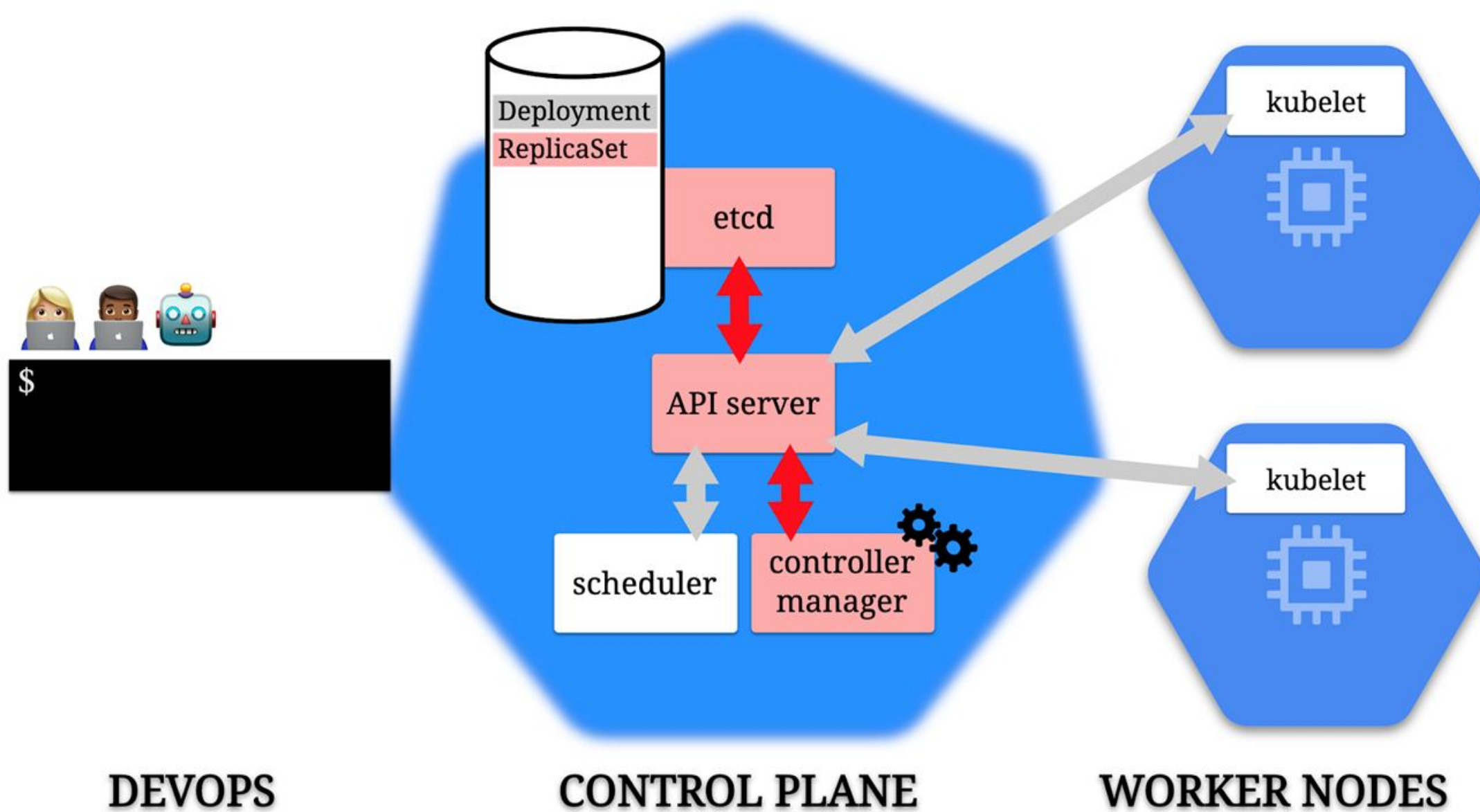
WORKER NODES

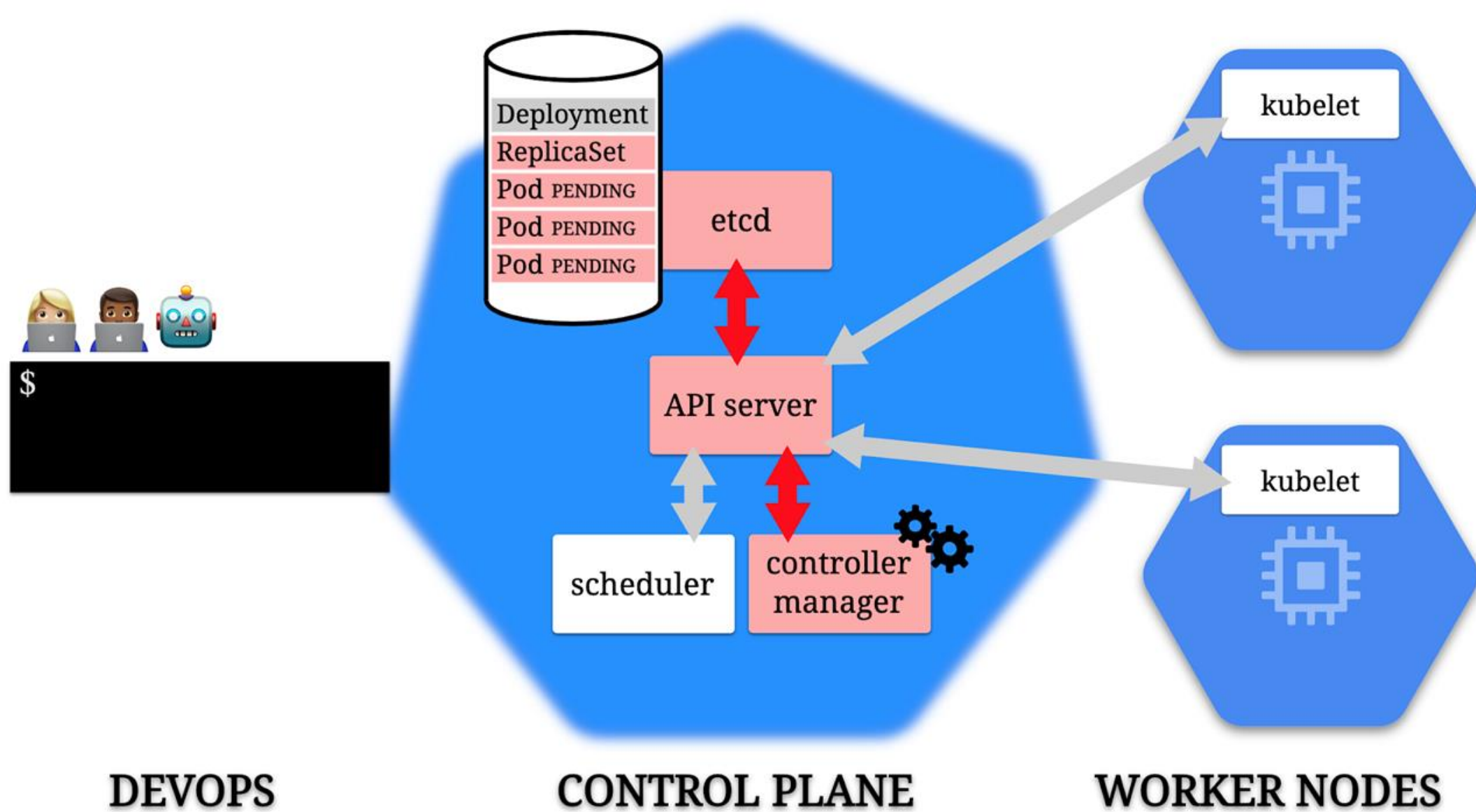


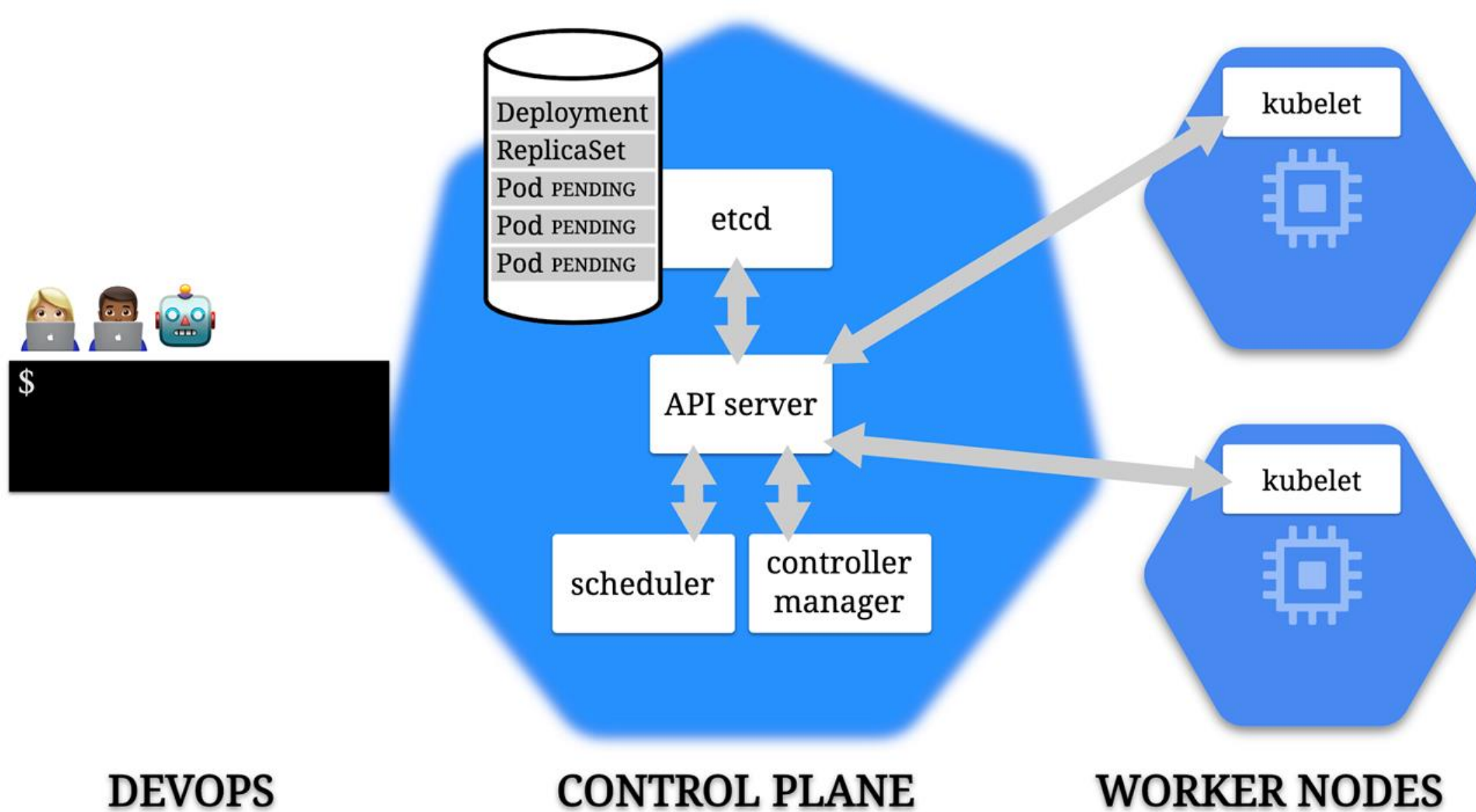


