Welcome

Working with GoLang Microservices

Prerequisites

This course assumes you:

- Have at least an intermediate level of experience with Go
- That experience could be gathered through any combination of previous courses, personal experience, or both

Why study this subject?

- Go is cool! ☺
- Microservices are cool! ☺
- Microservices in Go are REALLY cool! ☺ ☺

We teach over 400 technology topics

You experience our impact on a daily basis!

My pledge to you

I will...

- Make this interactive
- Ask you questions
- Ensure everyone can speak
- Use an on-screen timer

Objectives

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

- Construct & deploy microservices utilizing Go
- Describe the fundamental philosophy around microservices architecture, their benefits, and their cost
- Speak to Serverless as well as Containerization deployment models

Agenda

- Microservices – What they are and why do we use them?
- GoLang Deep Dive – Go routines, channels, and cross-compilation
- Serverless Architectures as an alternative
- Deploying microservices using Terraform
- Containerization and container orchestration for Go microservices

How we’re going to work together

- Slides / lecture
- Demos
- Team discussions
- Labs

Microservices vs. Monolith

Microservices

- One of the current trends in software development (though it’s not really new)
- Has its foundations in SOA (Service Oriented Architecture)
- Requires an approach that favors decomposition

Microservices – Key Characteristics

- “Small” units of service functionality accessible remotely
- Independently-deployable (KEY)
- By extension, independently-scalable
- Modeled around business domain (instead of tech domain)

Microservices – Key Characteristics

- Interact over networks (including the Internet)
- Technology can be flexible
- Encapsulate both business capability AND data
- Data sources are not directly shared but exposed through well-defined interfaces
- About cohesion of business functionality vs. technology

vs. Monolithic

- Monolithic applications can exhibit some of the same characteristics as microservices:
  - Distributed
  - Interact via network
  - Might use unshared data sources
- Key difference, though, is requirement to deploy all parts of the system together (not independently-deployable)

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

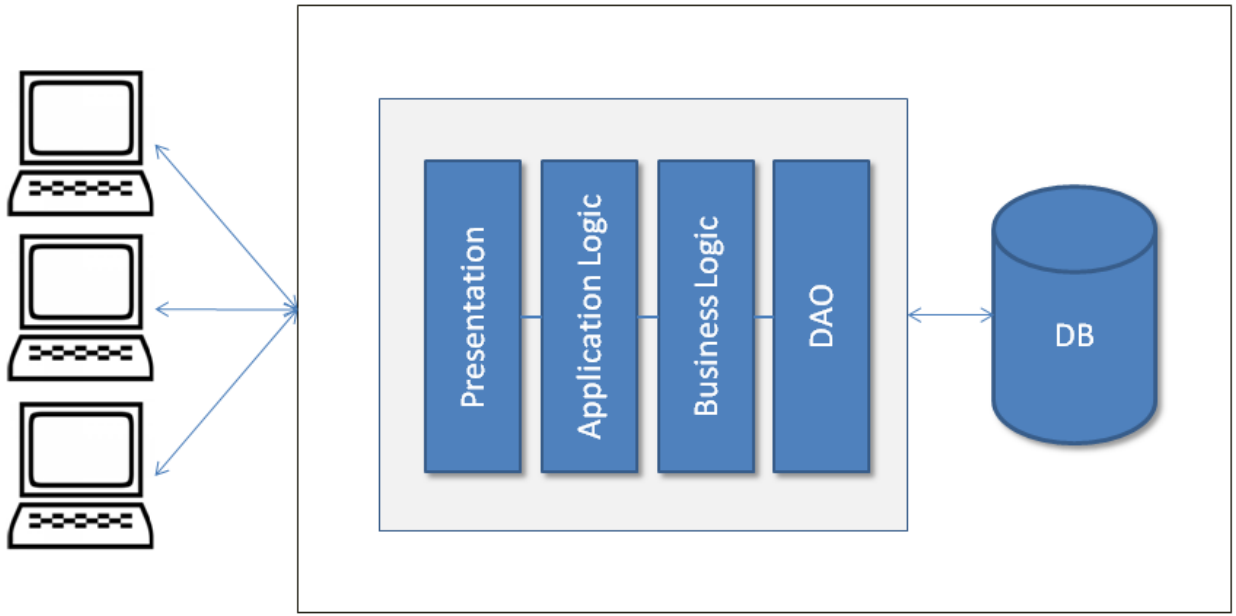
- With microservices, key concepts:
  - Cohesion (goal to increase)
  - Coupling (goal to decrease)

- SOLID principles and Domain Driven Design (DDD) can help with designing a microservices architecture:
  - 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

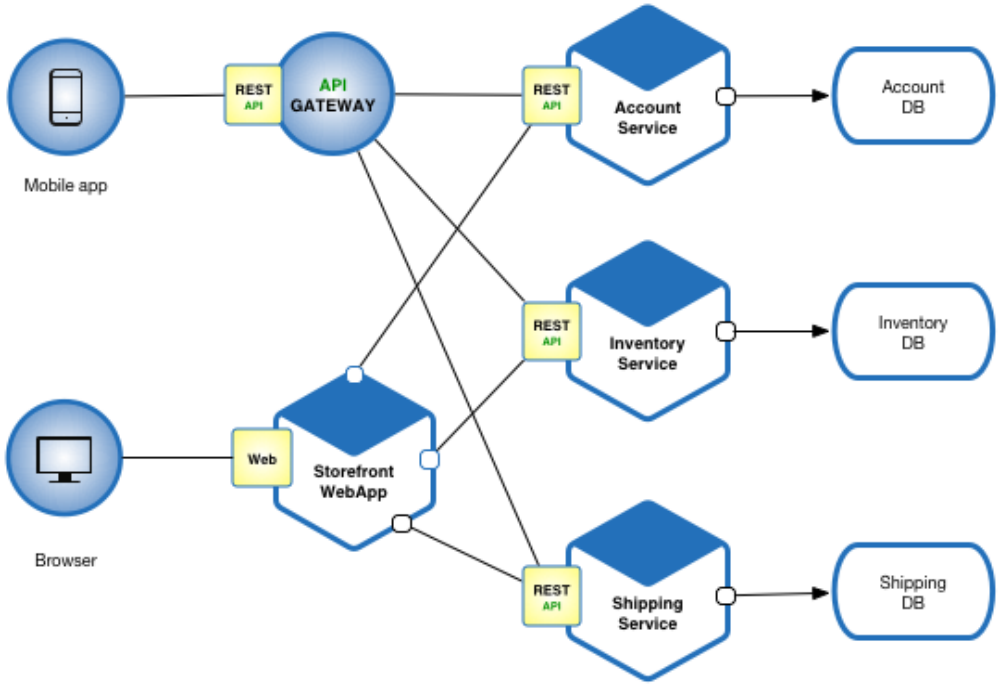
Microservices & Containers

- 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

Monolithic Architecture Example



Example Microservices Architecture



Lab

Lab Discussion

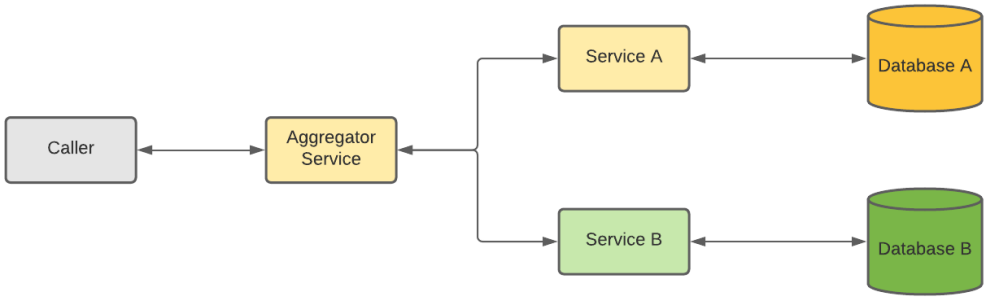
Microservice Design Patterns

Microservice Design Patterns

- Aggregator
- API Gateway
- Chain of Responsibility
- Asynchronous Messaging
- Circuit Breaker
- Anti-Corruption Layer
- Strangler Application
- Others as well (https://microservices.io/patterns/index.html)

Microservice Design Patterns - Aggregator

- Akin to a web page invoking multiple microservices and displaying results of all on single page
- In the pattern, one microservice manages calls to others and aggregates results for return to caller
- Can include update as well as retrieval ops



Microservice Design Patterns - Aggregator

Benefits:

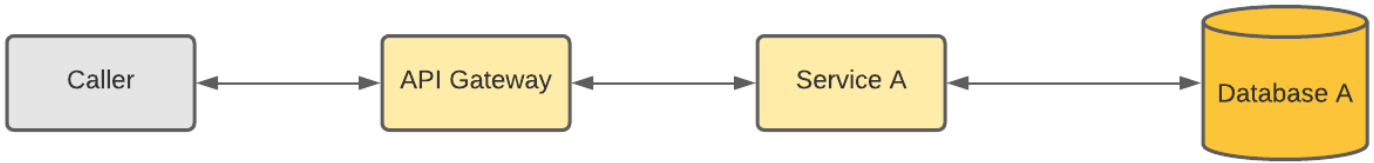
- Helps you practice DRY (Don’t Repeat Yourself)
- Supports logic at time of aggregation for additional processing

Potential “gotchas”:

- Performance if downstream microservices not called asynchronously

Microservice Design Patterns – API Gateway

- Specific type of infrastructure that helps manage the boundary
- Provides an intermediary for routing calls to a downstream microservice
- Provides a protection and translation layer (if calling protocol different from microservice)
- Can be combined with other patterns as well



Microservice Design Patterns – API Gateway

Benefits:

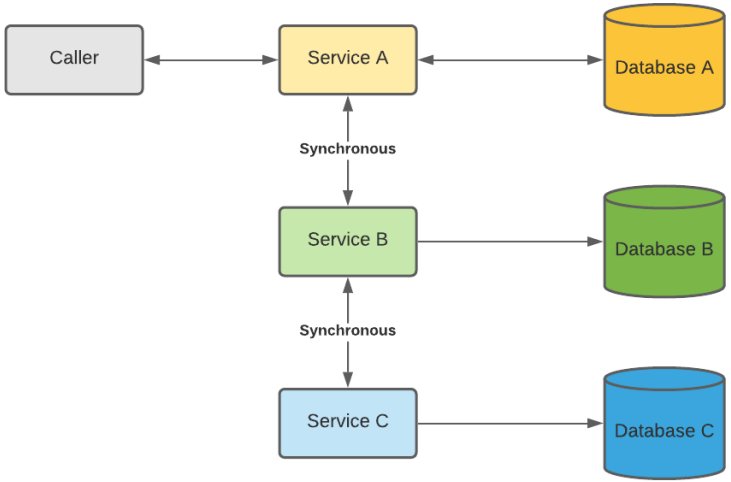
- Usually, built in throttling (to protect against DDoS)
- Can support translation between caller and downstream microservice for protocol, message format, etc.
- Provides a centralized layer for AuthN/AuthZ
- Can be combined with load balancing to enable scalability and resiliency

Potential “gotchas”:

- Increased complexity – another component and point-of-failure
- Increased response time due to additional network “hop”

Microservice Design Patterns – Chain of Responsibility

- Represents a chained set of microservice calls to complete a workflow action
- Output of 1 microservice is input to the next
- Uses synchronous calls for routing through chain



Microservice Design Patterns – Chain of Responsibility

Benefits:

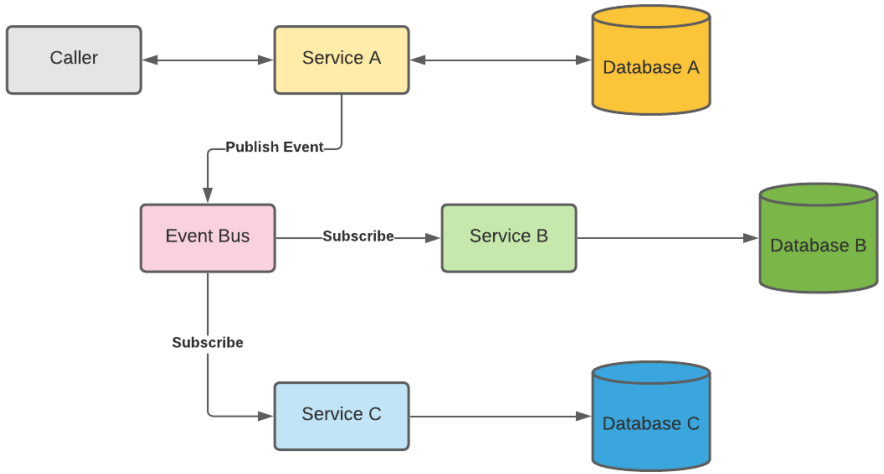
- Supports composition of microservices in a workflow that must happen in sequence
- Synchronous communication typically less complex

Potential “gotchas”:

- Increased response time – response time becomes the sum of each microservice’s response time in the chain

Microservice Design Patterns – Asynchronous Messaging

- Supports asynchronous interaction between independent microservices
- Messages published to a topic by a producer
- Topic subscribed to by one or more consumers



Microservice Design Patterns – Asynchronous Messaging

Benefits:

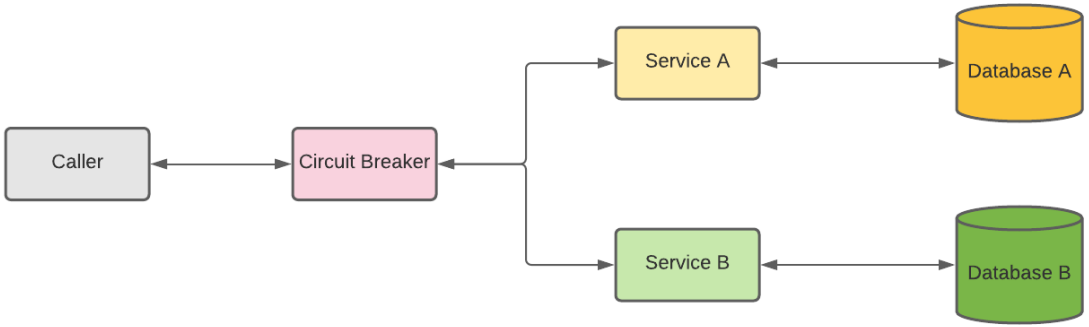
- Promotes looser coupling in cases where synchronous calls are not required
- Allows multiple services the option of being notified (vs. point-to-point)
- If given consumer is down, producer can continue to send messages and they won't be lost (Kafka)

Potential "gotchas":

- Additional complexity
- Can be harder to trace an action end-to-end (but there are ways to handle)
- Not a good fit if specific timing and sequencing required

Microservice Design Patterns – Circuit Breaker

- Prevents unnecessary calls to microservices if down
- Circuit breaker monitors failures of downstream microservices
- If number of failures crosses a threshold, circuit breaker prevents any new calls temporarily
- After defined time period, sends a smaller set of requests and ramps back up if successful



Microservice Design Patterns – Circuit Breaker

Benefits:

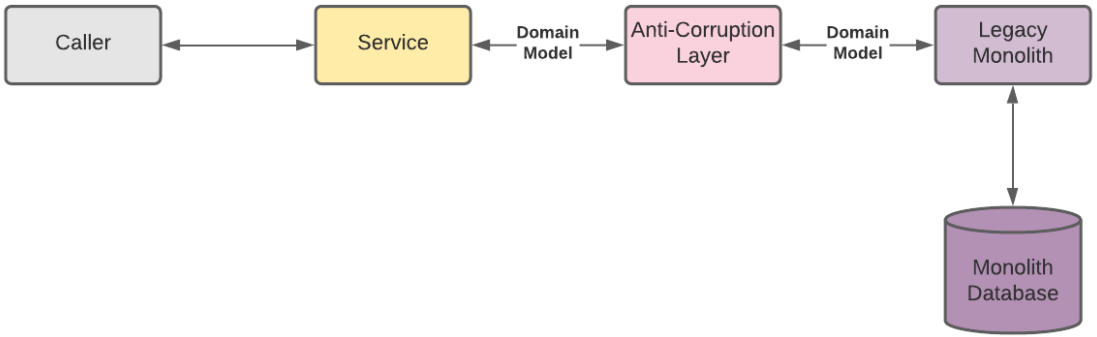
- Helps to optimize network traffic by preventing calls that are going to fail
- Can help prevent “noise” and network overload

Potential “gotchas”:

- Process supported needs to be able to absorb limited downtime
- If not managed correctly, can result in poor user experience

Microservice Design Patterns – Anti-Corruption Layer

- Prevents corruption of new microservice domain model(s) with legacy monolith domain model(s)
- Provides a proxy/translation layer to help keep models “clean”



Microservice Design Patterns – Anti-Corruption Layer

Benefits:

