Kubernetes Security Concepts

Daniel Hinojosa

Fast Guide to Securing your Cluster

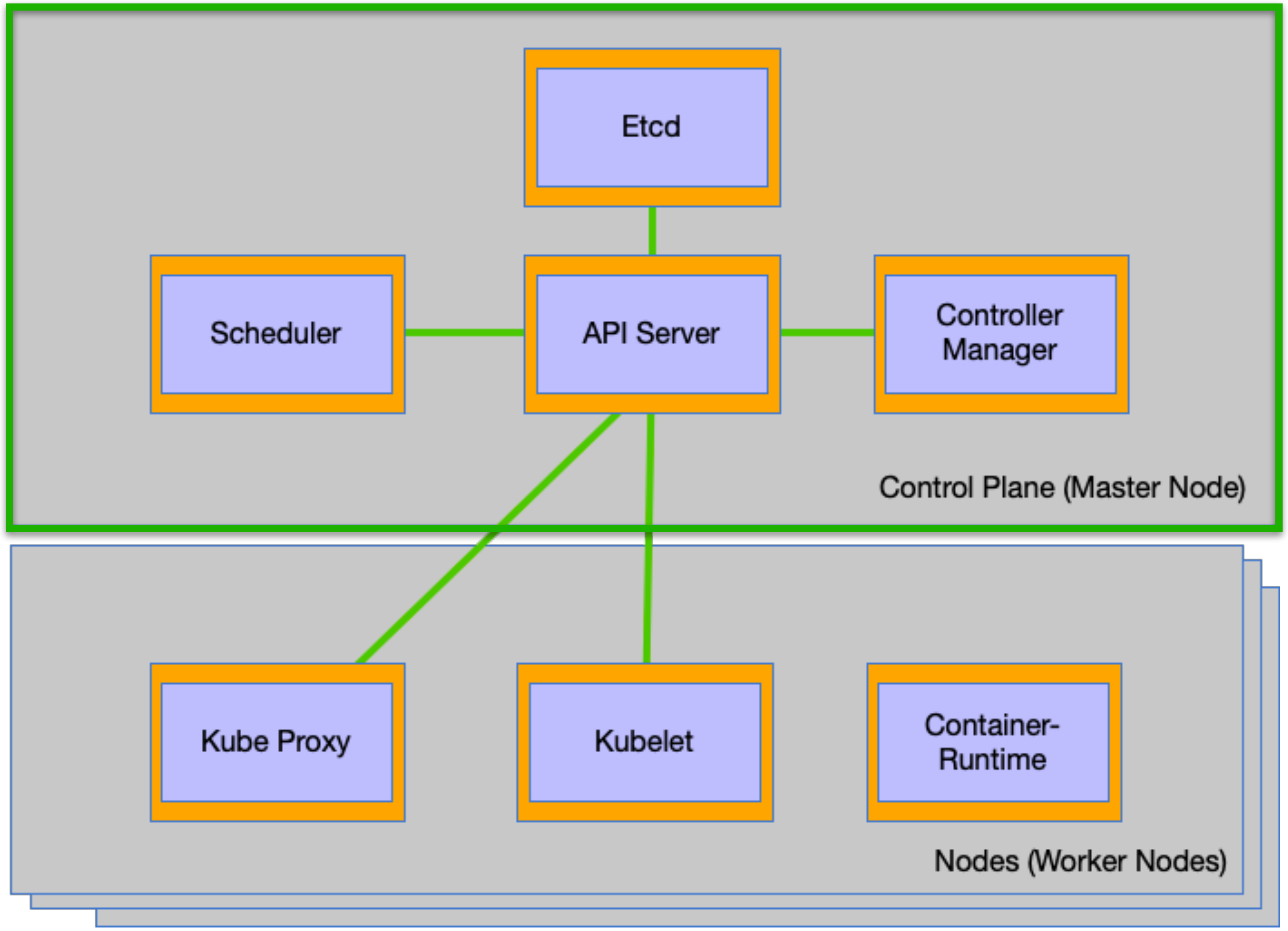
In this Presentation

- Review of Kubernetes Architecture
- Transport Layer Security (TLS)
  - mTLS within Kubernetes
  - Setting up TLS Termination
- Users and Roles
- Role Bindings and Cluster Bindings
- Authentication
- Kubernetes Network Policies