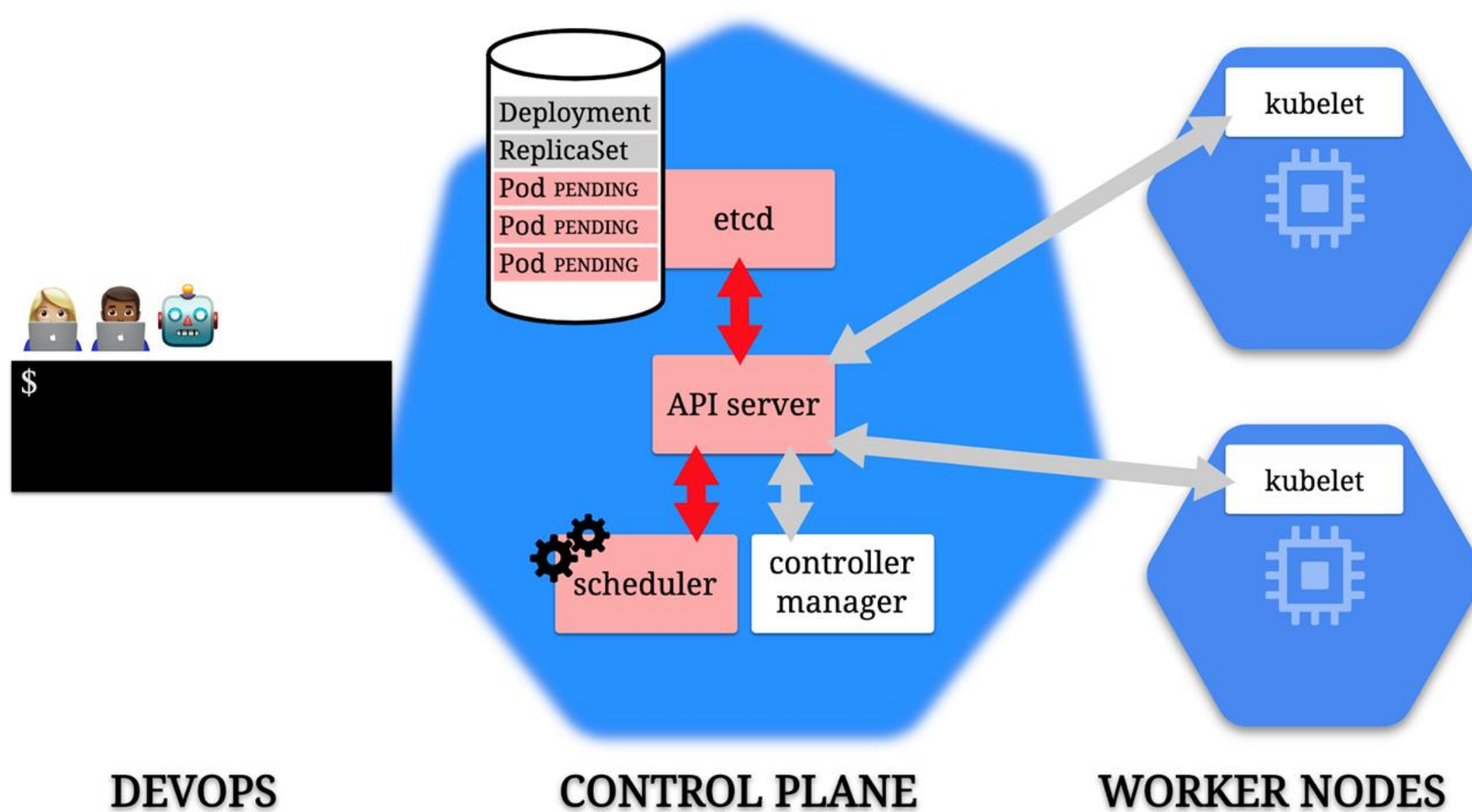


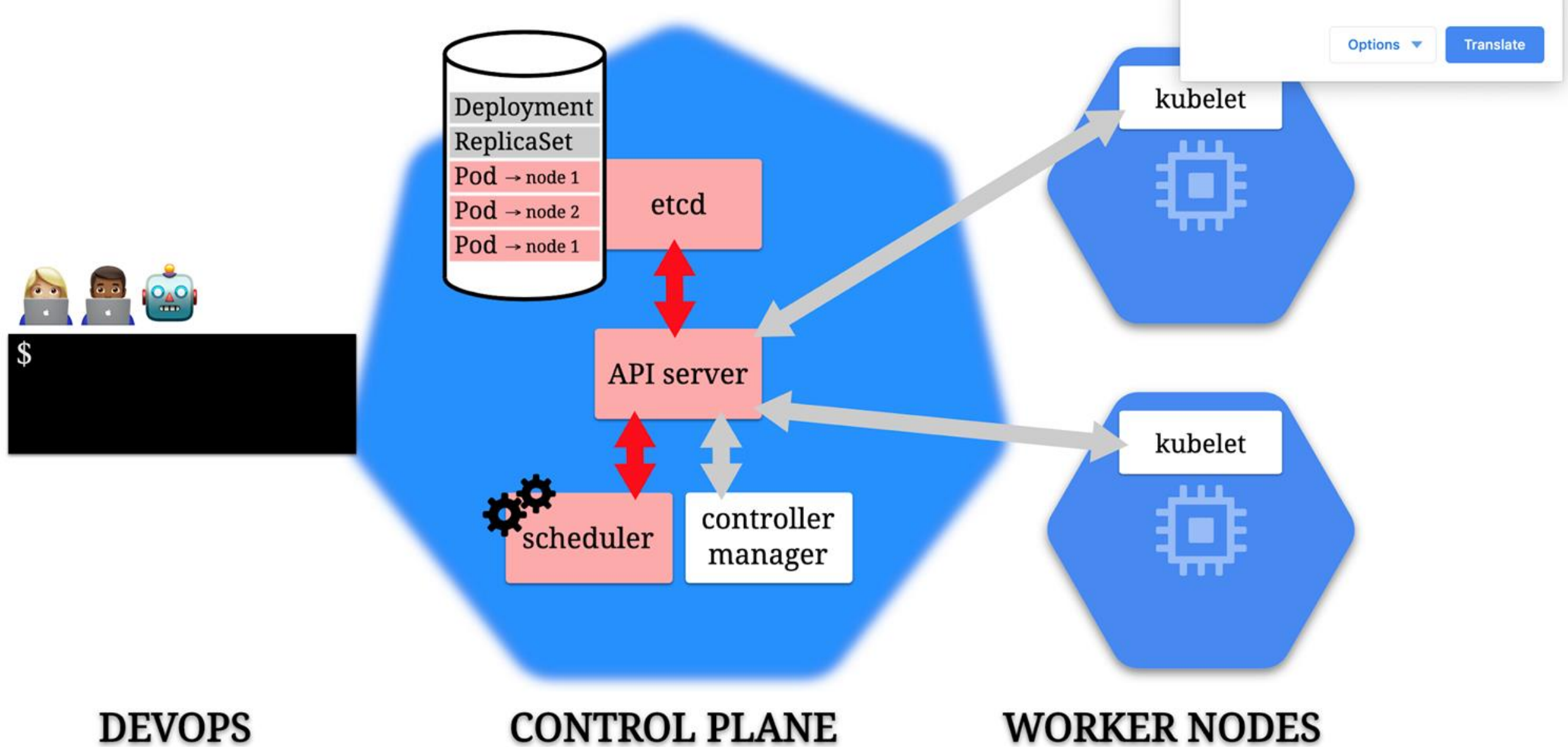


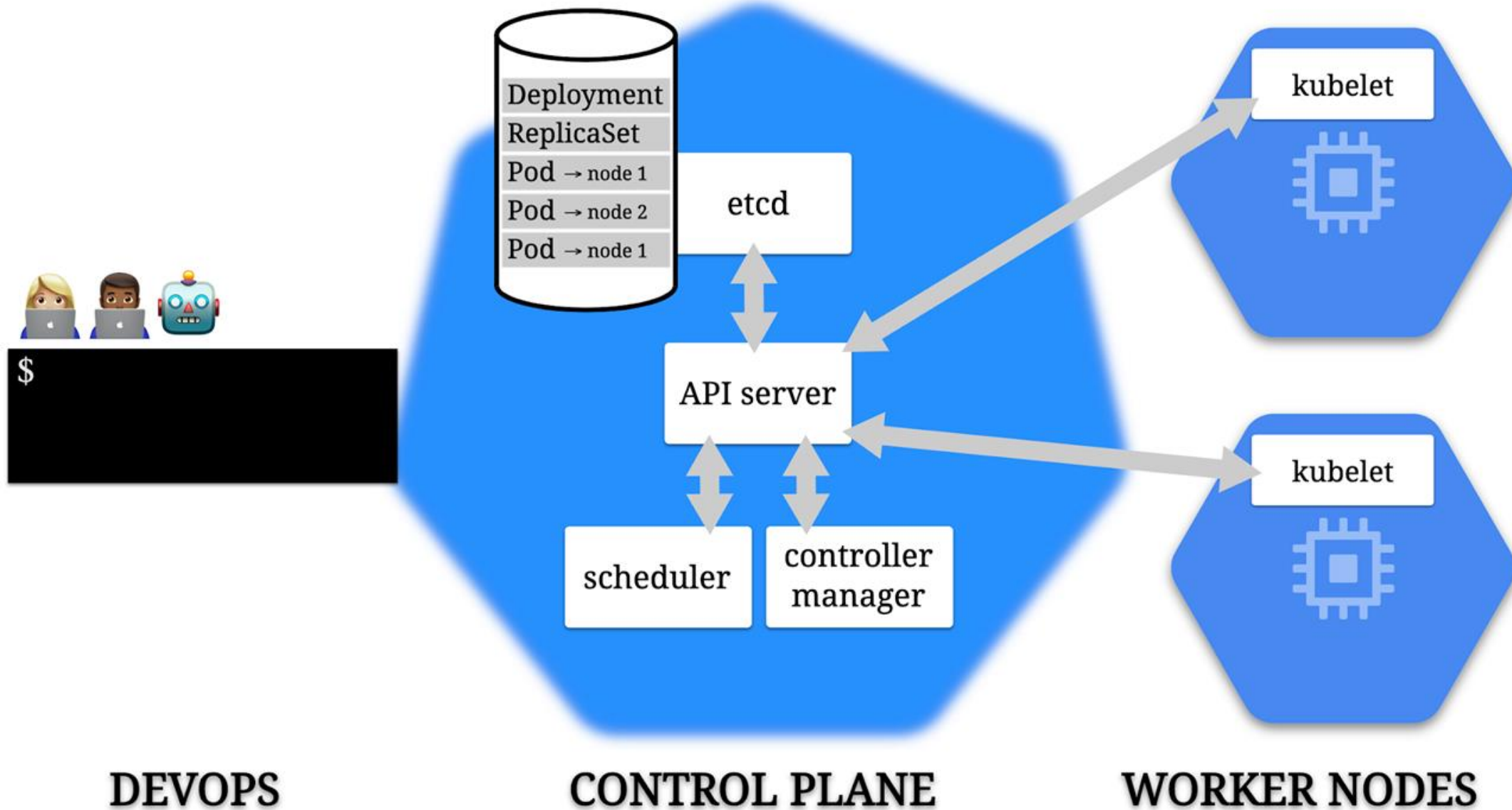