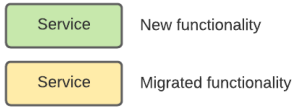
- Prevents crossing of boundaries and “leaking” of sub-optimal logic
- Helps to keep this insulation transparent between communicating parties

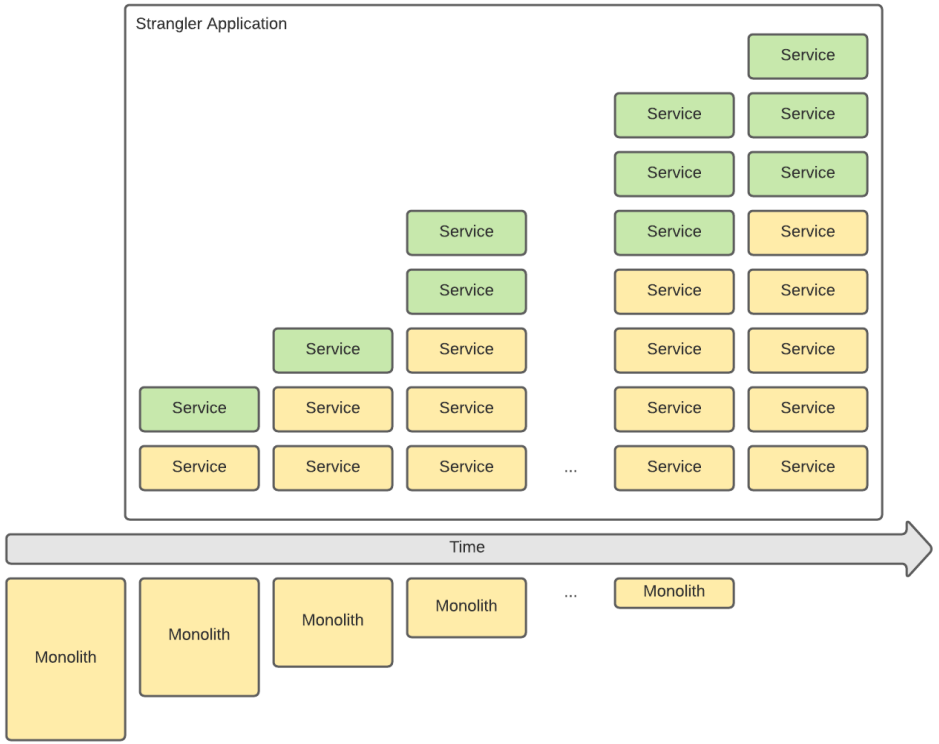
Potential “gotchas”:

- Additional complexity
- Additional testing required to validate the additional complexity

Microservice Design Patterns – Strangler Application

- Supports breaking up a monolith into microservices over time
- Can accommodate existing as well as new functionality





Microservice Design Patterns – Strangler Application

Benefits:

- Allows a gradual vs. “big bang” breakup
- Especially well-suited for large monoliths that are taking active traffic

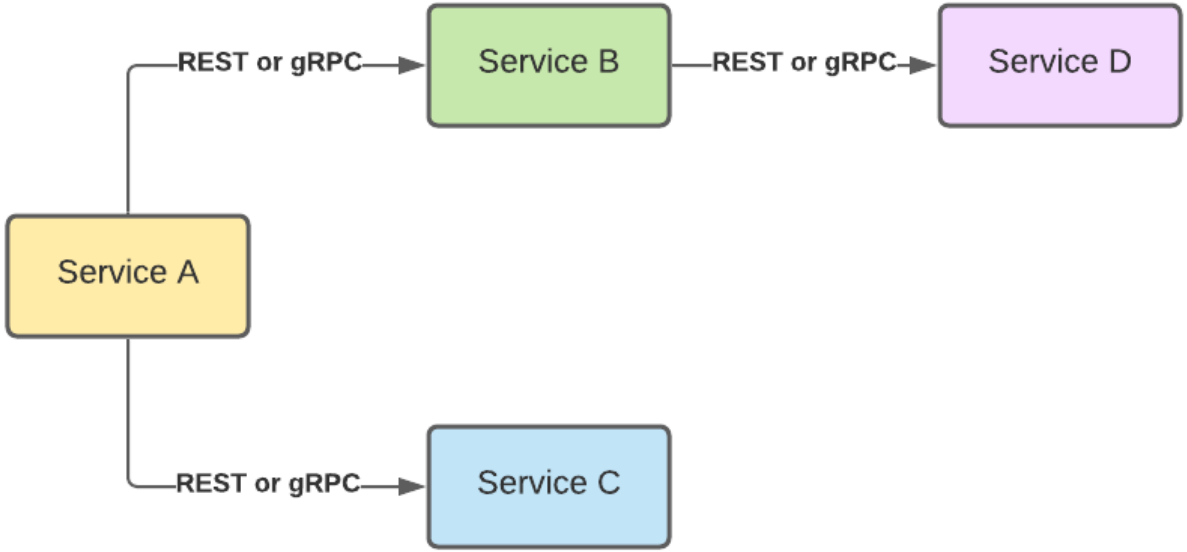
Potential “gotchas”:

- Requires business to be able to absorb gradual
- Adds complexity – coordinating new functionality, migrated functionality, and proper integration between them
- While transition in progress, will have to maintain two paths

Lab

Event-Driven Architectures

Microservices Integration Architecture – Option 1



What are some benefits of an architecture like this? What are some challenges?

Microservices Integration Architecture – Option 1

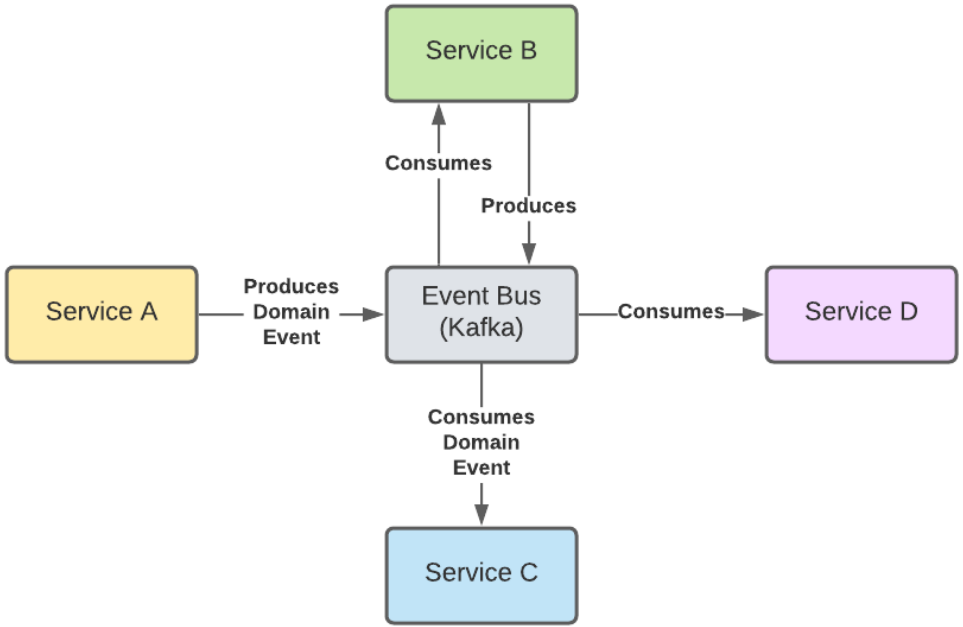
Benefits:

- Easier to trace a workflow end-to-end
- Of available options, usually less complex (relatively speaking)
- Supports some level of sequential timing through a workflow

Potential “gotchas”:

- Can promote point-to-point coupling
- If one of the target services is down, the workflow is down

Microservices Integration Architecture – Option 2



What are some benefits of an architecture like this? What are some challenges?

Microservices Integration Architecture – Option 2

Benefits:

- Looser coupling between services
- Flexibility in how a workflow gets processed
- If a consuming service is down, the event bus will hold on to the events
- When consumer recovers, can pick up where they left off

Potential “gotchas”:

- Observability/traceability through a workflow can be harder
- Adds complexity to the technical landscape
- Workload needs to be able to tolerate latency

Concurrency in Go

See https://github.com/KernelGamut32/golang-live

Concurrency Patterns in Go

Concurrency Patterns in Go

- Lexical confinement
  - Use lexical scope to manage access to data and concurrency primitives
  - Limit how multiple concurrent processes can interact with the data
  - Helps ensure correct approach to concurrency (but likely limited to simplest use cases)
  - For example, using directionality on channels to only expose read or write access in a controlled manner

Concurrency Patterns in Go

- Preventing goroutine leaks
  - Go is very efficient at multiplexing goroutines onto available threads
  - As a result, goroutines are cheap and easy to create
  - However, do cost resources and they are not garbage collected by runtime
  - Common to use a “done” channel to signify termination and defer close for channels created in goroutines

Concurrency Patterns in Go

- Error handling
  - If errors encountered in concurrent process, can send info as part of a struct
  - Rather than goroutine trying to print or handle error with limited context…
  - Can pass error to parts of the code that are in better position to interrogate and manage/handle

Concurrency Patterns in Go

- Pipelines
  - Using concurrency primitives, data can be passed (using channels) through a series of “phases”
  - At each “phase”, transformation logic or processing can be applied
  - Output from the current “phase” gets piped into the next phase for additional processing
  - Enables engineers to build a series of processing that combines sets of utility steps in different ways based on requirements

Demo

Lab

Serverless Architectures

Serverless / Functions-as-a-Service (FaaS)

- Represents a type of managed service provided by the CSP
- Cost structure is usually consumption-based (i.e., you only pay for what you use)
- Supports many different coding paradigms (C#/.NET, NodeJS, Python, Go ☺, etc.)
- Typically, with Serverless (and PaaS), the consumer is only concerned with the application code and data – elements of the CSP's "backbone" used to support are managed by the CSP
- Includes more sophisticated automated scaling capabilities – built for Internet scale

AWS Lambda Using Go

- Lambdas can be created in the AWS Management Console
- Allows language selection and testing or can use a blueprint as a starting point
- The in-browser IDE does not support Go which prevents editing in the Management Console

AWS Lambda Using Go

- But we’re able to zip it up, push it to an S3 bucket, and then deploy from there
- Likely a better development experience than the MC anyway
- In addition to the Lambda, you’ll want an API Gateway in place as well

AWS Lambda Using Go

- This Gateway will provide an HTTP REST interface to the Lambda’s operations
- The Gateway will be setup as a trigger (it’s one of Lambda’s standard triggers)
- The Gateway includes internal components that can proxy to/from the Lambda

AWS Lambda Using Go

- Go packages have been provided by AWS for coding the Lambda’s handler
- Provide several utility functions for managing function execution and results processing

Review Lambdas in AWS

Lab

Infrastructure-as-Code (IaC) Using Terraform

Terraform

- While microservice components can be created manually in the AWS MC, it’s not ideal
- IaC (Infrastructure-as-Code) is a better alternative
- Infrastructure code (like all other code) is code
- Can be tested, versioned, put in source control, etc.
- Multiple options available for tooling to support

What is Terraform?

- A form of “infrastructure as code”
- *Declarative* domain-specific language
  - What is declarative?
- Used to describe *idempotent* resource configurations, typically in cloud infrastructure
- According to Hashicorp:
  - *Terraform enables you to safely and predictably create, change, and improve infrastructure. It is an open-source tool that codifies APIs into declarative configuration files that can be shared amongst team members, treated as code, edited, reviewed, and versioned*

What is Terraform? (cont'd)

- Open source CLI tool for *infrastructure automation*
- Utilizes plugin architecture
  - Extensible to any environment, tool, or framework and works primarily by making API calls to those environments, tools, or frameworks
- Detects implicit dependencies between resources and automatically creates a dependency graph
- Builds in dependency order and automatically performs activities in parallel where possible
  - ...Sequentially for dependent resources

HashiCorp
Terraform

Why Use Terraform?

- Readable
- Repeatable
- Certainty (i.e., no confusion about what will happen)
- Standardized environments
- Provision quickly
- Disaster recovery

What Does Terraform (HCL) Look Like?

```
resource "aws_instance" "web" {
  ami                  = "ami-
19827362728"
  instance_type = "t2.micro"

  tags = {
    Name = "my-first-instance"
  }
}
```

Hashicorp Configuration Language (HCL)

- The goal of HCL is to build a structured configuration language that is both human and machine friendly for use with command-line tools, but specifically targeted towards DevOps tools, servers, etc.
- Fully JSON compatible
- Made up of **stanzas** or **blocks**, which roughly equate to JSON objects. Each stanza/block maps to an object type as defined by **Terraform providers** (we'll talk more about providers later)
- https://github.com/hashicorp/hcl

Terraform Project Content Types

`*.tf, *.tf.json`

- HCL or JSON
- these files define your declarative infrastructure and resources

`*.tfstate`

- JSON files that store state, reference to resources
- created and maintained by terraform

`terraform.tfvars, terraform.tfvars.json` and/or `*.auto.tfvars, *.auto.tfvars.json`

- HCL or JSON
- variable definitions in bulk
- (more to come on setting variable values at runtime)

Resources

- `*.tf` files contain your **HCL declarative** definitions

```
resource "aws_instance" "web" {
  ami           = "ami-19827362728"
  instance_type = "t2.micro"

  tags {
    Name = "my-first-instance"
  }
}
```

- Most **blocks** in your HCL represent a **resource** to be created/maintained by Terraform

Resources

- *Resources* are key elements and captured as top-level objects (stanzas) in Terraform configuration files
- Each resource stanza indicates the intent to *idempotently* create that resource
- Body of resource contains configuration of attributes of that resource
- Each provider (e.g., AWS, Azure, etc.) provides its own set of resources and defines the configuration attributes
- When a resource is created by Terraform, it's tracked in Terraform *state*
- Resources can refer to attributes of other resources, creating implicit dependencies
    - Dependencies trigger sequential creation

Terraform Commands and the CLI

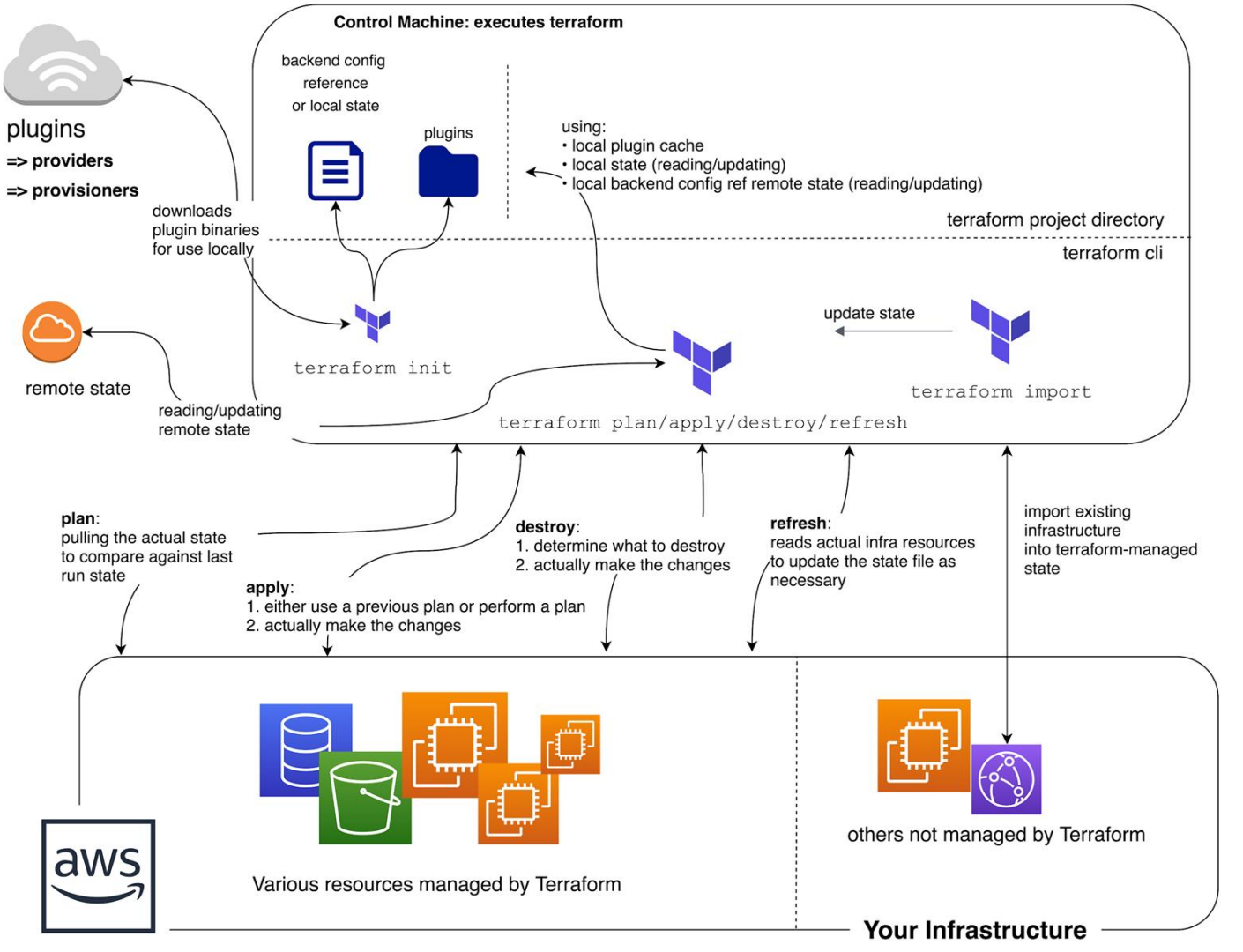
- The CLI is how you'll most often use terraform

```
terraform init …
terraform plan …
terraform apply …
```