Slides and Material: https://github.com/dhinojosa/k8s-security-concepts

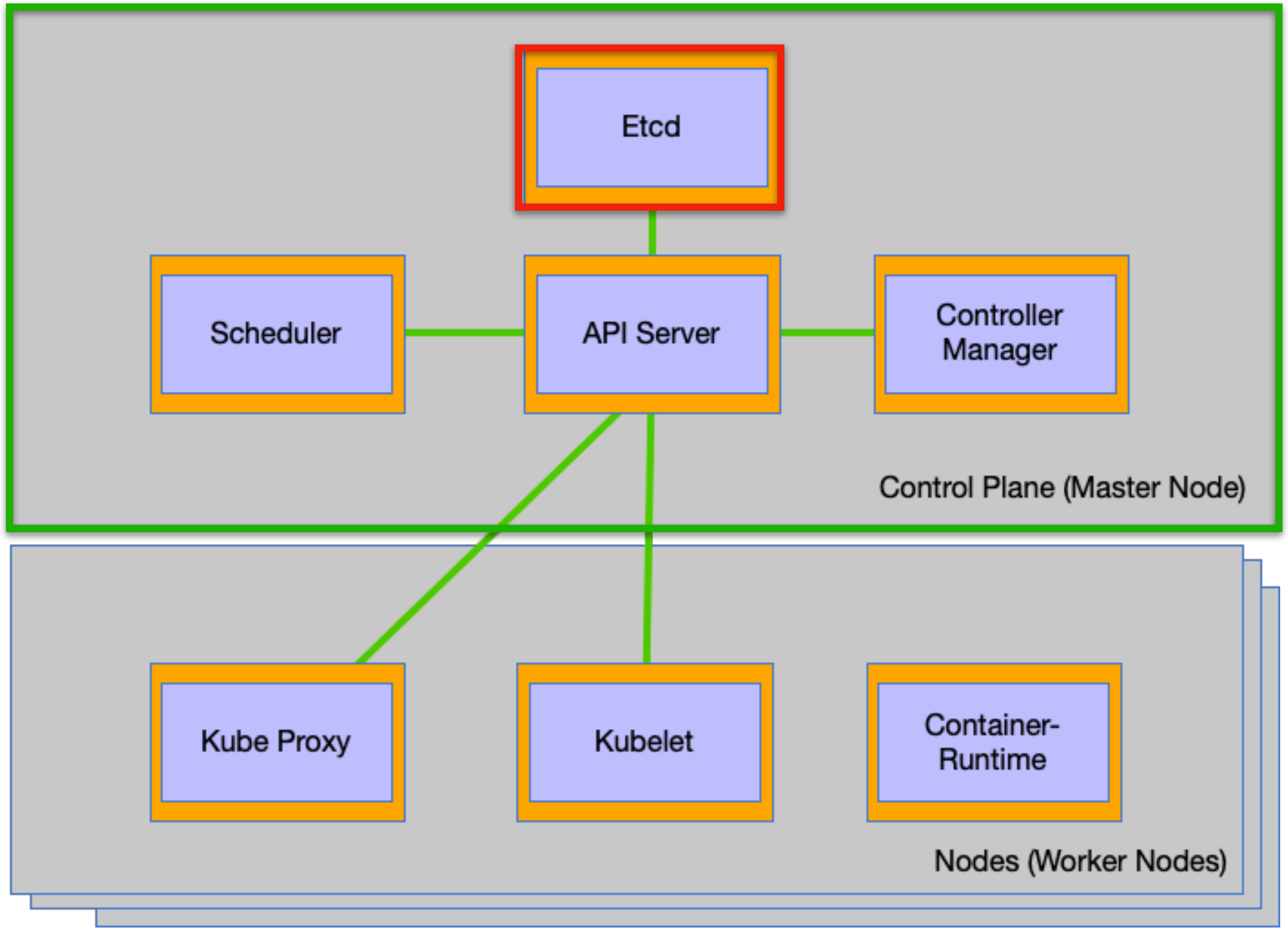
Architecture Overview

Control Plane