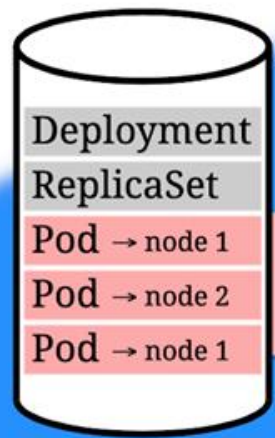








DEVOPS



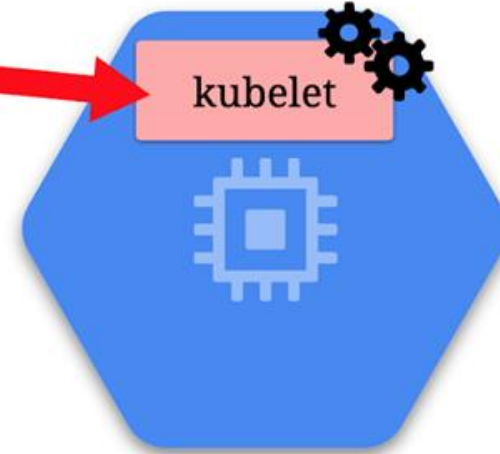
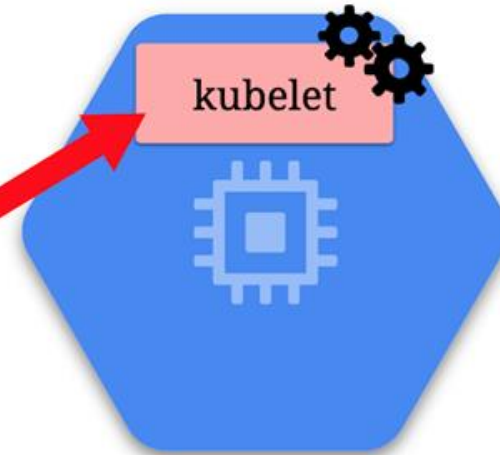
etcd