- And plenty more: **`terraform --help`** or https://www.terraform.io/docs/commands/index.html
- Third-party SDKs also available for running and interacting with Terraform (e.g., **`scalr`, `terragrunt`, `terratest`**)

Big picture look at

Terraform Command Flow



`terraform init`

- A special command, run before other commands/operations
- What does it do?
  - Downloads required provider packages
  - Downloads modules referenced in the HCL (more on modules later)
  - Initializes state
    - Local state: ensuring local state file(s) exist
    - Remote state: more complex initialization (more on remote state later)
  - Basic syntax check
- Idempotent
- Uses a `.terraform` directory?
  - **`init`** downloads the provider packages and modules to this directory
  - Also, where state files live (locally)

Input Variables

- Enable interchangeable values to be stored centrally and referenced single or multiple times
- Similar to variables in other languages
- Declared in **`variable`** stanzas
- Parsed first
- Cannot interpolate or reference other variables
- Allow for default values
- Optionally specify value type, e.g.,
  - **`List, Map, String`**

Input Variables

- Input variable definitions support the following
  - default – provides default value if not specified; makes optional
  - type – type of value accepted for the variable
  - description – string description/documentation
  - validation – block for defining validation rules for input
  - sensitive – true or false; limits output as part of TF operations (plan or apply)

Example Variable Definition

```
variable "instance_size" {
  default     = "t2.micro"
  type        = string # changed
in 0.12
  description = "Size of EC2
instance"
}
```

Example Variable Definition

```
variable "student_alias" {
  type        = string
  description = "Your student alias"
  validation {
    condition     = trimprefix(var.student_alias, "test") ==
var.student_alias
    error_message = "Please do not use test aliases with this
deployment."
  }
}
```

Data Sources

- Logical references to data objects stored externally to the `tfstate` file
- Allows you to reference resources not created by Terraform
- Examples
  - current default region in AWS CLI
  - AMI ID search
  - AWS ARN lookup
  - AWS VPC CIDR range

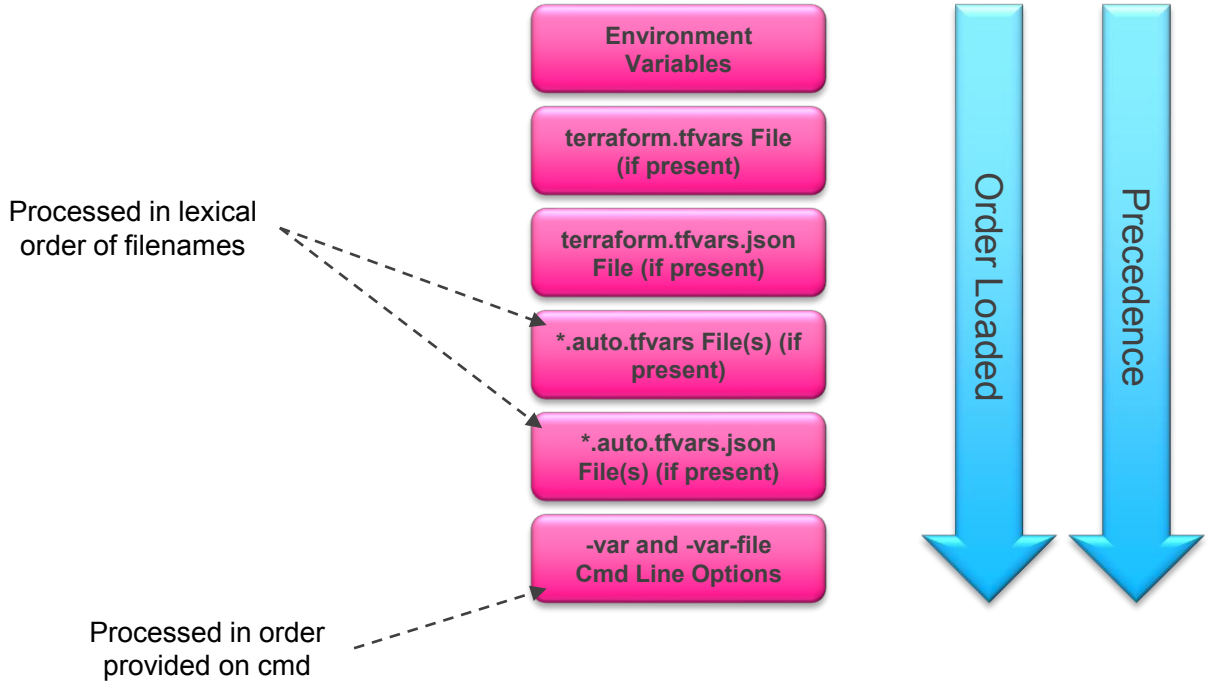
Data Source Example: AWS AMI Lookup

```
data "aws_ami" "latest-ubuntu" {
  most_recent = true
  owners = ["099720109477"]

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-
server-*"]
   }

  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }
}
```

Providing Values for Input Variables

- Multiple options
  - Using environment variables (prefixed with “TF_VAR_”)
  - Defining inputs in a terraform.tfvars file
  - Defining inputs in a terraform.tfvars.json file
  - Defining inputs in one or more *.auto.tfvars files
  - Defining inputs in one or more *.auto.tfvars.json files
  - -var and -var-file options on the command-line

Providing Values for Input Variables



Providing Values for Input Variables

- Primarily used when executing Terraform via CLI
- Not really used with Terraform Enterprise
- Can “push” those variables + values to Enterprise (in files)
- But manage from “Variables” section of the environment

State

- Stores information about resources that are created by Terraform
  - Also includes values computed by the provider APIs

- Local file
  - `.tfstate`

- Or backends are also available…

Backends

- Determines how state is loaded and how operations like **`apply`** are executed

- Enables non-local file state storage, remote execution, etc.

- Why use a backend?
  - Can store their state remotely and protect it to prevent corruption
    - Some backends, e.g., *Terraform Cloud* automatically store all revisions
  - Keep sensitive information off local disk
  - Remote operations
    - Apply can take a *LONG* time for large infrastructures

Backends (cont'd)

- Examples
  - S3
  - swift
  - http
  - Terraform Enterprise
  - etc.

Providers

- Responsible for understanding API interactions and exposing resources
- Hashicorp helps companies create providers to be added to ecosystem
- Declared in HCL config files as a **`provider`** stanza
- Each Terraform project can have multiple providers, even of the same type
- Describes resources, their inputs, outputs, and the logic to create and change them
- Many options
  - AWS, GCP, Azure, and many many others
  - providers available for non-infra services as well such as gmail, MySQL, and Pagerduty

The AWS Provider

- Provider documentation
  - https://www.terraform.io/docs/providers/aws/index.html

- HUGE amount of resources

- Something like 8 resources per service on average

Configuring the Provider

```
provider "aws" {
  region     = "us-west-1"
  access_key = "[your access key]"
  secret_key = "[your secret access
key]"
}
```

Output Variables

- *inputs* to a Terraform config are declared with variables stanzas
- *outputs* are declared with a special output stanza
- Can be referenced through the modules interface or the CLI

Output Variables

- Output variable definitions support the following
  - value – value to be returned as output
  - description – string description/documentation
  - sensitive – true or false; limits output as part of TF operations (plan or apply)

Output Definition

```
output "instance_public_ip" {
  value = aws_instance.web.public_ip
}
```

Demo

https://github.com/ludesdeveloper/terraform-lambda-golang

Lab

Docker

Setting Up a Local 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

```
# Grabs OpenJDK image upon which the new image will be based
FROM openjdk:17

# Creates a new target folder in image
RUN mkdir /usr/src/JavaDemoApp

# Copies current directory contents to newly created folder
COPY . /usr/src/JavaDemoApp

# Switches working directory in image to app folder
WORKDIR /usr/src/JavaDemoApp

# Compiles/builds Java app
RUN javac JavaDemo.java

# Executes new Java app
CMD ["java", "JavaDemo"]
```

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

Docker Hub

- 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

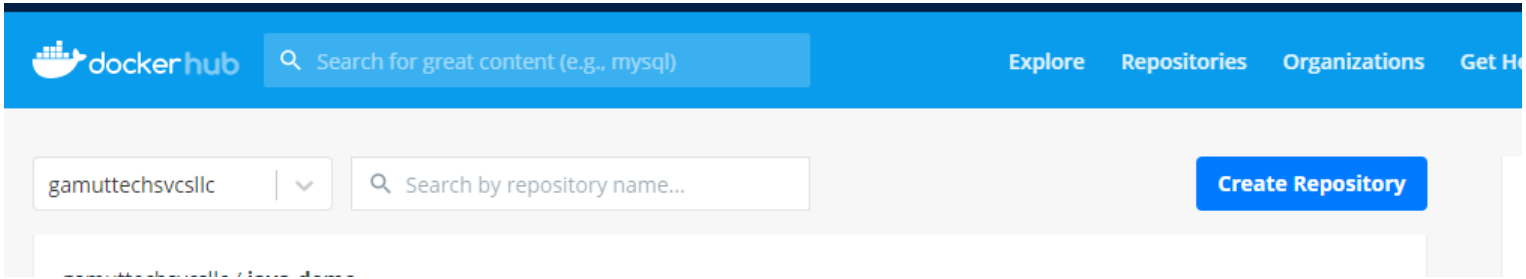
Docker Hub

- 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

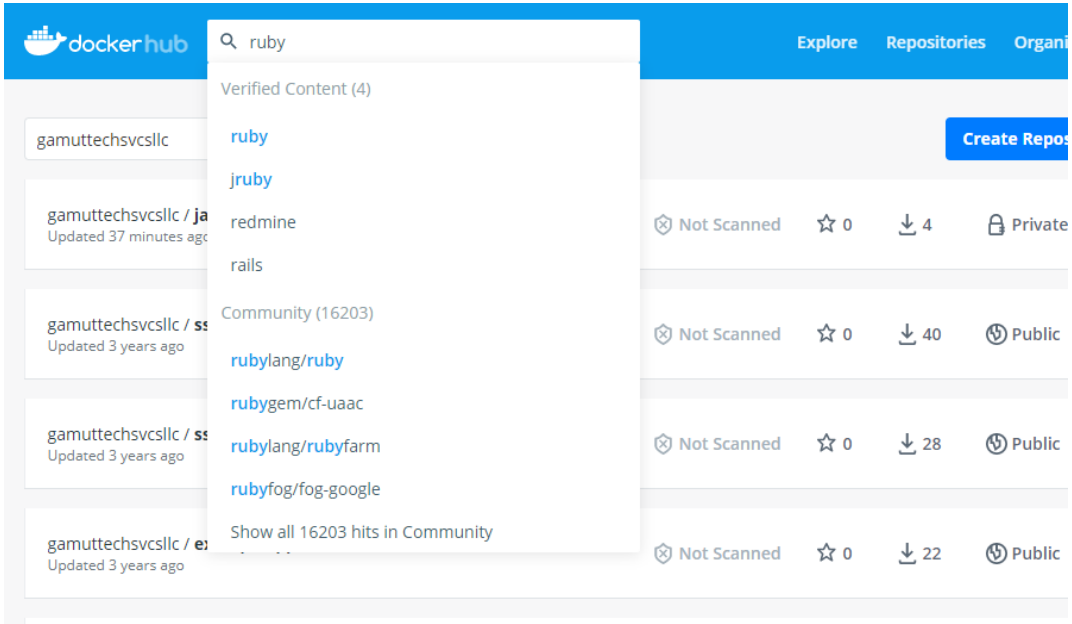
Docker Hub

There are other registry types, including private registries

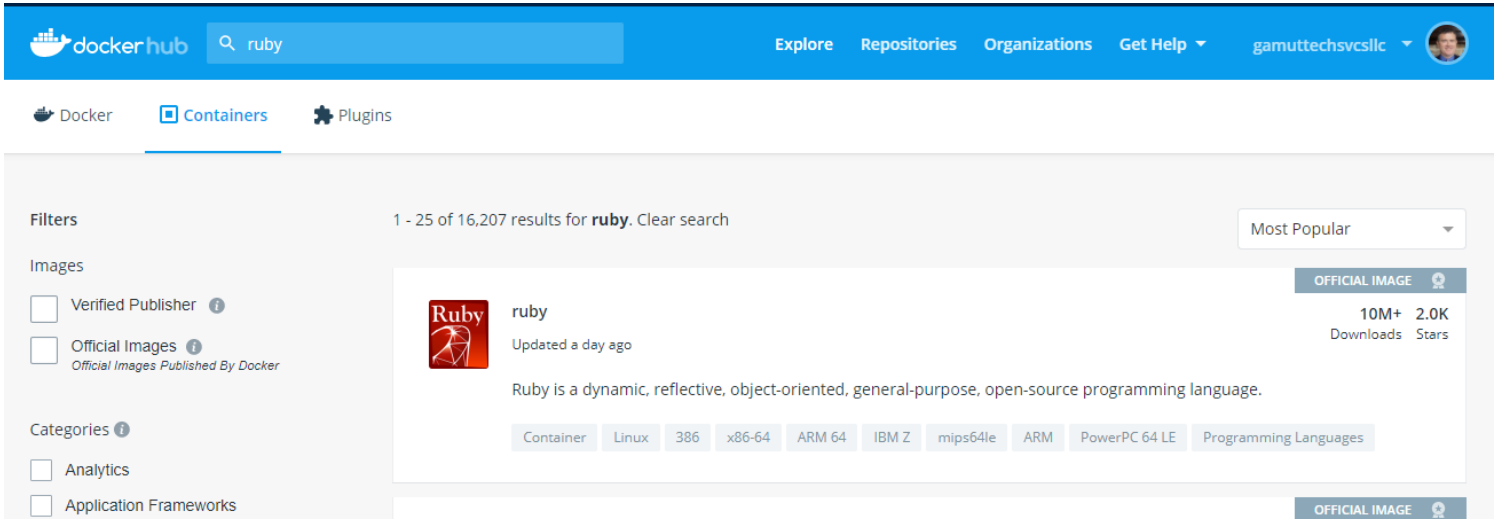
Docker Hub


dockerhub
Search for great content (e.g., mysql)
Explore
Repositories
Organizations
Get H
gamuttechsvcsllc
Search by repository name...
Create Repository

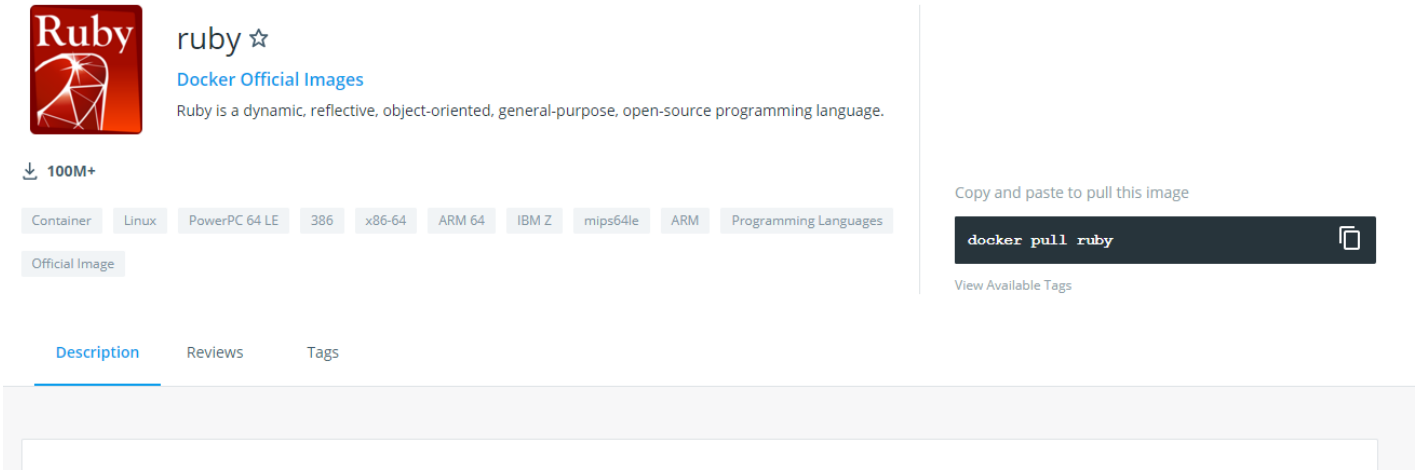
Docker Hub


dockerhub
ruby
Explore
Repositories
Organi
Verified Content (4)
ruby
jruby
redmine
rails
Community (16203)
rubylang/ruby
rubygem/cf-uaac
rubylang/rubyfarm
rubyfog/fog-google
Show all 16203 hits in Community
gamuttechsvcsllc
Create Repos
gamuttechsvcsllc / ja
Updated 37 minutes ago
Not Scanned
0
4
Private
gamuttechsvcsllc / ss
Updated 3 years ago
Not Scanned
0
40
Public
gamuttechsvcsllc / ss
Updated 3 years ago
Not Scanned
0
28
Public
gamuttechsvcsllc / ez
Updated 3 years ago
Not Scanned
0
22
Public