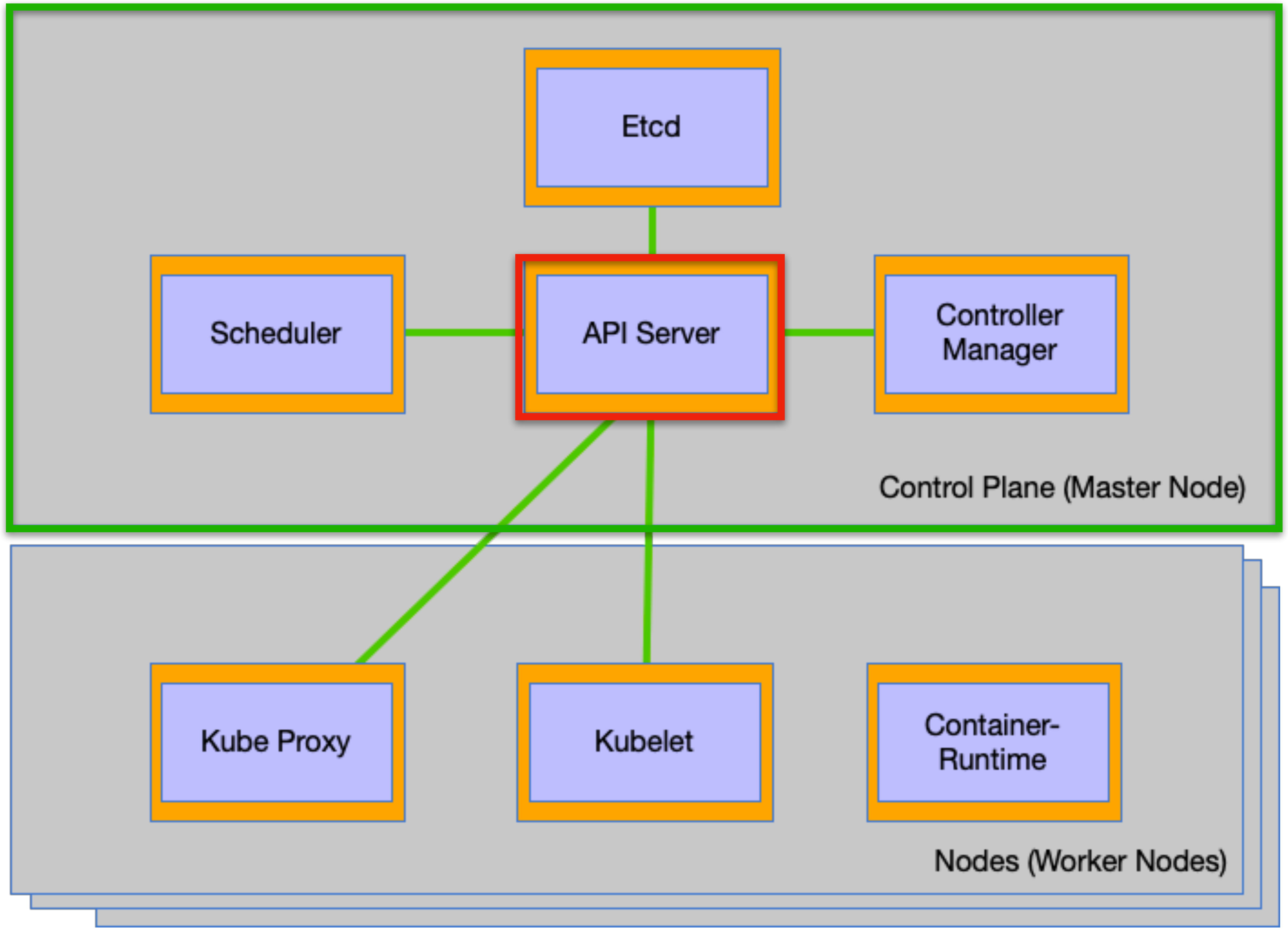
- Makes the whole cluster function
- Components included
  - Etcd Storage
  - API Server
  - Scheduler
  - Controller Manager