API server

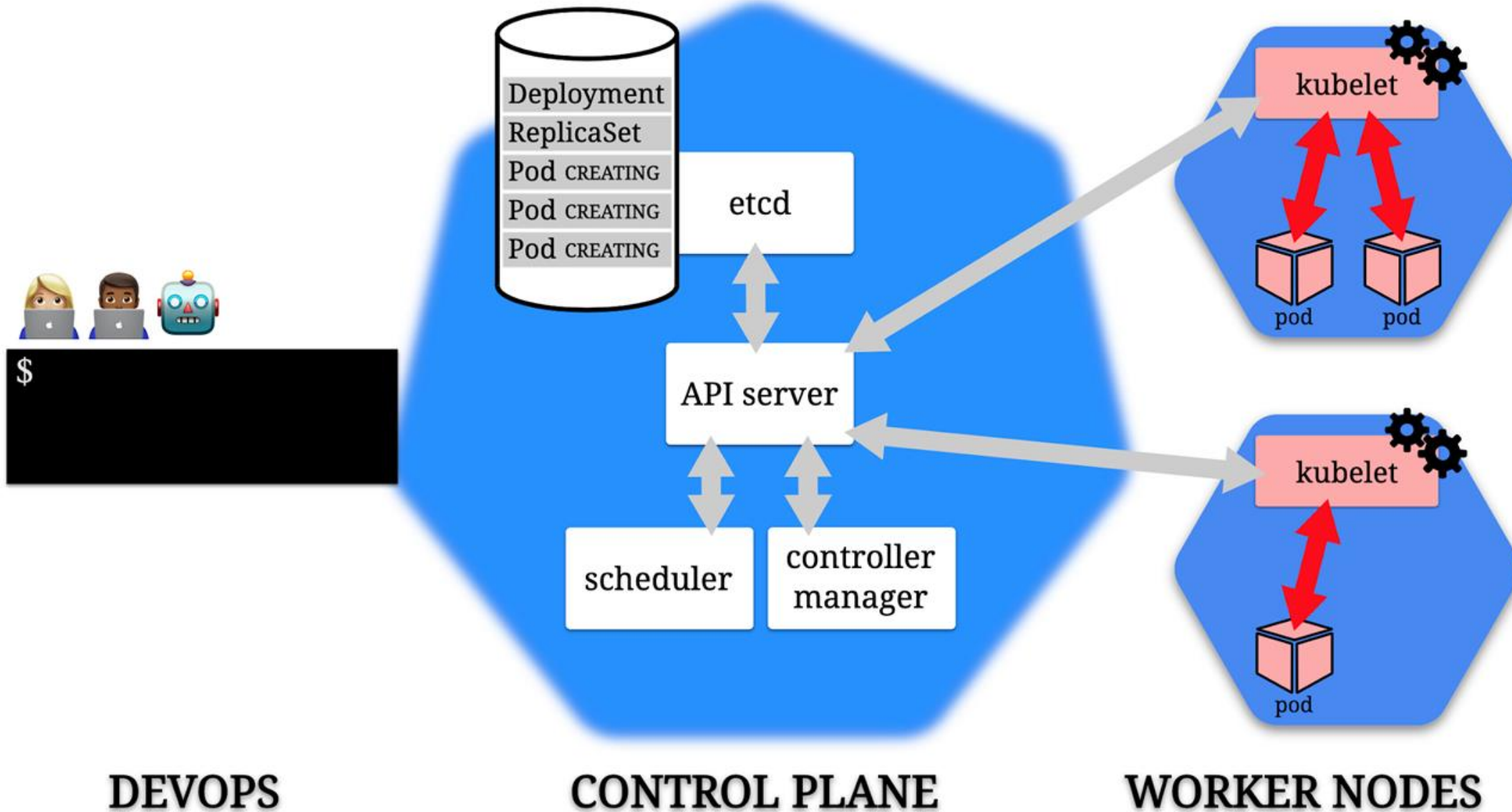
scheduler

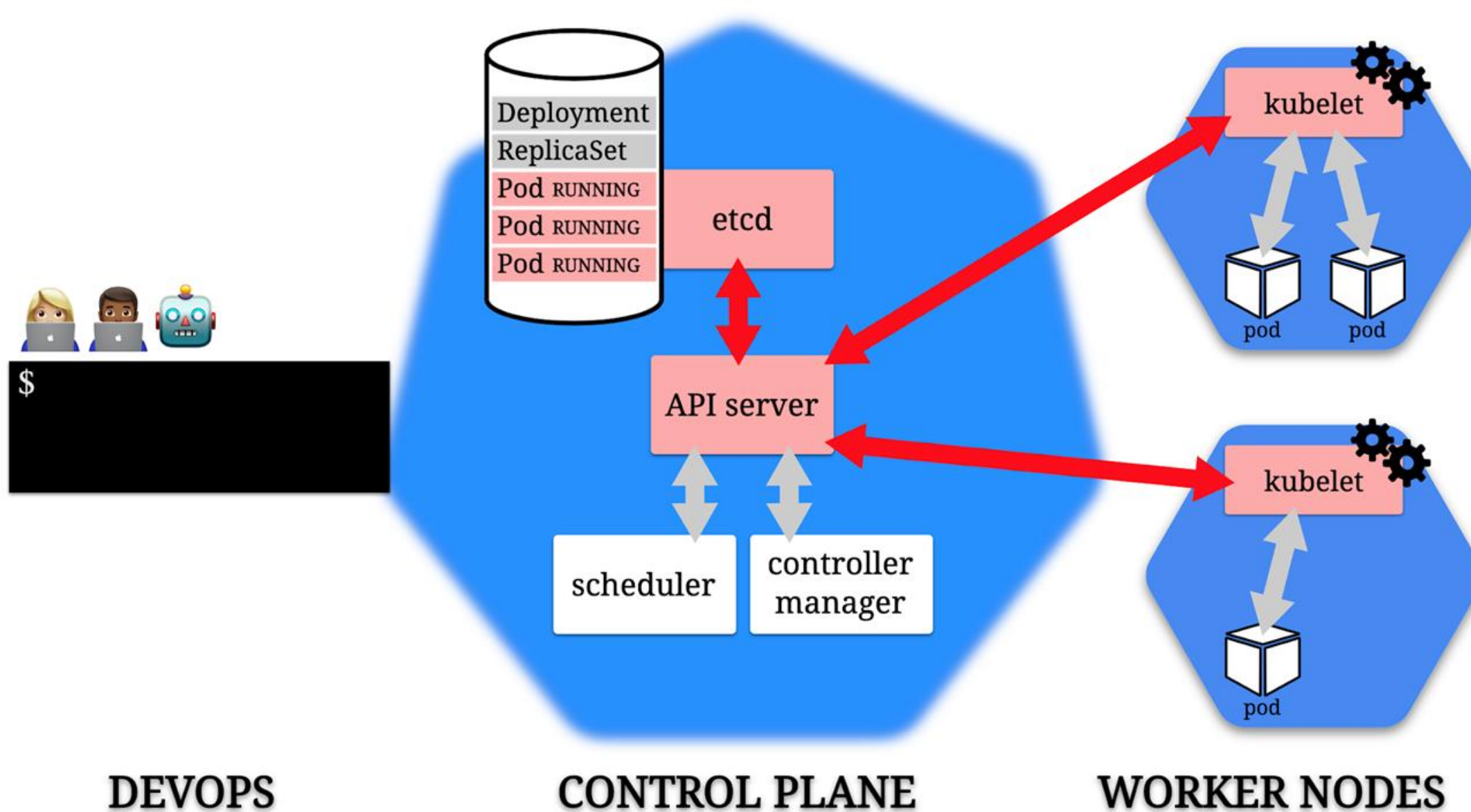
controller
manager

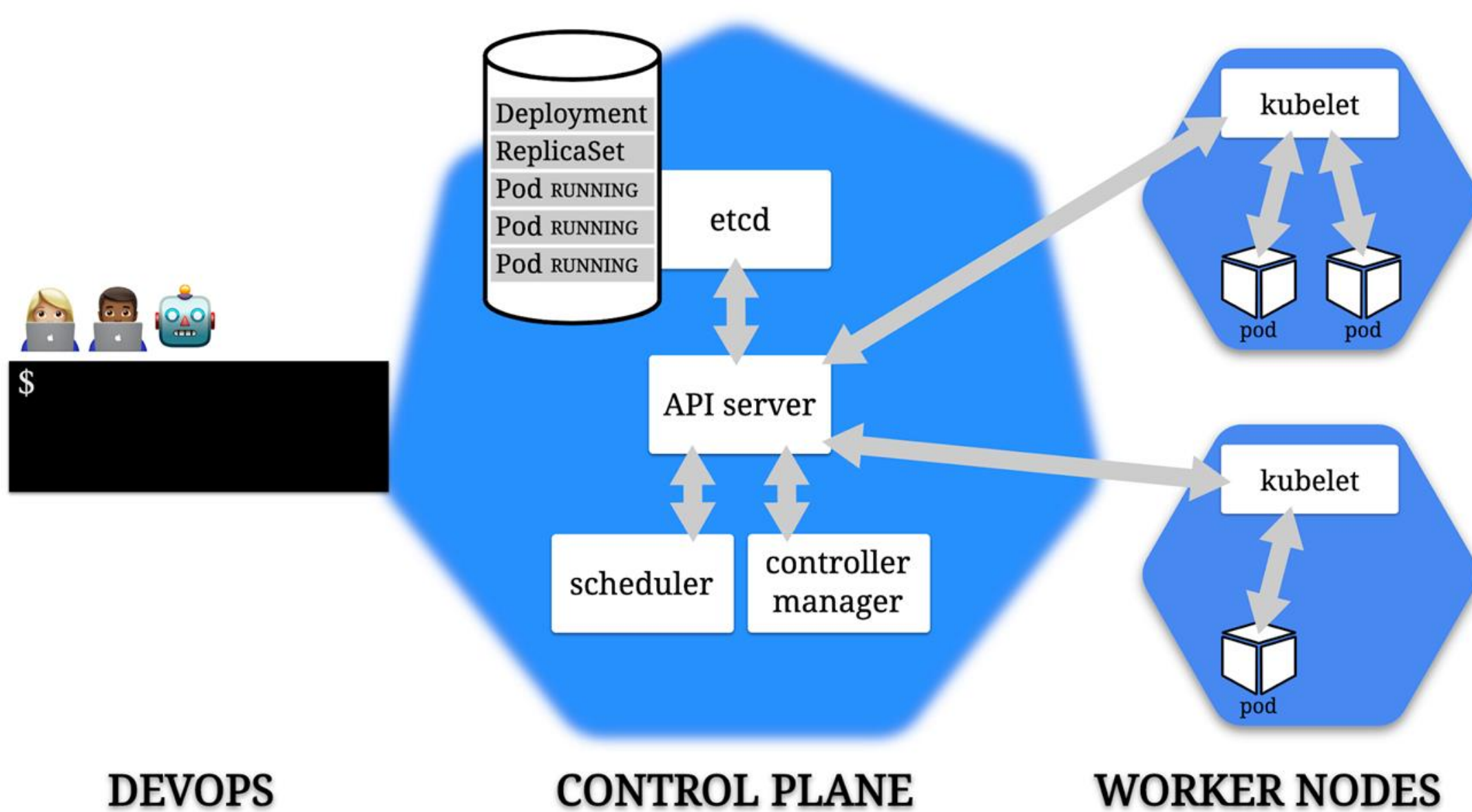
CONTROL PLANE



WORKER NODES









Control Plane: The API

- The fundamental fabric of Kubernetes
- RESTful API
- Enables all communication between k8s components, and all operations they can do
- Enables external user commands into the cluster