Docker Hub


dockerhub
ruby
Explore
Repositories
Organizations
Get Help
gamuttechsvcsllc
Docker
Containers
Plugins
Filters
1 - 25 of 16,207 results for ruby. Clear search
Most Popular
Images
Verified Publisher
Official Images
Official Images Published By Docker
OFFICIAL IMAGE
ruby
Updated a day ago
10M+ Downloads
2.0K Stars
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
Categories
Analytics
Application Frameworks
OFFICIAL IMAGE

Docker Hub


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
Copy and paste to pull this image
docker pull ruby
View Available Tags
Description
Reviews
Tags

Docker Hub

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`

Building The Image

- 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

Demo

Continue Lab

Kubernetes & Container Orchestration

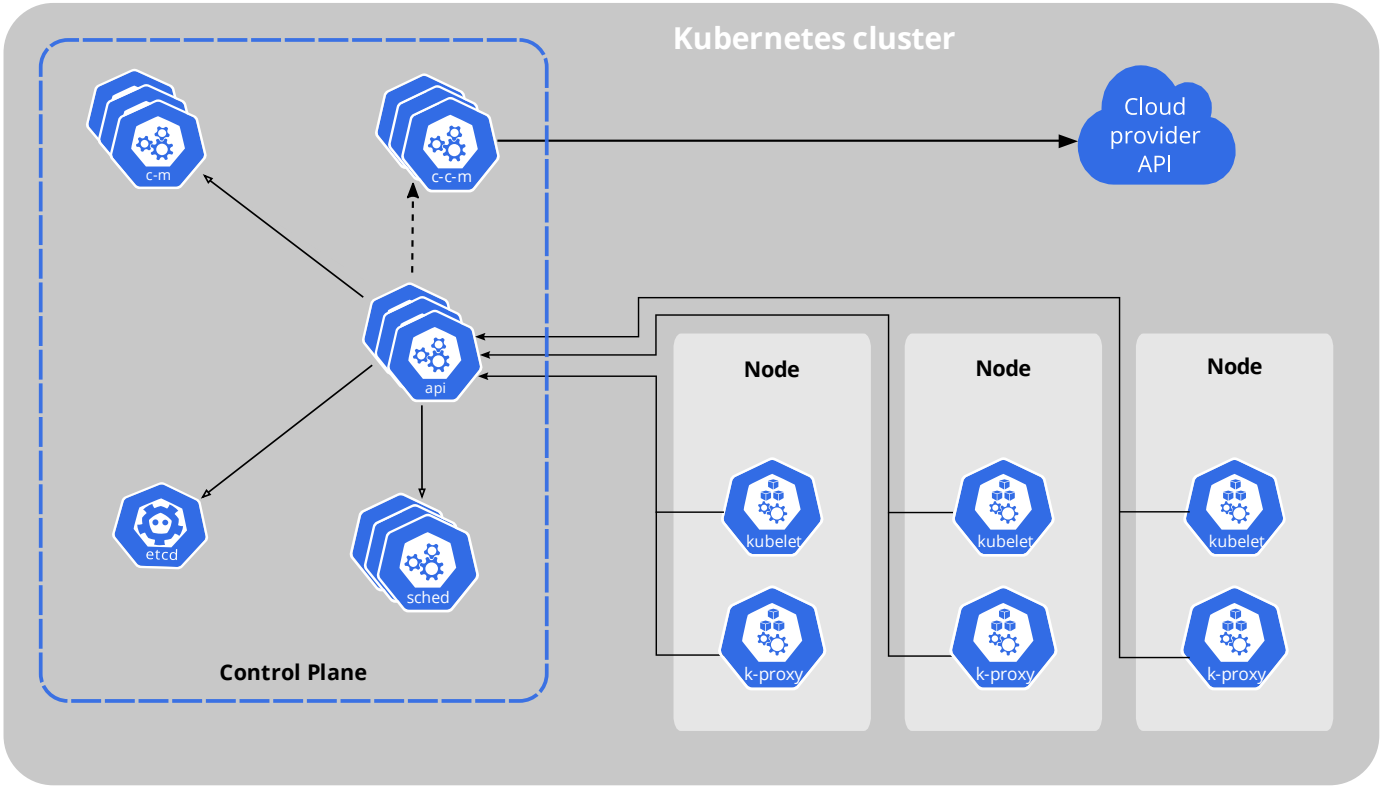
Kubernetes (k8s) Overview

- 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



API server

Cloud controller manager *(optional)*

Controller manager

etcd (persistence store)

kubelet

kube-proxy

Scheduler

Control plane

Node

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

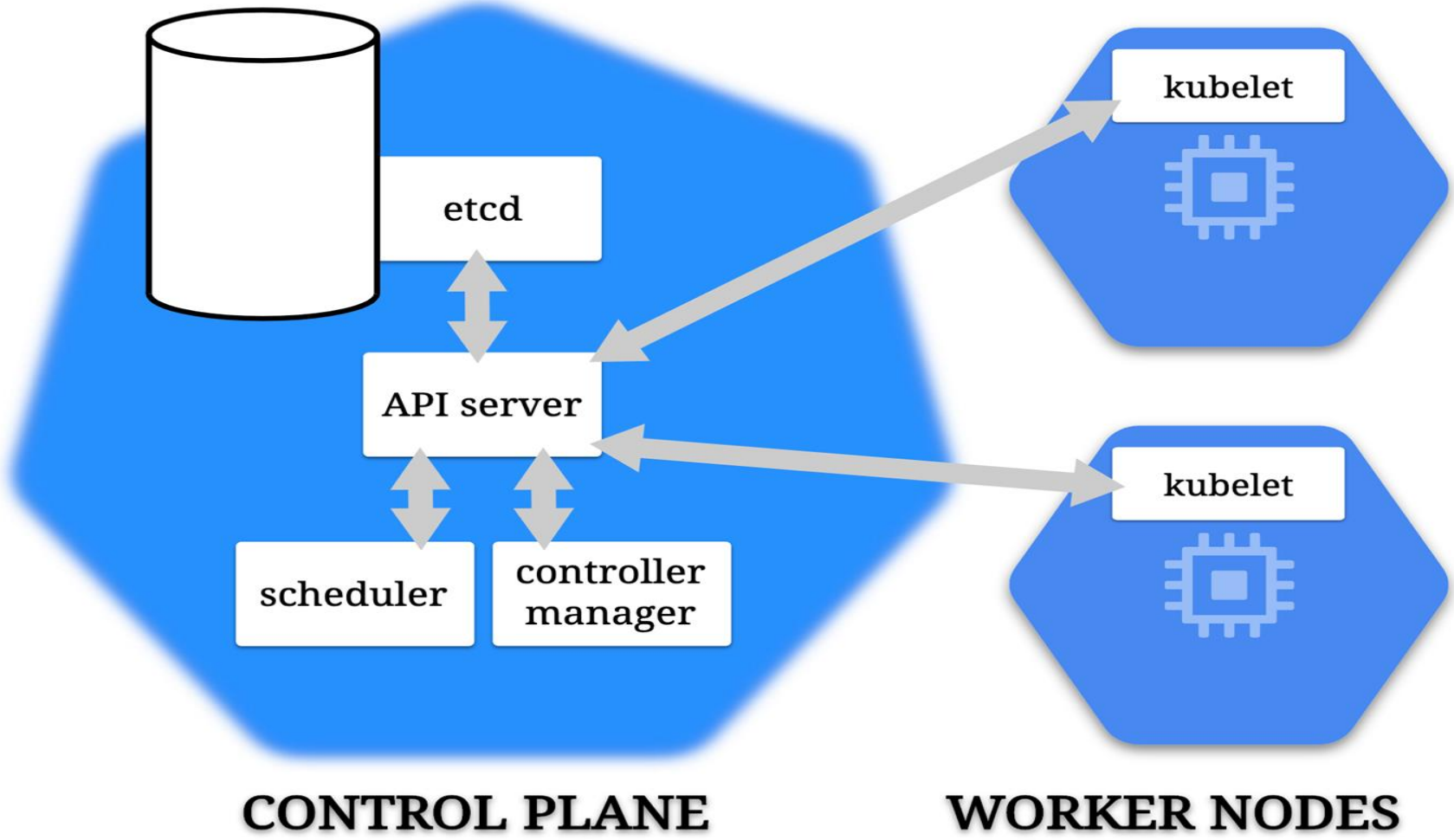
- Kubernetes is actually a collection of pods implementing the k8s api and the cluster orchestration logic (AKA the Kubernetes control plane)
- The application/data plane is everything else, meaning all the nodes that host all the pods that the application runs on
- The *controllers* of the control plane 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

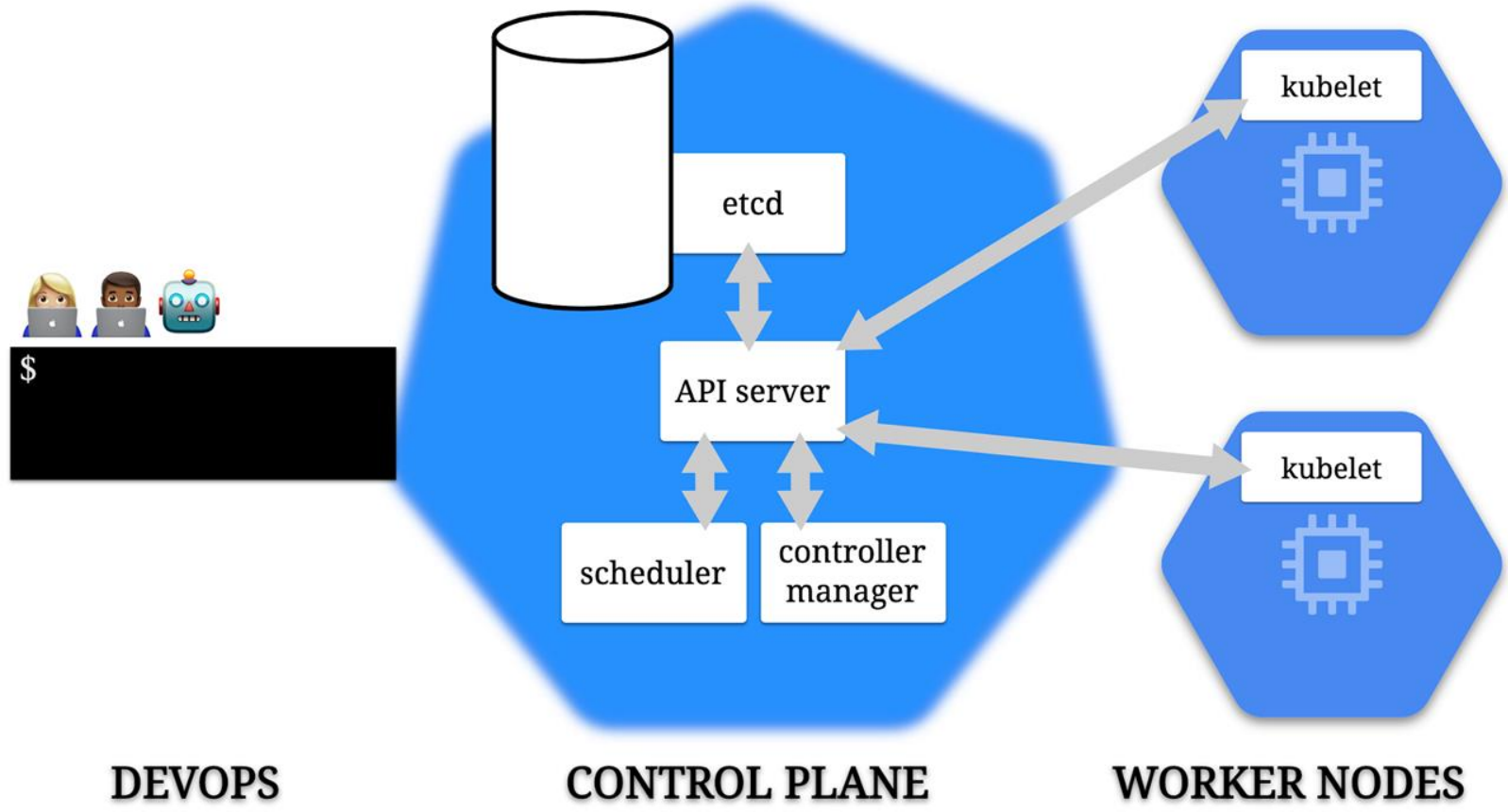
Declarative and Desired State

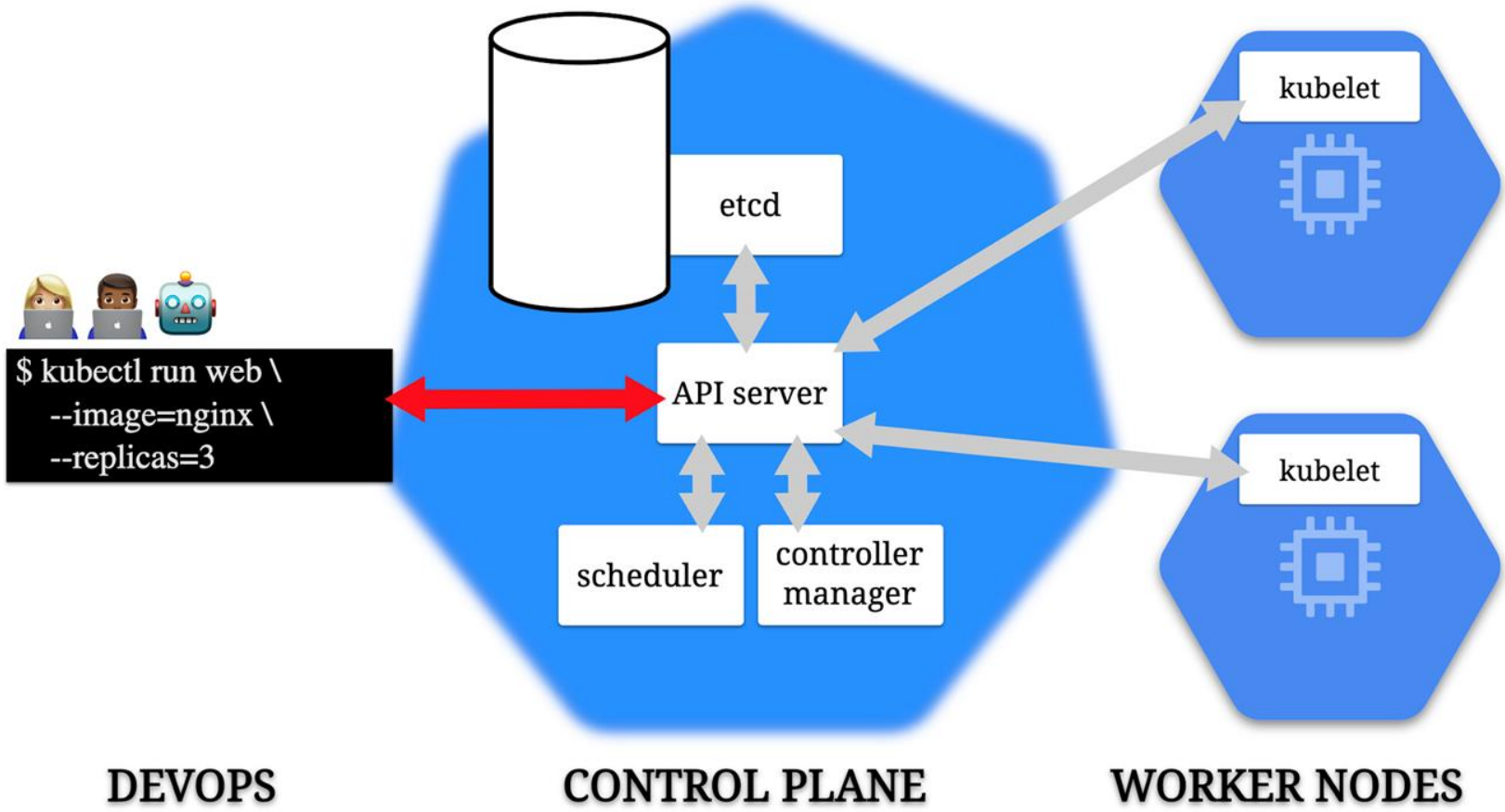
- Kubernetes supports a declarative model – we can send the api a yaml file in a declarative form
- Desired state means that we specify in the yaml file our desired state and the cluster takes responsibility to make sure it will happen
- We describe the desired state using a yaml or json file that serves 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