Etcd



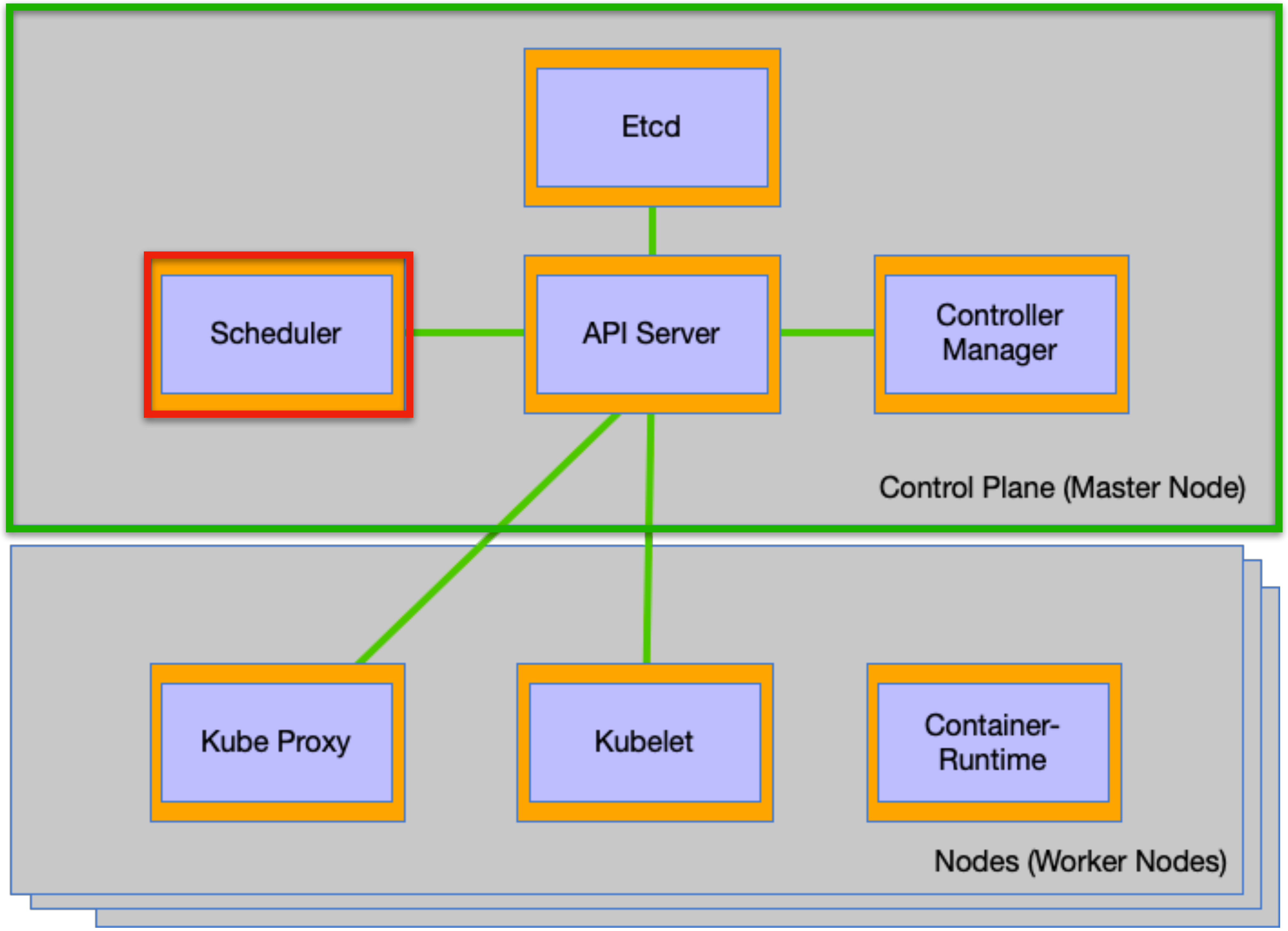
- Fast, Distributed, Key-Value Store
- Component Manifests are stored in etcd
- More than one instance can used for High Availability
- All read-writes are done through the API Server
- Only component that stores state and metadata