Master in more detail

- The Master is composed of multiple components
- **kube-apiserver** which is the front facing interface into the master exposes its interface via REST api and consumes json or yaml
- **Etcd** is the cluster store – an open source distributed key value database used to keep the state and config for the cluster
- We send a manifest file that defines our intents to the api server, the api server validates it, and saves it to the cluster store



- **kube-scheduler**
 - Component on the master that watches newly created pods that have no node assigned, and selects a node for them to run on
 - Factors taken into account for scheduling decisions include individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference and deadlines
- **kube-controller-manager** - Component on the master that runs controllers
 - Node Controller: Responsible for noticing and responding when nodes go down
 - Replication Controller: Responsible for maintaining the correct number of pods for every replication controller object in the system
 - Endpoints Controller: Populates the Endpoints object (that is, joins Services & Pods)
 - Service Account & Token Controllers: Create default accounts and API access tokens for new namespaces



Control Plane Components: kube-apiserver

- Exposes the API for the control plane
- It can scale as traffic increases



Control Plane Components: kube-controller-manager



- Controllers are what keep the desired state maintained
- The kube-controller-manager runs all controllers in the cluster



Control Plane Components: cloud-controller-manager



- Acts as liaison between the cluster and your cloud provider



Node Components: kubelet

- Makes sure the pods in the node stay running and healthy
- They operate off the “spec” in a pod’s manifest YAML file



Node Components: kube-proxy

- Manages network communication to pods
- Every pod has one as a network proxy



Node Components: Container Runtime

- The software that runs the containers in the pod
- Docker is most common
- Others are containerd and CRI-O
- They need to comply with a standard called [Kubernetes CRI](#)



Addons: Cluster DNS

- A DNS system that runs in a pod in your cluster
- It also spins up a k8s Service
- Allows the use of consistent DNS names instead of IP addresses



- The pods in a node are not always up and running, ready for incoming traffic
- We need a way to check
- The kubelet does this through periodic “liveness” checks into the pods in its node



- kubectl (“kube-cuttle”) is command-line tool used to interact with API
- May require installation (automatic if using k8s via Docker Desktop)
- Config file defines details of connection to cluster (e.g., ~/.kube/config)
- Run ``kubectl cluster-info`` to confirm connectivity
- Run ``kubectl`` from command-line (with arguments) to see options
- NOTE: k8s in Docker Desktop should handle most of this for you



- Two types of pod: single container and multiple container
 - Making a single container pod is simple – use kubectl run
- ```
$ kubectl run <name of pod> --image=<name of the image from registry>
```



- Usually, you will have one container per pod
- There are cases when multi-container pods make sense: When both pods have the same lifecycle, or they **MUST** run on the same node.
- An example: The primary container needs a helper process – it needs to be in the same node to be utilized.
- There are three main patterns for multi-container pods:
- Sidecar pattern
- Adapter pattern
- Ambassador pattern



# Multi-container Pod: Sidecar Pattern

- Main App plus a “helper” app that isn’t part of the main app
- Example: Main app is a web app; helper is a logging utility



# Multi-container Pod: Adapter Pattern

- Used when you need to standardize output from various apps in the cluster
- Each app type may format output in a unique way, but the cluster's monitoring needs to work with standardized formats
- Adding an adapter container to the pod for each app lets you still use the normal app output format in other areas, but have the cluster-level data work use a standardized format



# Multi-container Pod: Ambassador Pattern

- One more way to allow the pod's containers to communicate with the outside world
- Add an “ambassador” container to the pod that acts as a proxy for network traffic to and from the primary container
- This lets you control traffic patterns based on the specific use case of the pod in question.
- Example: database connection in dev/qa/prod environments

# Diving Deeper into Kubernetes







# Namespaces in k8s

- Provide a grouping mechanism in Kubernetes
- Every k8s object belongs to a namespace
- Every Cluster automatically includes a default namespace



# Namespaces in k8s

- Allow you create logical separation within a physical Cluster
- Provide a boundary for security and resource control
- Can explicitly create a namespace and deploy resources to it using namespace field in manifest or --namespace arg on kubectl



# Namespaces in k8s

- Objects within a namespace are isolated from others
- Enables creation of multiple instances of same apps with same names
- Resources from one namespace can communicate with another namespace using Services
- Controller only looks for matching resources in its namespace



- Can be used to define connection details for a k8s Cluster
- Sets the default namespace to be used
- Prevents need to specify namespace – will use context to automatically set if excluded (like with default in standard install)
- Able to switch between contexts using use-context command



- Can also be used to manage switching between different Clusters (local or remote)
- For remote, API server will be secured with TLS and kubectl uses a client cert for AuthN



# ConfigMaps vs. Environment Variables

- A ConfigMap is an API object that lets you store configuration for other objects to use
- These are key/value pairs
- This lets you decouple environment-specific config data from your container images, making your apps more portable
- CAUTION: ConfigMaps do not provide encryption or secrecy. Use a Kubernetes Secret for that.



- Can be:
  - Set of key-value pairs
  - Blurb of text
  - Binary files
- One pod can use many ConfigMaps and one ConfigMap can be used by many pods
- Data is read-only – pod can't alter



- Can be created from a literal:

```
`kubectl create configmap <name> --from-literal=<key>=<value>`
```

- Can be created from a file (to group together multiple settings):

```
`kubectl create configmap <name> --from-env-file=<file-path>`
```

Where <file-path> contains .env file like:

```
key1=value1
```

```
key2=value2
```

- Can be created from a .yaml definition file (like all things k8s)





- Can be presented as files inside directories in the container
- Uses volumes – making contents of ConfigMap available to pod
- Uses volume mounts – loads contents of ConfigMap volume into specific container path in pod
- Can contain settings to override defaults



# ConfigMaps - Precedence

- If env vars defined in multiple places, definitions in `env` section in pod spec will trump others (localized)
- Typical approach:
  - Default app settings “baked in” to container image (e.g., to support dev mode)
  - Specific settings for an environment stored in ConfigMap and surfaced to container filesystem
  - Merges with default settings (overwriting where applicable)
  - Final tweaks can be accomplished with env variables in pod spec



- Secrets let you store and manage sensitive information, such as passwords, OAuth tokens, and ssh keys.
- A pod needs to reference a secret. There are three ways get to the secret:
- As a file in a Volume that is mounted to a container;
- As an environment variable for a container;
- By the kubelet, when it pulls images for the pod
- NOTE: The default config for a Secret does NOT have encryption – data is plain text. You need to configure “[Encryption at Rest](#)” for the Secret”, and [Role Based Access Control](#) in your cluster