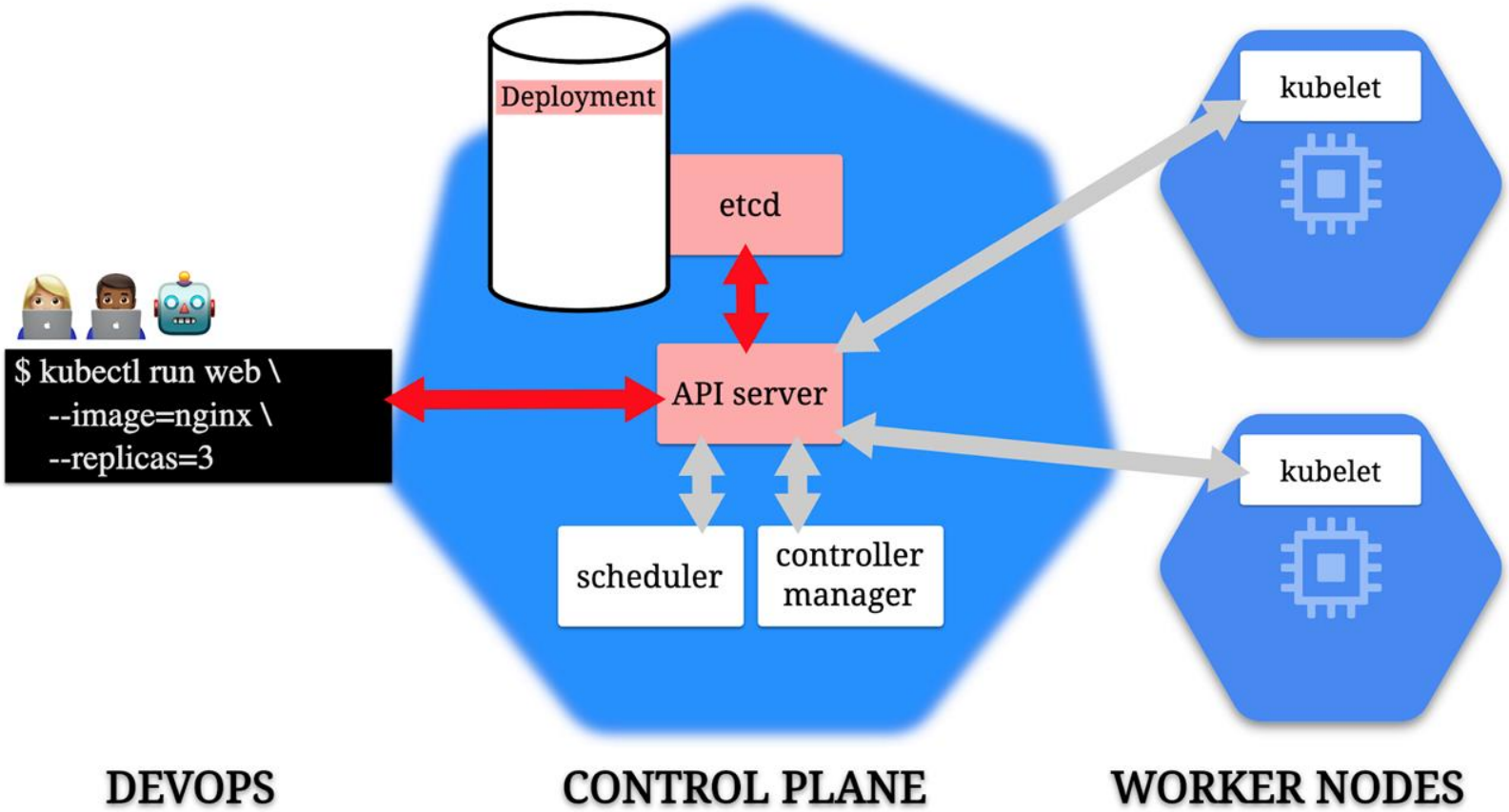
Reconciling state

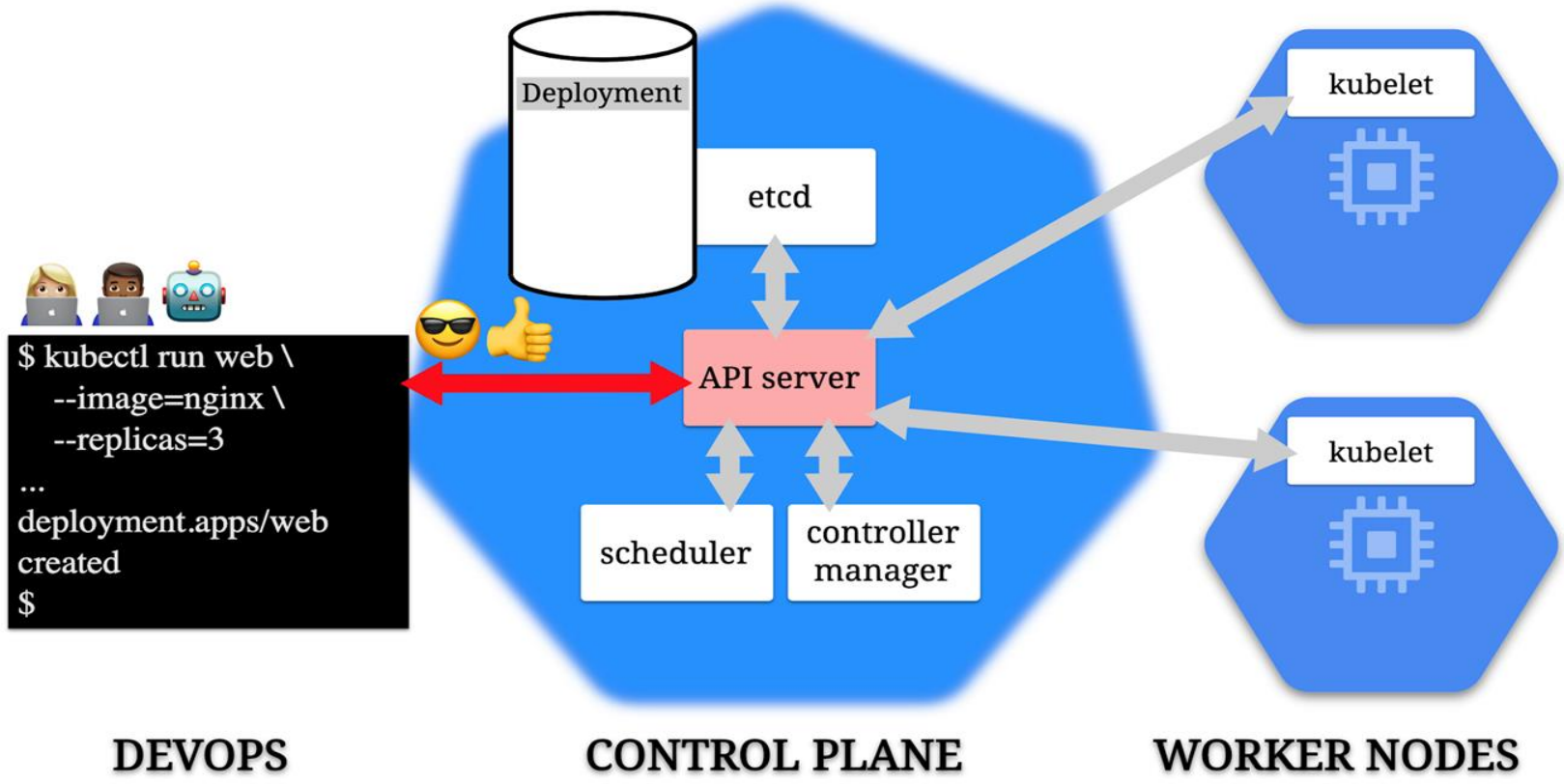
- Watch for the spec fields in the YAML files
- The *spec* describes *what we want the thing to be*
- Kubernetes will *reconcile* the current state with the spec (technically, this is done by a number of *controllers*)
- When we want to change a resource, we update the *spec*, and reapply
- Kubernetes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

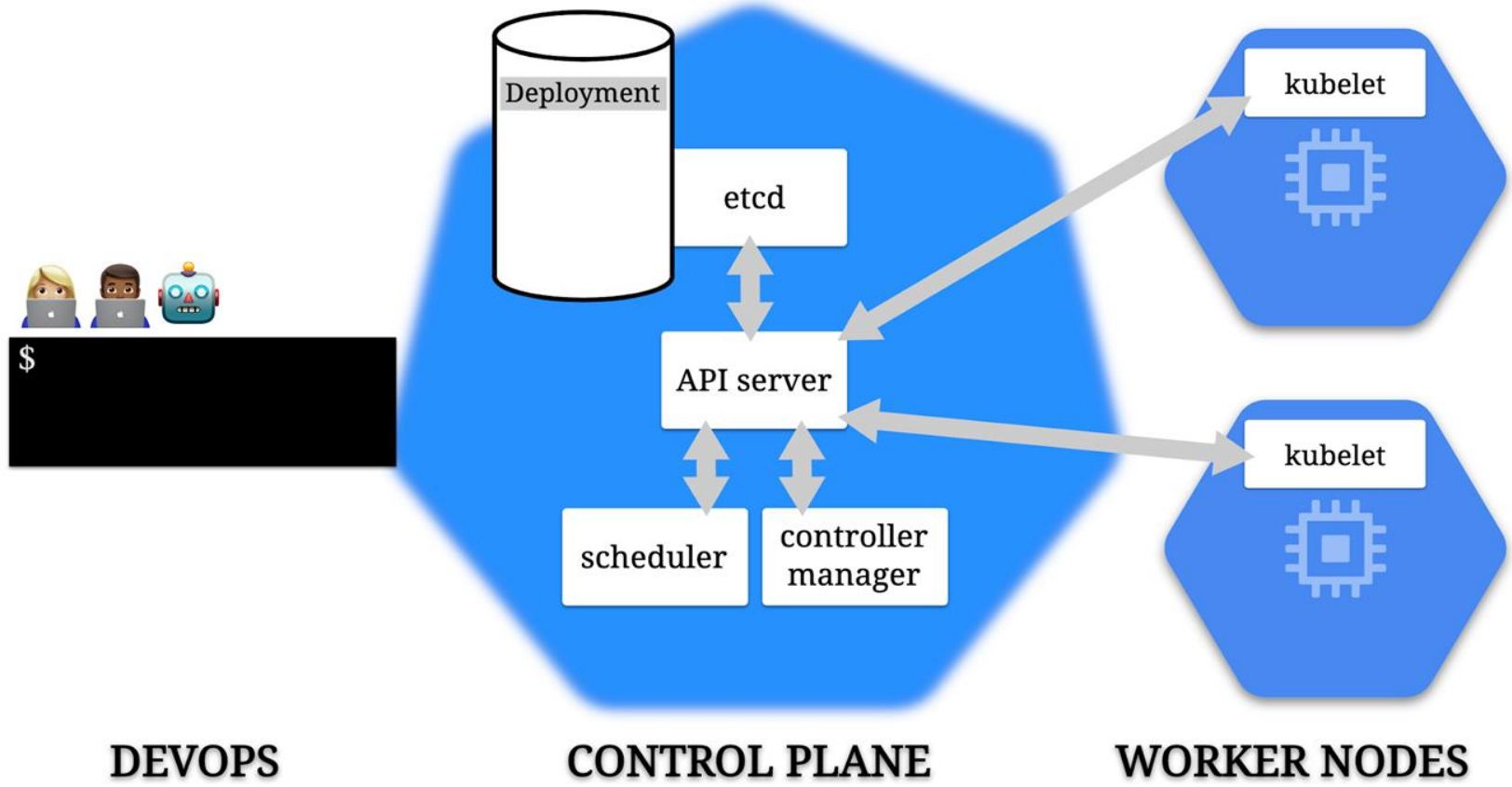

etcd
API server
scheduler
controller manager
kubelet
kubelet
CONTROL PLANE
WORKER NODES

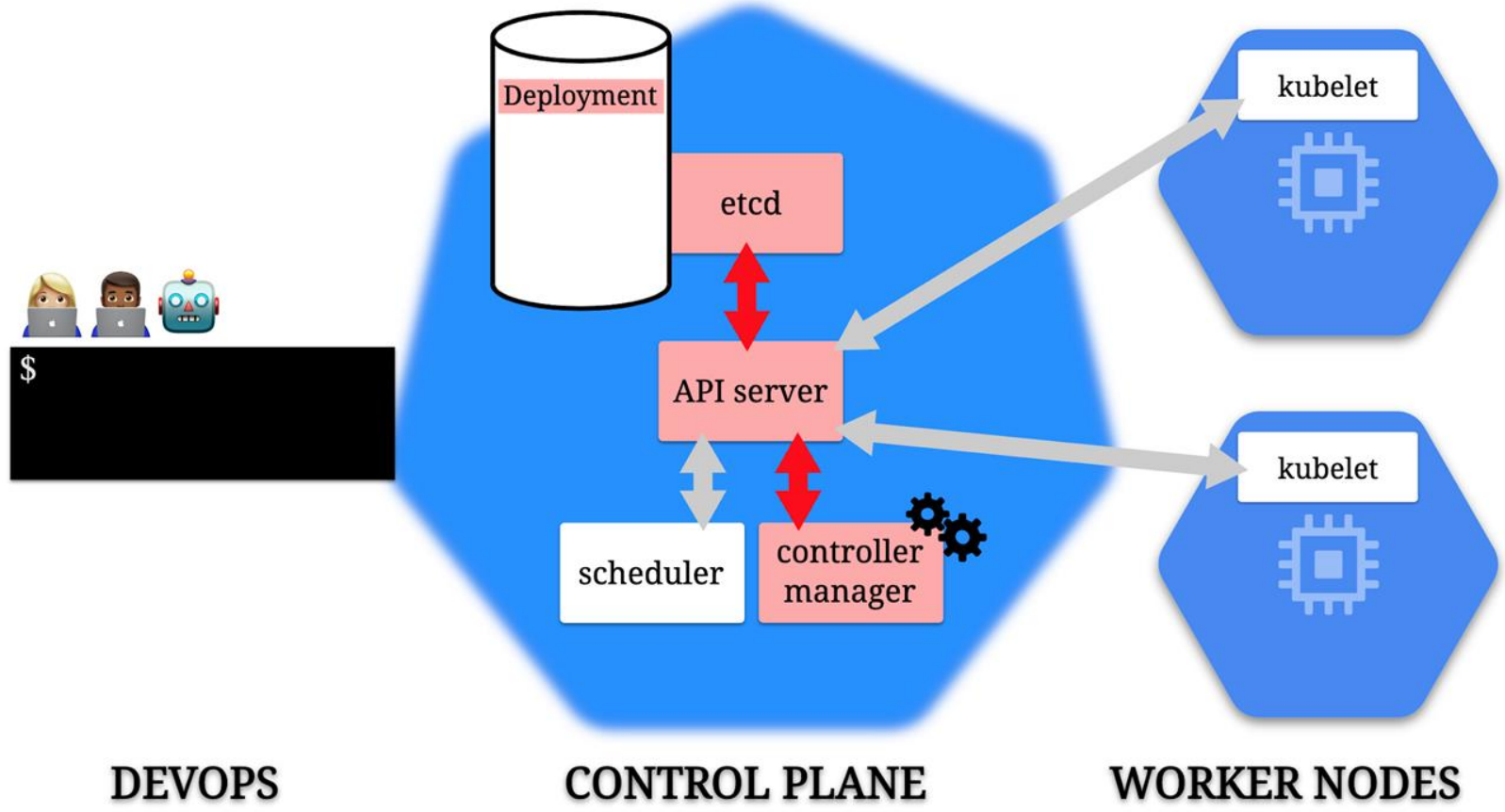

etcd
API server
scheduler
controller manager
kubelet
kubelet
$
DEVOPS
CONTROL PLANE
WORKER NODES