- Services are an abstraction
- They define a set of pods, and an access policy for them
- A pod has an IP address
- Pods are ephemeral – what happens to IP traffic if they die?
- Pods in a service can have fixed IP addresses that stay the same even if the actual pod dies and is spun up again
- Services are k8s objects; they are [created](#) like any other



# k8s Service Types

- ClusterIP (default): IP is internal to the cluster
- NodePort: IP for the port is exposed with a static IP address
- LoadBalancer: IP traffic to nodes is managed by external load balancer
- ExternalName: IP traffic is routed by DNS Name (foo.example.com)



# Cluster Access – Internal vs. External

- You can access the cluster API using kubectl (“proxy”)
- You can get libraries for popular languages that can access the cluster
- k8s objects (pods) can also access the API internally
- This uses an internal DNS
- “kubernetes.default.svc” maps to the API server
- The pod usually uses a “sidecar” container to do this



# Container File System

- Each container in a pod has its own filesystem built by k8s
- Can be built from multiple sources, including:
  - Layers from the container image
  - Writable layer for the container
- Can be expanded with other sources like ConfigMaps and Secrets – mounted to a specific path in a container
- Volumes can provide another source of storage – volumes are defined and mounted to containers



- Empty directory stored at pod level vs. container level
- Mounted as a volume into the container so visible there
- Any data stored there from the container remains in pod between restarts
- Replacement containers can access data from predecessors





- Data physically stored on node in cluster
- Extends beyond lifecycle of pod
- Available to replacement pods (but only if run on same node)
- Mounted as a volume into the container like others
- However, can have issues:
  - In cluster with multiple nodes, limits usefulness
  - If not careful, can expose large blocks of host data



- Like pods (abstraction over computer) and services (abstraction over network), persistent volumes provide abstraction over storage
- k8s object that defines available section of storage
- Can be “mapped” to shared storage (like NFS) or locally (for dev/test purposes)



# PersistentVolumeClaim

- Pods are not allowed to use persistent volumes directly
- Instead, pods claim using PersistentVolumeClaim object type in k8s
- Requests storage for a pod
- k8s handles matching up a claim to a persistent volume
- Provides virtual storage abstracted from actual storage



# Dynamic Provisioning

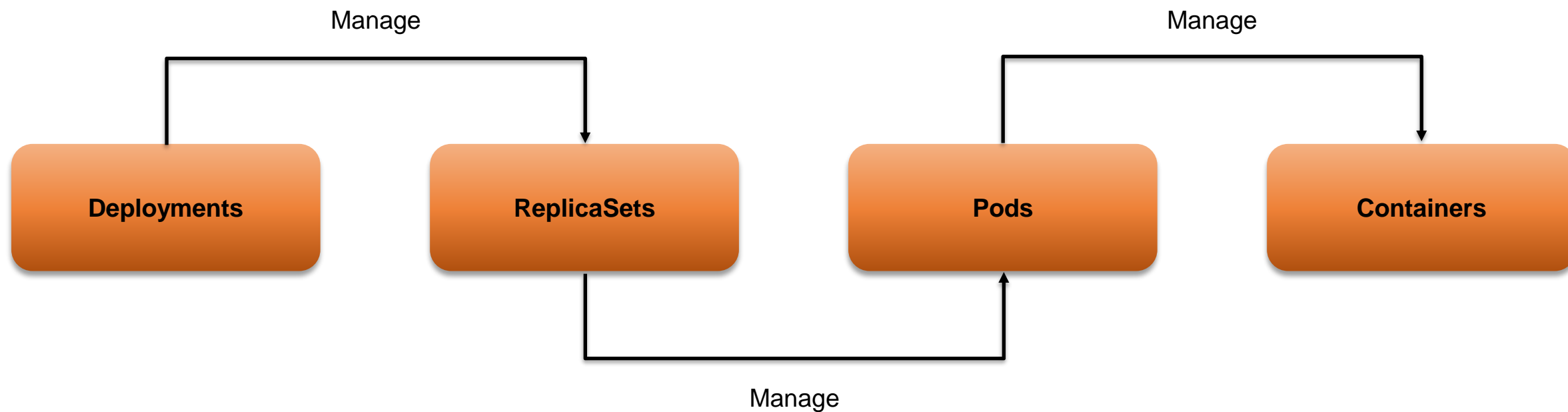
- Static provisioning accomplished by explicitly creating persistent volume and then claim (k8s binds claim to volume)
- Dynamic provisioning supported as well
- You create the persistent volume claim and let persistent volume be created on demand by cluster
- Can leverage storage classes for defining and matching types of storage



- Defined by three properties:
  - provisioner – component that creates persistent volumes on demand
  - reclaimPolicy – what to do with dynamically created volumes when claim is deleted
  - volumeBindingMode – eager vs. lazy binding/creation (at same time as claim creation or only when pod using the claim get created)



- Able to create custom storage classes
- Defined by three properties:
  - provisioner – component that creates persistent volumes on demand
  - reclaimPolicy – what to do with dynamically created volumes when claim is deleted
  - volumeBindingMode – eager vs. lazy binding/creation (at same time as claim creation or only when pod using the claim get created)
- Remember – persistent data can limit statelessness; be aware and use wisely





- Can be created directly as an object in k8s
- However, better practice would be to create/manage through deployments
- `replicas` property in pod spec in deployment can be used to build out multiple replicas for a pod config
- If not included, defaults to 1





# Setting Limits in Deployments

- When defining pod spec in deployment, can use limits to manage
- `resources` attribute defines compute resource requirements for pod
- `limits` can be used to apply limits (e.g., in amount of memory and cpu)
- Usually better to optimize once running vs. prematurely optimizing
- Target environment when running will help determine correct settings



- daemon is usually a system process that runs constantly as single instance
- In k8s, DaemonSet runs single replica of a specific pod on every node in cluster
- Usually used for infrastructure-level concerns:
  - Logging/central data collection
  - High availability for a reverse proxy running in cluster
- Looks similar to other controllers (e.g., Deployment) but no replica count



# Using DaemonSets in Deployments

- Control loop watches for nodes joining cluster
- New pod instance for the daemon will be started on new node
- Works to stop/remove daemon pod(s) if node(s) taken out of cluster



# Using DaemonSets in Deployments

- Can also target just a subset of nodes for daemon pod
- As with most other things in k8s, use label matching
- Watches for nodes join cluster and for metadata matching label
- DaemonSet control loop is different from ReplicaSets because has to monitor combo of node activity and pod count



- Represents a Pod controller that enables running apps at scale in a predictable manner
- Deployments/ReplicaSets create Pods with random names
- Also, starts Pods in parallel
- Some use cases (e.g., clustering) require a more deterministic approach



# StatefulSets

- For example, creating a cluster of PostgreSQL instances
- With StatefulSets, can create Pods with predictable names
- Can access the Pods individually over DNS
- k8s will start the Pods in sequence



# StatefulSets

- With a PostgreSQL cluster, able to earmark a Pod as primary
- Other Pods will wait for the primary and can use primary for replication
- Allows clustering of multiple database instances with read-replicas
- Pod name will be StatefulSet name followed by an index (starts at 0)



# StatefulSets

- Provides stability for the deployment
- Replacement of missing Pods will respect this ordering
- If Pod 0 goes down, a replacement Pod 0 will be created (with same config)
- Other Pod indexes will stay in place