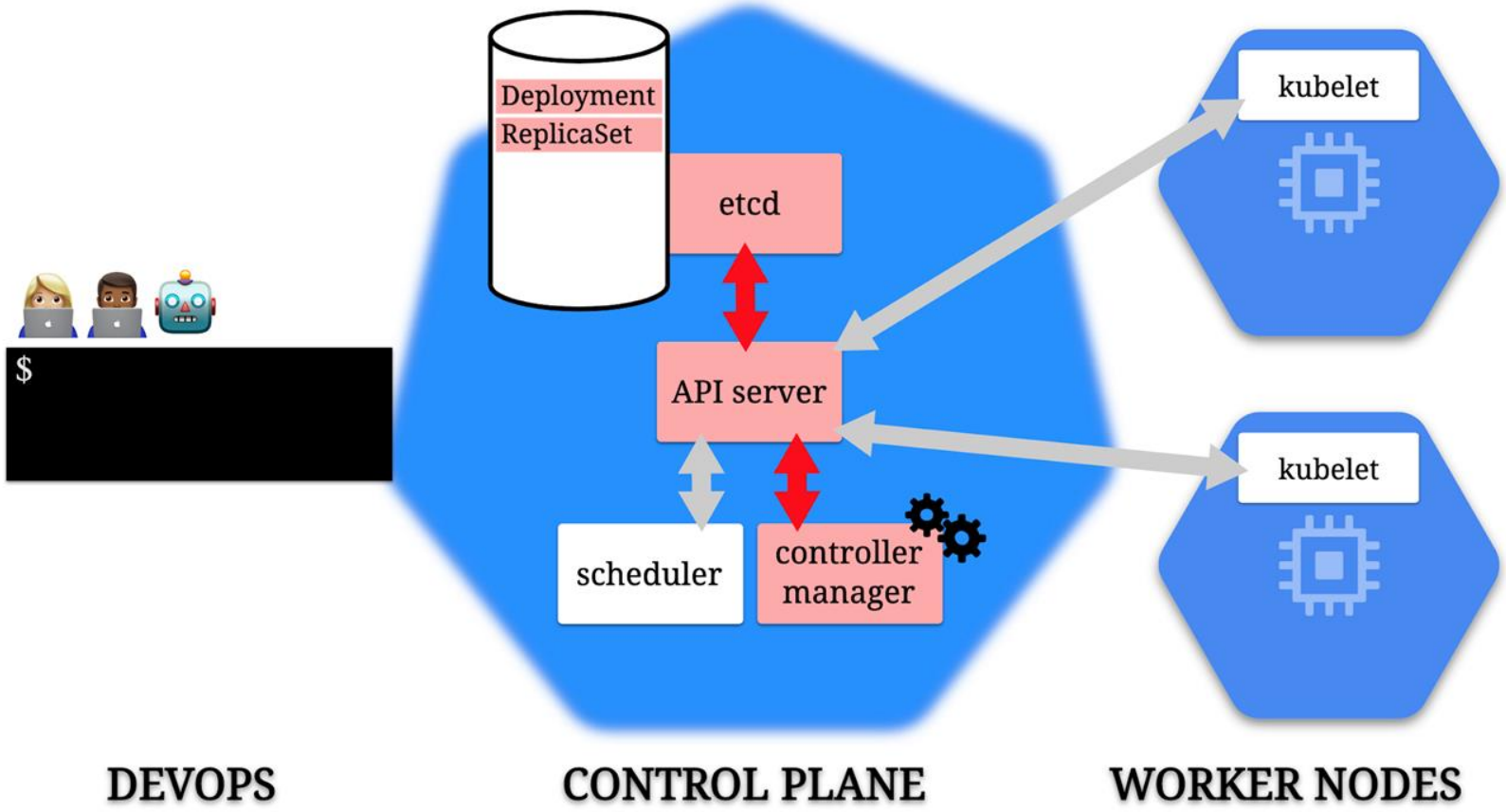

etcd
kubelet
$ kubectl run web \
--image=nginx \
--replicas=3
API server
kubelet
scheduler
controller manager
DEVOPS
CONTROL PLANE
WORKER NODES

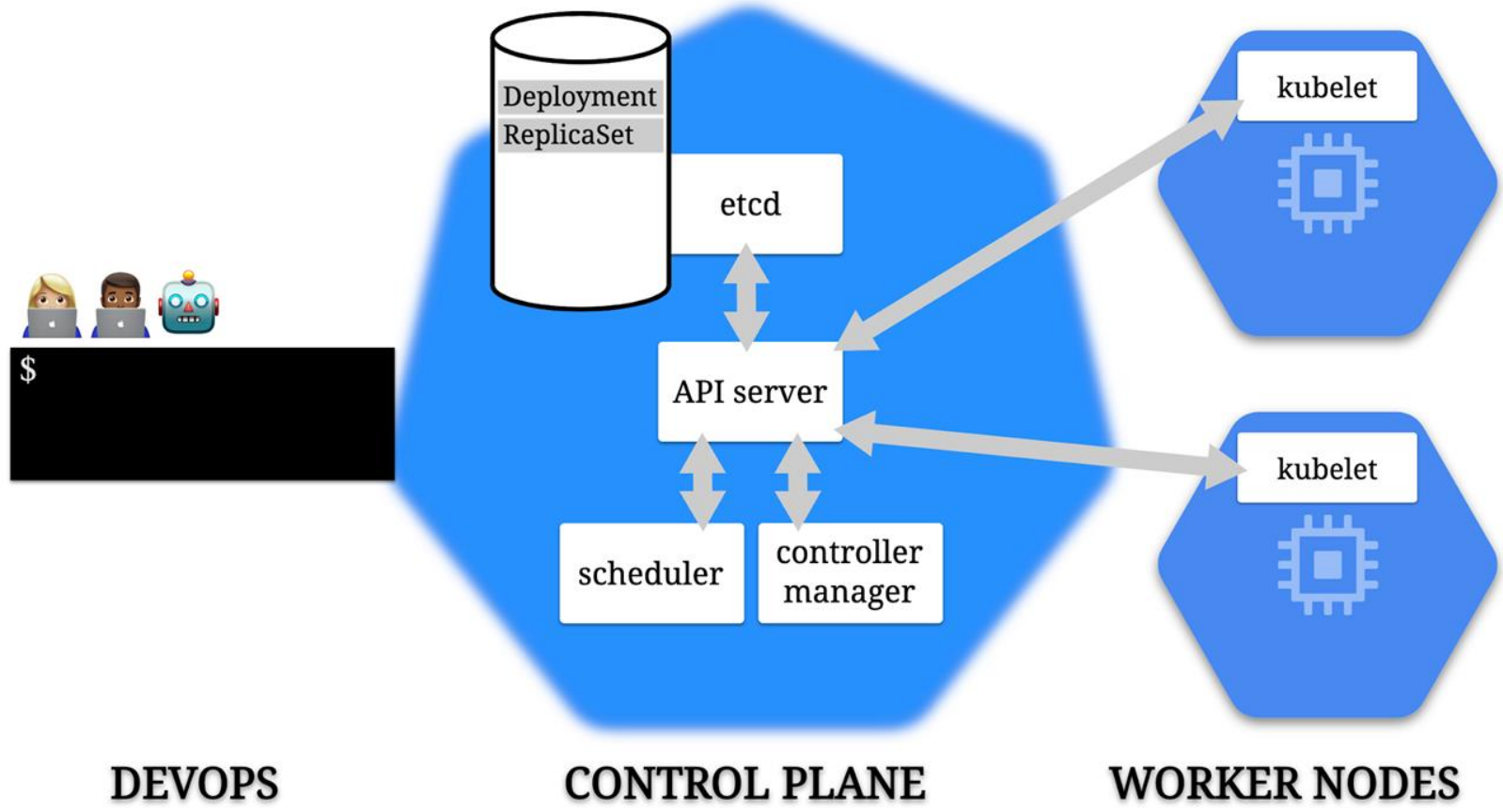

Deployment
etcd
kubelet
$ kubectl run web \
--image=nginx \
--replicas=3
API server
kubelet
scheduler
controller manager
DEVOPS
CONTROL PLANE
WORKER NODES

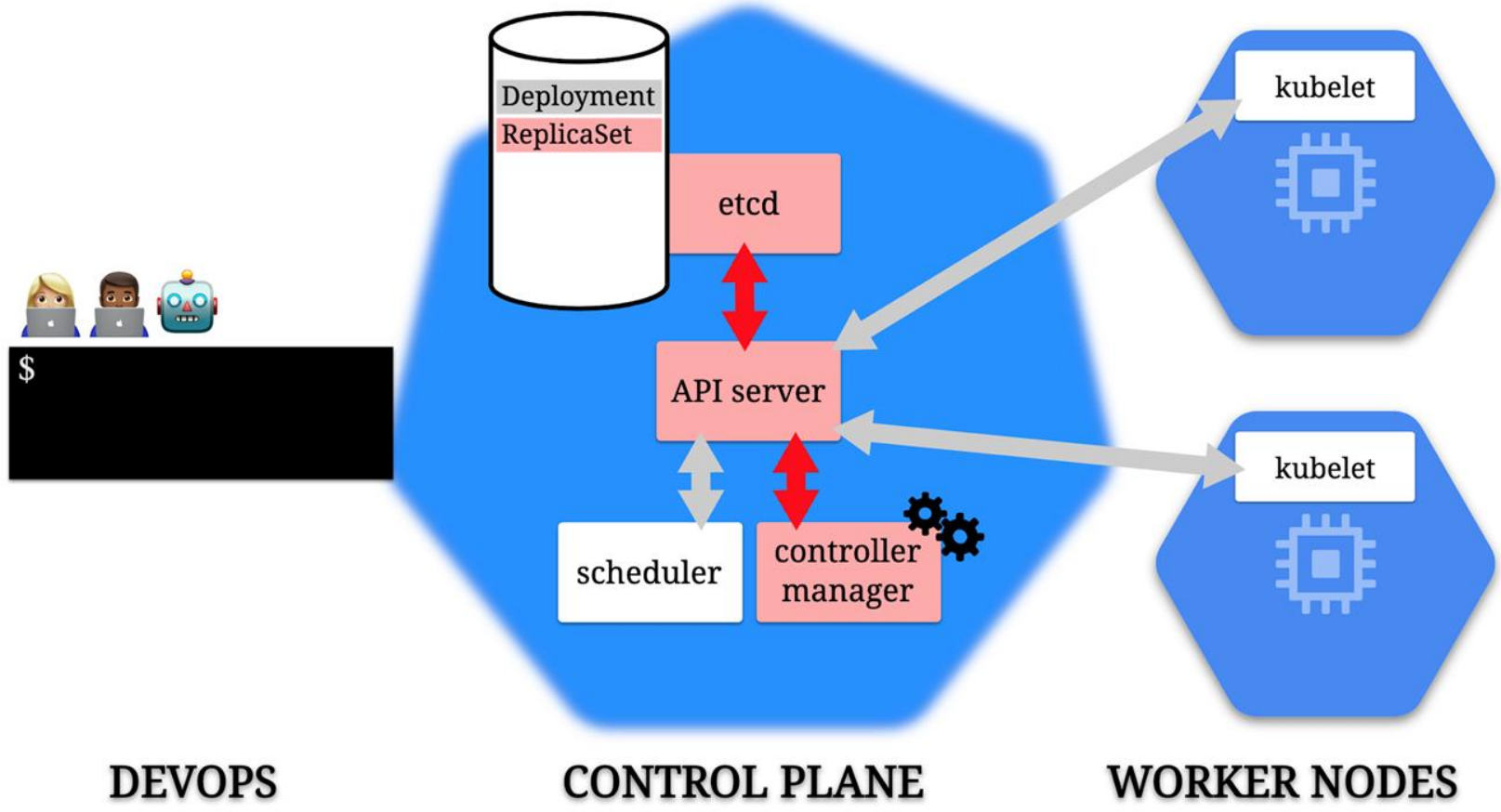

Deployment
etcd
kubelet
$ kubectl run web \
--image=nginx \
--replicas=3
...
deployment.apps/web
created
$
API server
kubelet
scheduler
controller
manager
DEVOPS
CONTROL PLANE
WORKER NODES

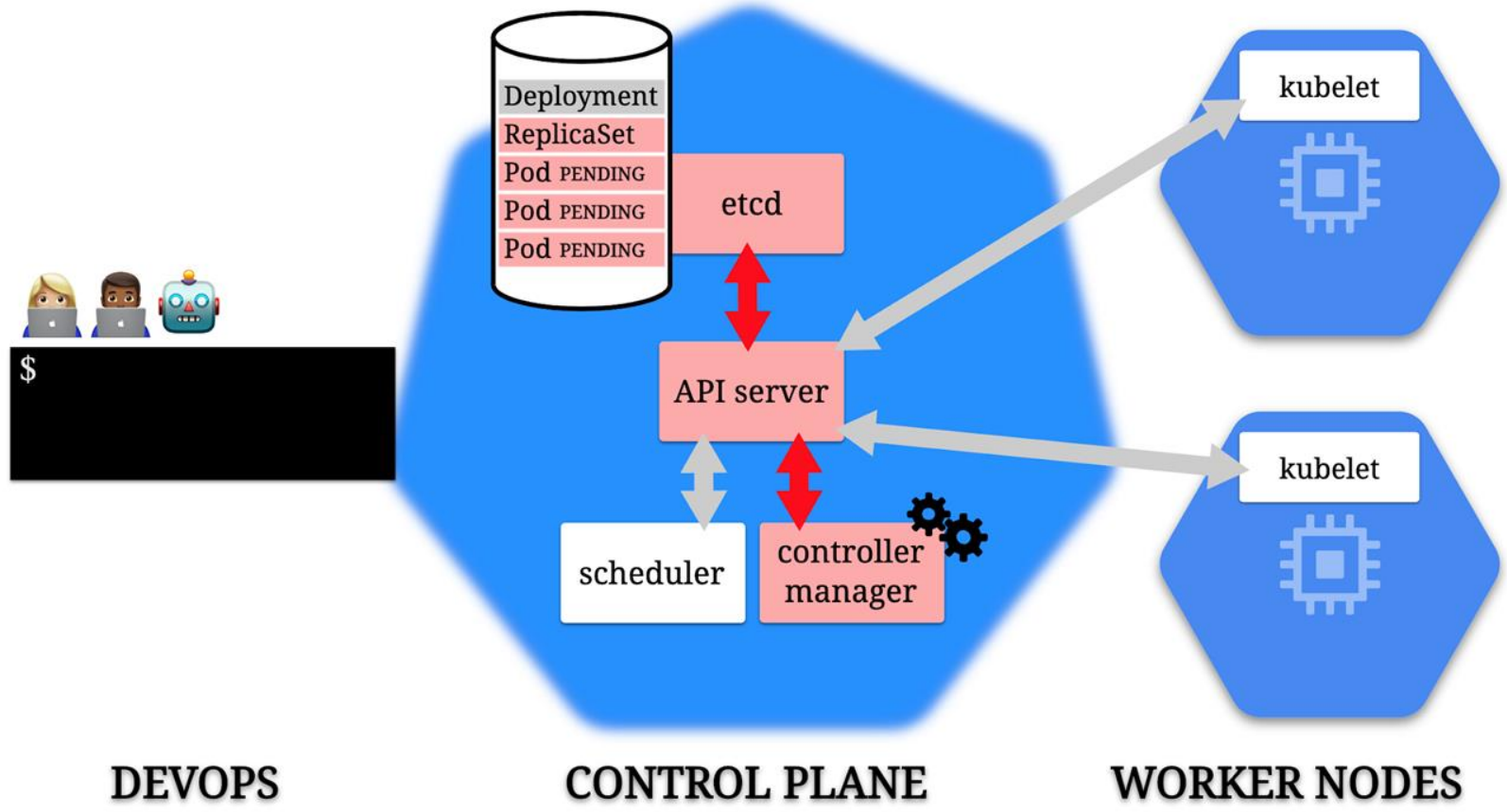

Deployment
etcd
API server
scheduler
controller manager
kubelet
kubelet
$
DEVOPS
CONTROL PLANE
WORKER NODES