- To get access to DNS names for individual Pods, create a “headless” service
- Create the service with a clusterIP of None
- Does not use fixed IP for the Service
- DNS entry for the service returns an IP address for each Pod in the StatefulSet
- Each Pod gets its own DNS entry as well



# StatefulSets

- Can require adjustment to data storage approach
- With something like PostgreSQL, each database instance will write data in its own writable layer instead of a shared location
- Cannot simply mount a standard PVC for data storage as every Pod would try to write to that volume shared across all Pods



- StatefulSet specification provides a volumeClaimTemplates field
- Can include storage class, capacity, and access mode definitions
- Each Pod in the spec gets its own dynamically-created PVC
- As previously discussed, creates PersistentVolumes for each using defined storage class (or default if not specified)
- This link between Pod and PVC remains in place even across Pod replacement



- Keep in mind – StatefulSets are intended to provide a stable and predictable deployment environment
- Less flexible than other controller types
- Can make changes more impactful
- Be sure app requirements fit model



- Provide options for running maintenance tasks in your k8s Cluster
- For example, with PostgreSQL cluster from previous, can schedule tasks against the DB “out-of-band”
- Tasks like backing up databases, data migration, data loads, etc.
- Defined with a Pod spec and run to completion as a batch process



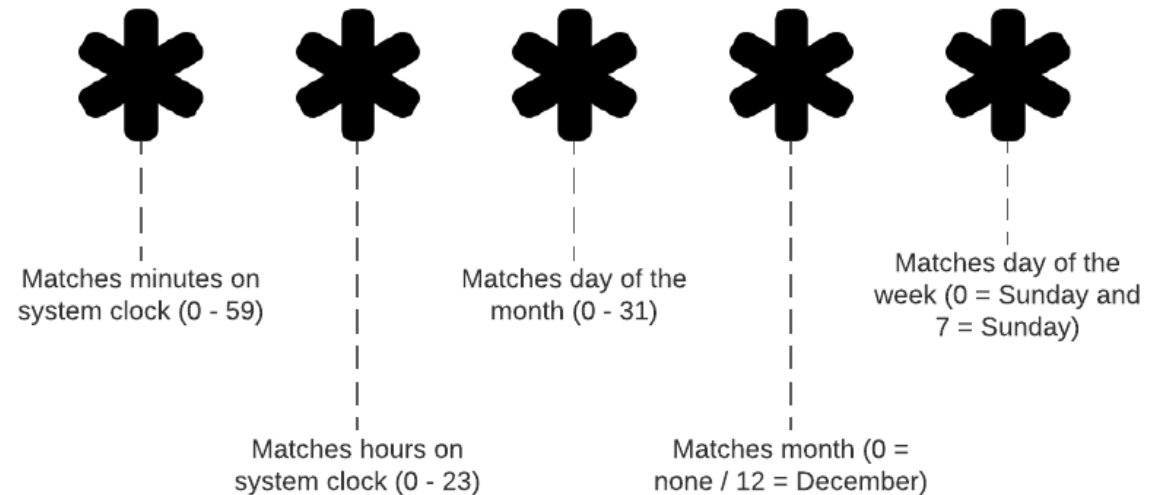
- Can run Pods for any container image but should represent a process with a discrete end (intended to run to completion)
- Jobs add labels to the underlying Pods created
- Includes a completions property for specifying how many times the Job should run



- And a parallelism property for specifying how many Pods to run in parallel
- Enables distribution of Job processing for multiple types/instances across the Cluster
- Also, can assist with managing the speed of Job execution



- Specific type of Job for executing regularly and on a schedule
- Allows definition of a Job spec for use by CronJob for Job creation on the defined schedule
- Use the Linux Cron format







- Can move CronJobs to suspended state
- Logic needs to account for standard Pod lifecycle management
- Could include restarts or parallel runs – logic needs to support



# Startup Probes

- First probe executed on startup and determines if app is initialized and ready for traffic
- Only runs on initialization – once complete, will not run again in the Pod
- Can specify number of probe attempts (failureThreshold)
- And number of seconds to wait between retries (periodSeconds)
- If all defined attempts fail, k8s will kill the Pod and startup a replacement



# Readiness Probes

- k8s knows if Pod container is running but not if application inside is healthy (specific to the app)
- Defined in the Pod spec
- Execute health checks on a fixed schedule



# Readiness Probes

- Act at the network level and manage routing for listening components
- If deemed unhealthy, Pod will be taken out of ready state and removed from list of active Pods
- k8s keeps running probe looking for Pods that have recovered for reassociation to the Service
- Different types of probes are supported – e.g., HTTP GET for web apps



# Liveness Probes

- Use same health checks as readiness probes
- Action on failure is different
- Act at compute level and restart unhealthy Pods
- Different types of probes are supported – e.g., HTTP GET for web apps



- k8s collects logs from container runtime and stores as files on node
- Goal is to get logs out of the container and potentially centralize for aggregated processing
- Often uses a sidecar container for the aggregation



- Logs can benefit from normalization
- Parsing can be applied as the information passes through the pipeline
- Several 3<sup>rd</sup> party tools (e.g., Fluentd for log collection, Elasticsearch for log storage, and Kibana for view)



# Deploying and Scaling Clusters

- Clusters can scale as needed
  - Increased load
  - Rolling cluster updates
- This is often managed by the cloud provider with cloud native





# The Ingress Controller

- Specialized load balancer for k8s environments
- Manages traffic to pods in your cluster
- Like Nodes, Services etc., they are deployed using the cluster API
- There are [several](#) available besides the k8s offering



# Rolling Deployments

- Code changes, so your pod (containers) change
- We want a stable experience for users
- k8s does this via rolling deployments
- As updates nodes become available, traffic is load balanced to ensure uptime



# Blue/Green Deployments

- How they work: keep two identical environments, conventionally called blue and green, to do continuous, risk-free updates. This way, users access one while the other receives updates.
- With bare metal servers, this is expensive – one of the environments isn't used much when the other is
- k8s makes this easy – we can make a second environment, switch traffic over, then kill the first with little impact



# Canary Pods for Testing and Production

- A strategy to try out a new feature only for a small segment of users or traffic (A/B testing)
- k8s makes this easy
- Use a tag to designate what pods will receive a small percentage of traffic
- Also, allows smoke testing on small scale before “opening floodgates”



# NetworkPolicy and SecurityContext

- NetworkPolicy can be used to manage access within a Cluster
- Work like firewall rules and use label selectors for blocking traffic
- Provides ingress rules and egress rules to manage flow
- Rely on network implementation in Cluster for enforcement



# NetworkPolicy and SecurityContext

- SecurityContext provides an option for container security
- Alternative to containers running as root or super user account
- Offers more fine-grained control than at Pod level, enabling explicit addition and removal of Linux kernel options
- Must remain aware of potential breaks in apps caused by security config

# Review





# Kubernetes Fundamentals

- Why would we need container orchestration?
- How does k8s work?
- What to we use environment variables for?
- How do we handle sensitive info in k8s?





- What are k8s Services? What do we use them for?
- How do we scale Pods?
- What is the Ingress Controller? Why do we have alternatives to it?
- What are some deployment strategies for k8s? What should we think about when choosing one?
- What are “canary pods” and when can we use them?



# Future Study



# THANK YOU