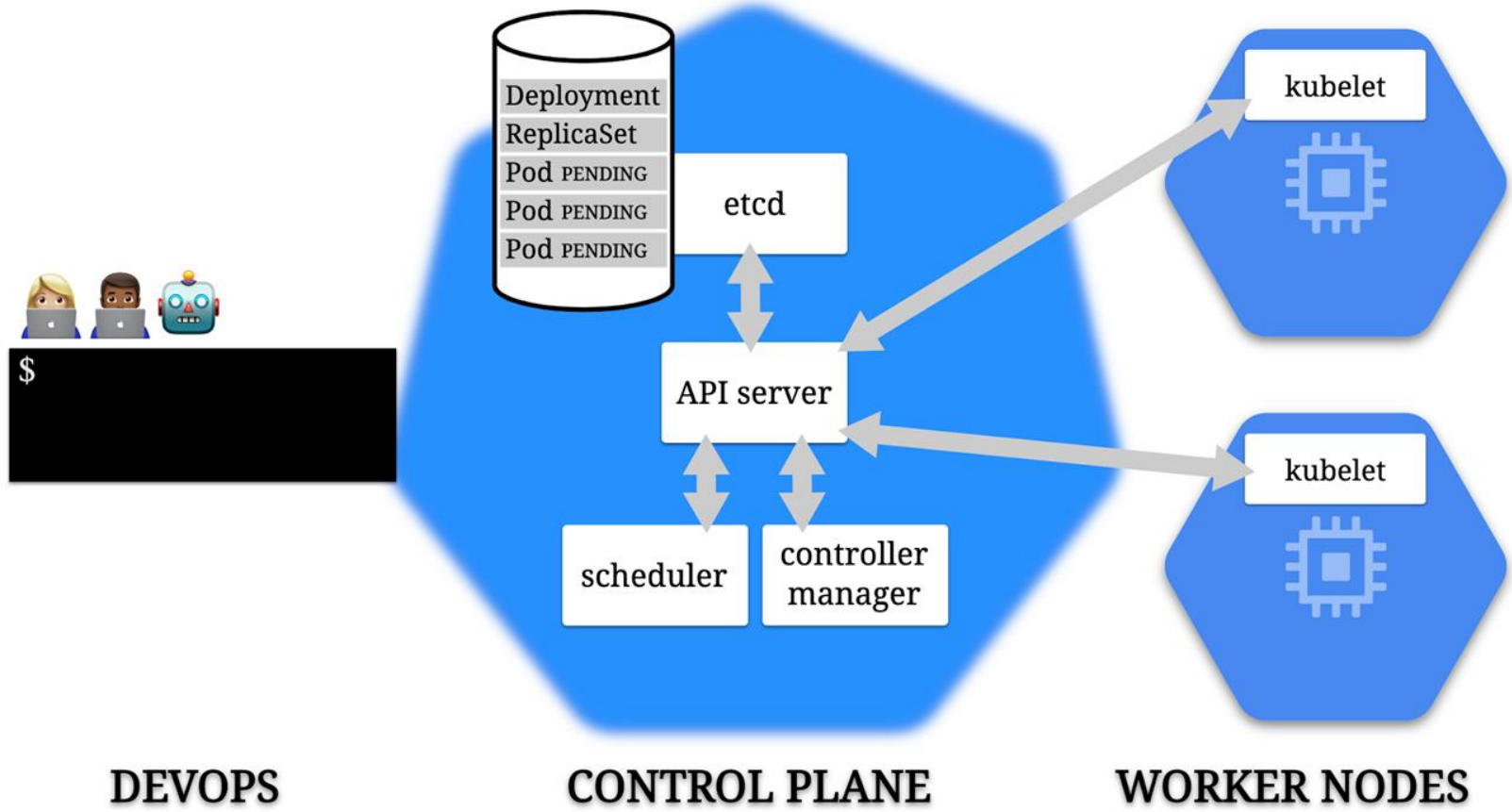

Deployment
etcd
API server
scheduler
controller manager
kubelet
kubelet
$
DEVOPS
CONTROL PLANE
WORKER NODES

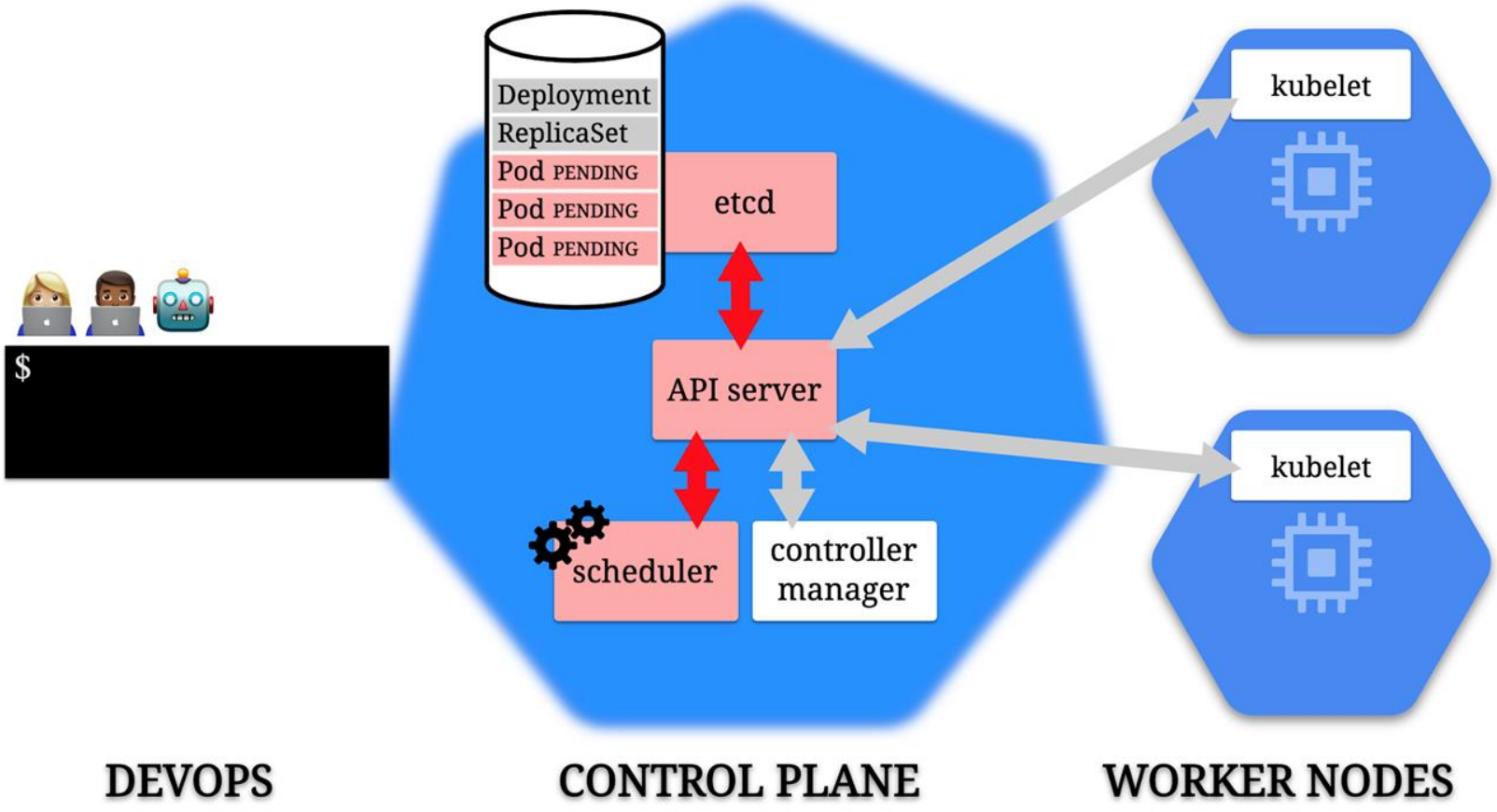

Deployment
ReplicaSet
etcd
API server
scheduler
controller manager
kubelet
kubelet
$
DEVOPS
CONTROL PLANE
WORKER NODES

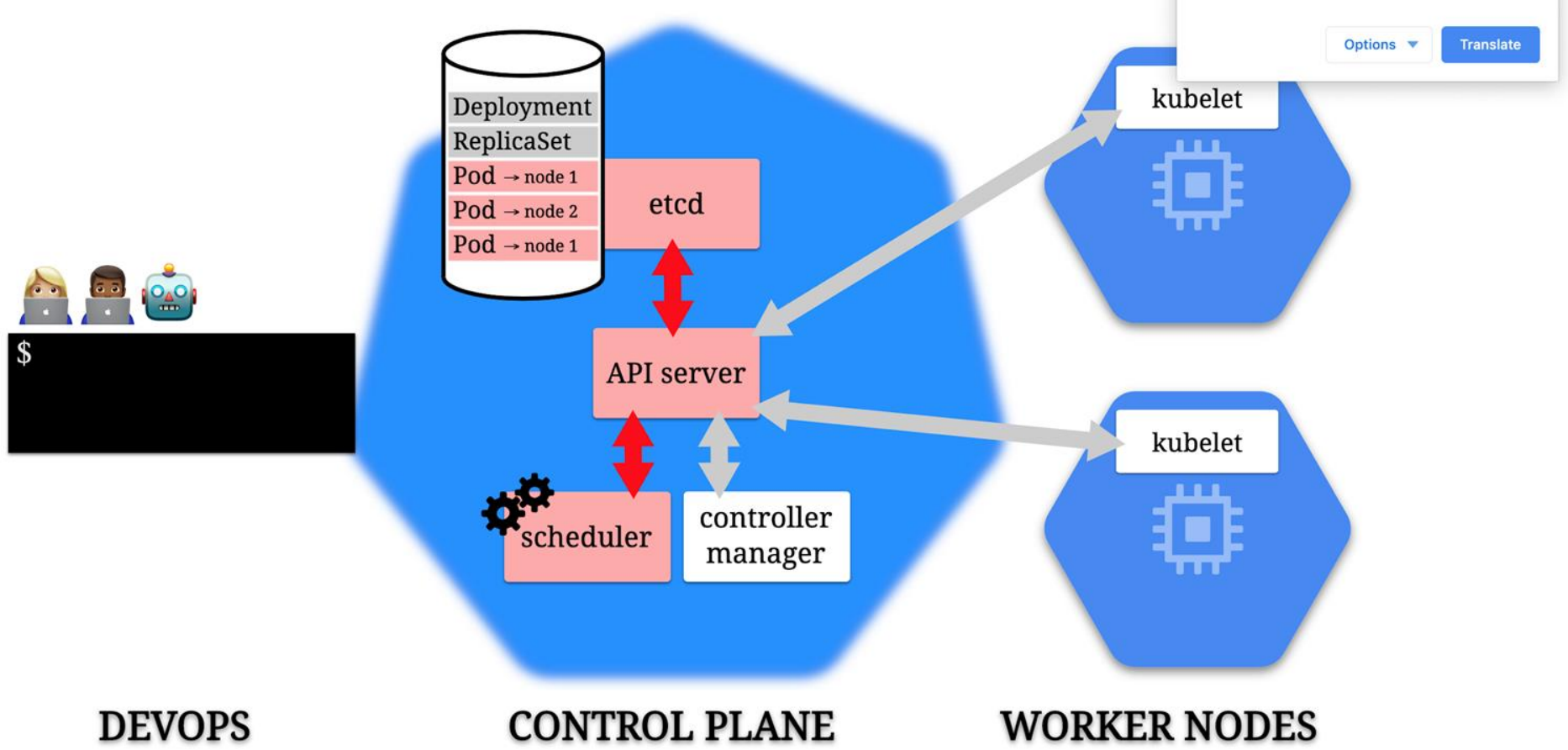

Deployment
ReplicaSet
etcd
API server
scheduler
controller manager
kubelet
kubelet
$
DEVOPS
CONTROL PLANE
WORKER NODES

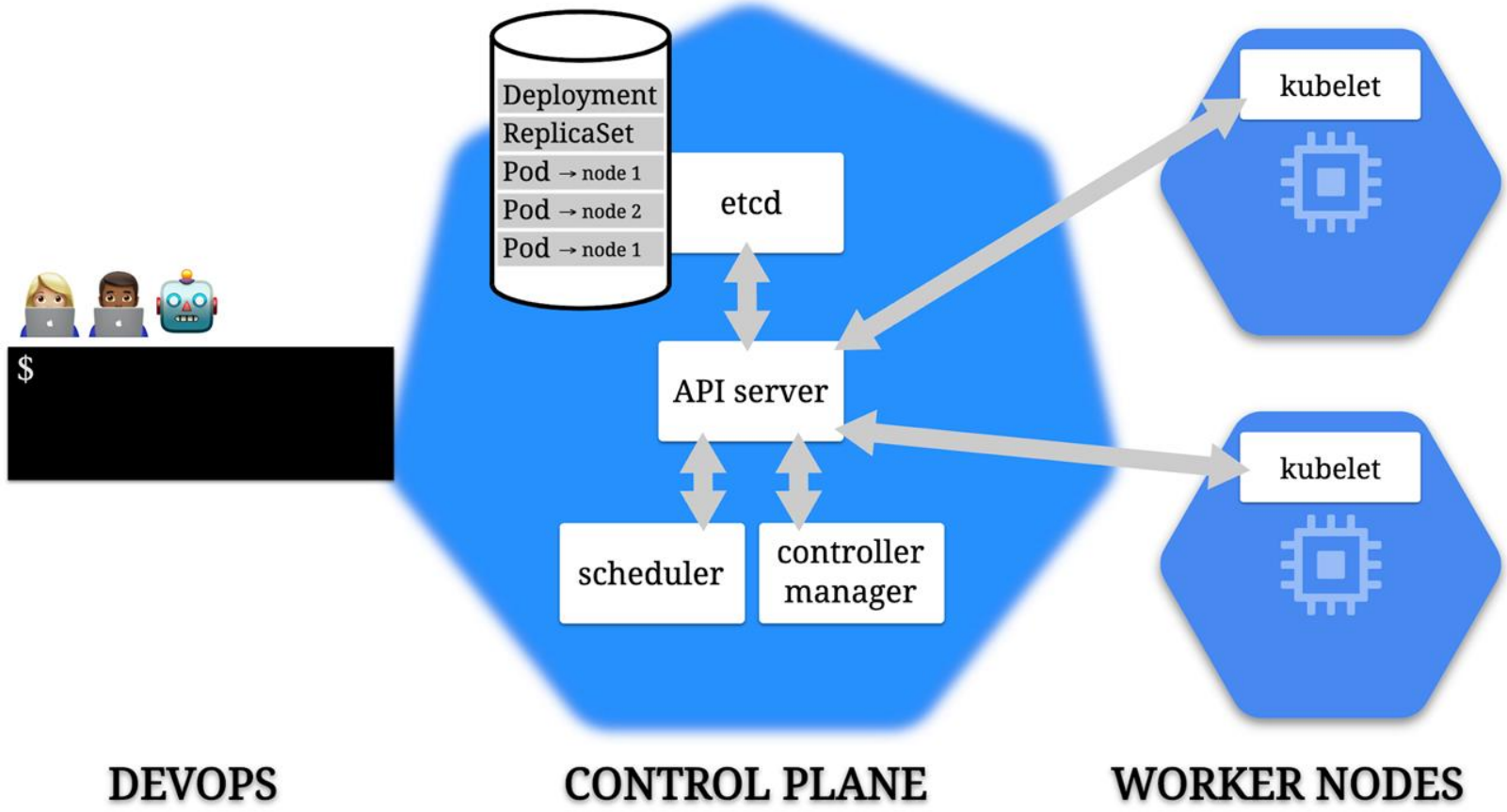

Deployment
ReplicaSet
etcd
API server
scheduler
controller manager
kubelet
kubelet
$
DEVOPS
CONTROL PLANE
WORKER NODES

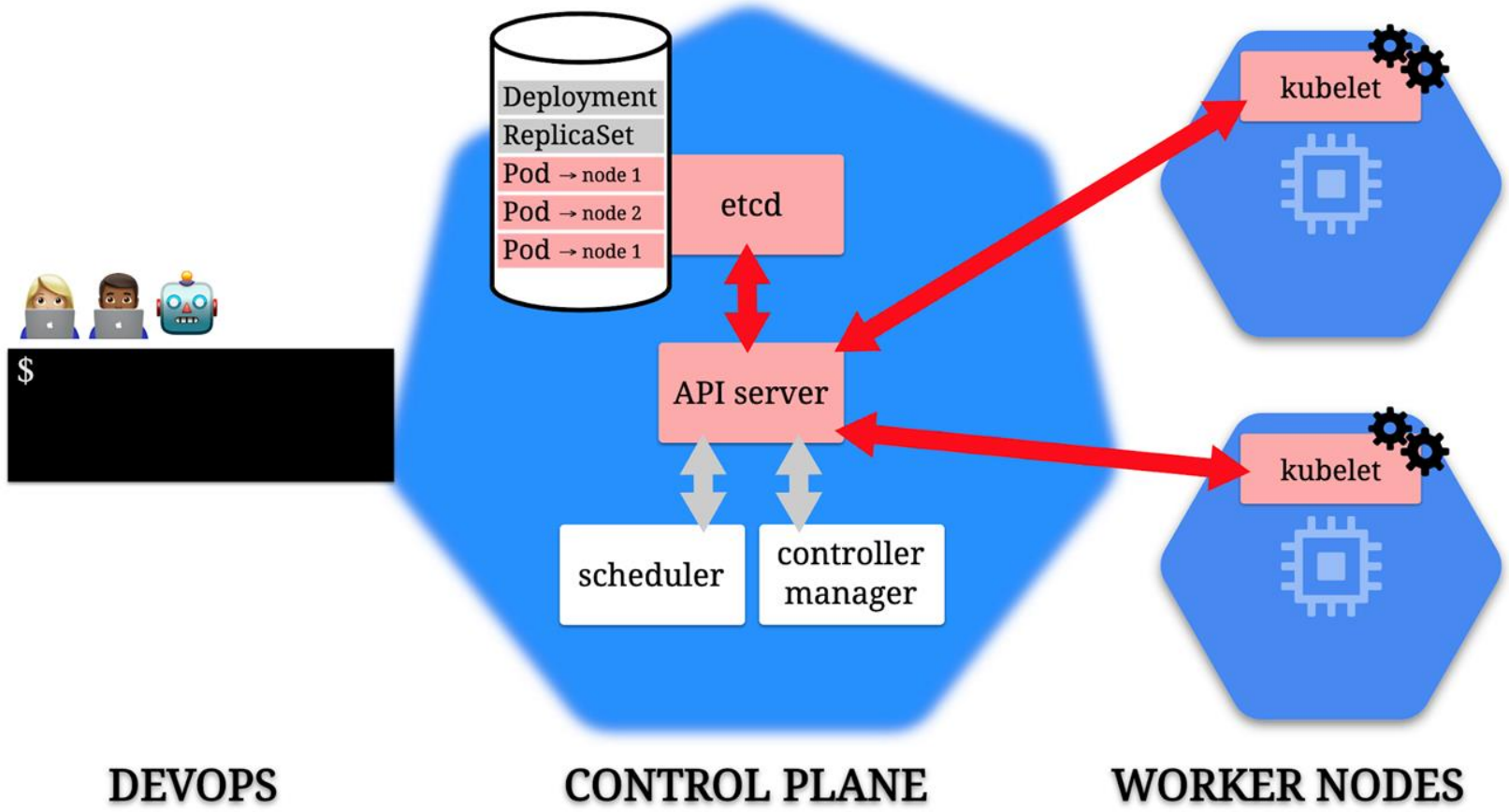

Deployment
ReplicaSet
Pod PENDING
Pod PENDING
Pod PENDING
etcd
API server
scheduler
controller manager
kubelet
kubelet
$
DEVOPS
CONTROL PLANE
WORKER NODES

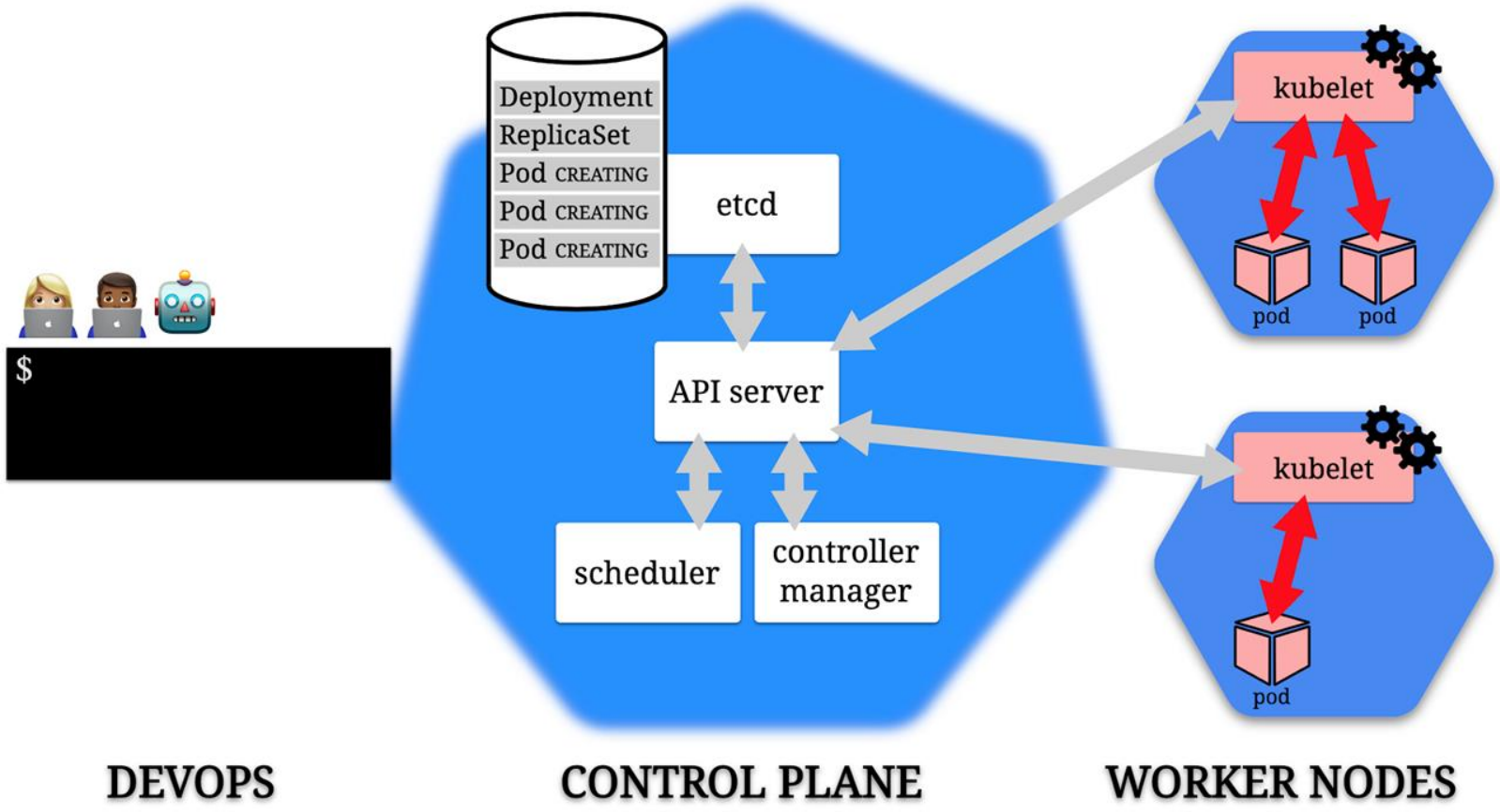

Deployment
ReplicaSet
Pod PENDING
Pod PENDING
Pod PENDING
etcd
API server
scheduler
controller manager
kubelet
kubelet
$
DEVOPS
CONTROL PLANE
WORKER NODES

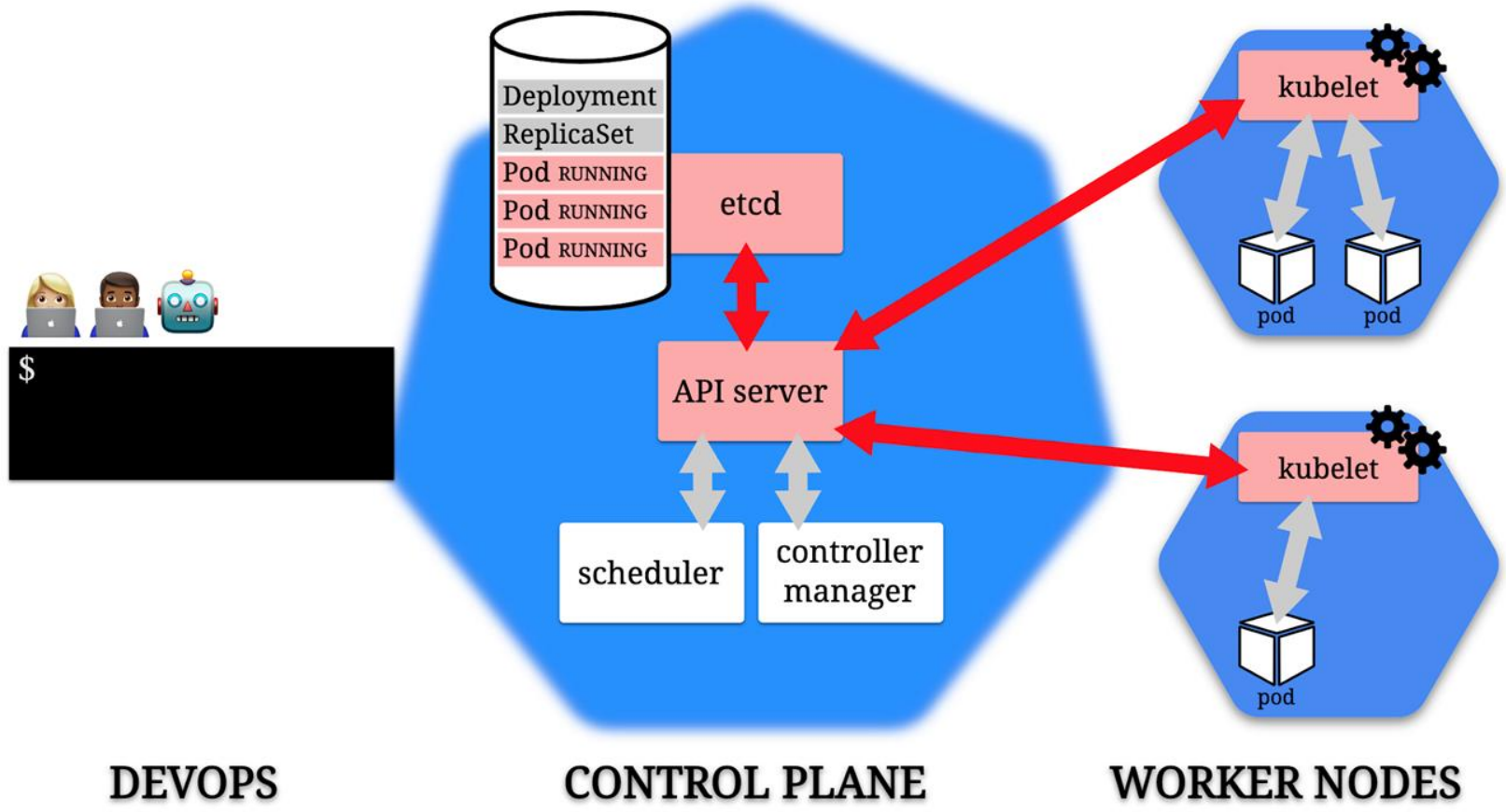

Deployment
ReplicaSet
Pod PENDING
Pod PENDING
Pod PENDING
etcd
API server
scheduler
controller manager
kubelet
kubelet
$
DEVOPS
CONTROL PLANE
WORKER NODES

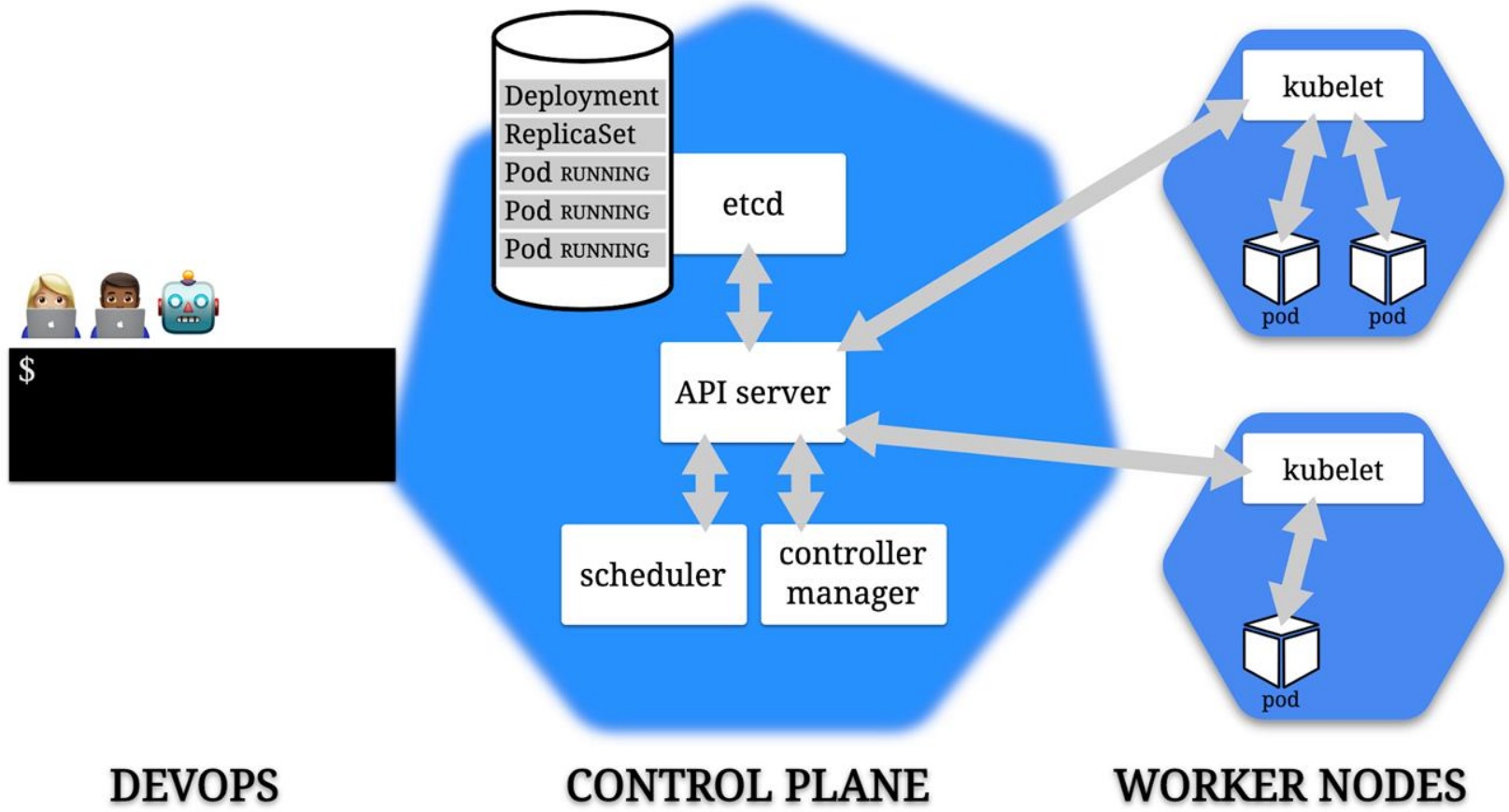

Options
Translate
Deployment
ReplicaSet
Pod → node 1
Pod → node 2
Pod → node 1
etcd
API server
scheduler
controller manager
kubelet
kubelet
$
DEVOPS
CONTROL PLANE
WORKER NODES


Deployment
ReplicaSet
Pod → node 1
Pod → node 2
Pod → node 1
etcd
API server
scheduler
controller manager
kubelet
kubelet
$
DEVOPS
CONTROL PLANE
WORKER NODES


Deployment
ReplicaSet
Pod → node 1
Pod → node 2
Pod → node 1
etcd
API server
scheduler
controller manager
kubelet
kubelet
$
DEVOPS
CONTROL PLANE
WORKER NODES


Deployment
ReplicaSet
Pod CREATING
Pod CREATING
Pod CREATING
etcd
API server
scheduler
controller manager
kubelet
pod
pod
kubelet
pod
$
DEVOPS
CONTROL PLANE
WORKER NODES


Deployment
ReplicaSet
Pod RUNNING
Pod RUNNING
Pod RUNNING
etcd
API server
scheduler
controller manager
kubelet
pod
pod
kubelet
pod
$
DEVOPS
CONTROL PLANE
WORKER NODES


Deployment
ReplicaSet
Pod RUNNING
Pod RUNNING
Pod RUNNING
etcd
API server
scheduler
controller manager
kubelet
pod
pod
kubelet
pod
$
DEVOPS
CONTROL PLANE
WORKER NODES

kubectl

- 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
- `*kubectl apply –f path-to-yaml-file.yml*`

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”

Pod state

- 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

Deployments

- 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

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

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.

ConfigMaps

- 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

ConfigMaps as Env Vars

- 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 .yml definition file (like all things k8s)

ConfigMaps

- 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 in Pods

- 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

k8s Services

- 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)

PersistentVolume

- 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

Demo

Continue Lab

Thank you!

If you have additional questions,
please reach out to me at:
(asanders@gamuttechnologysvcs.com)

DevelopIntelligence
A PLURALSIGHT COMPANY