API Server



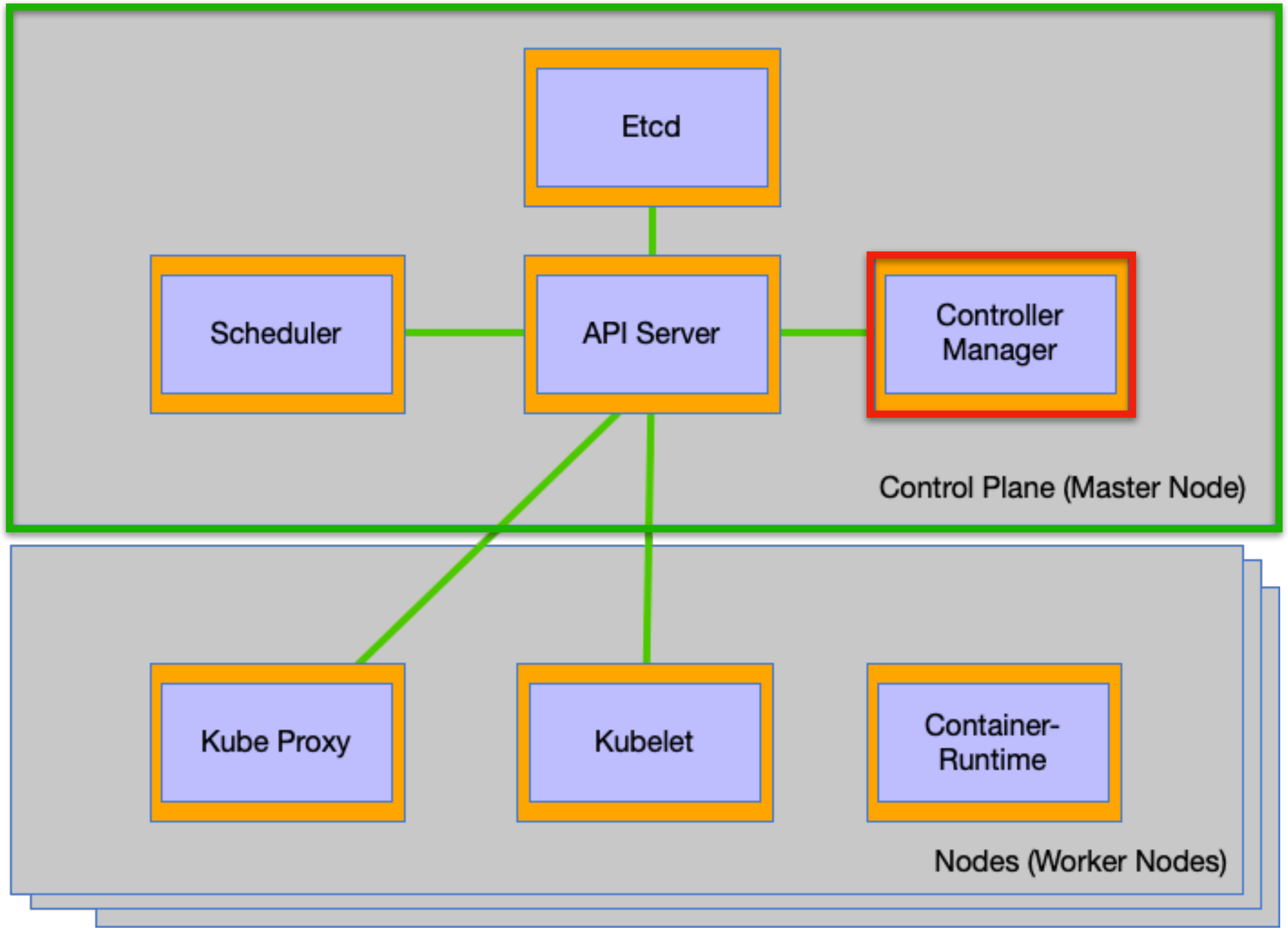
- RestFUL server used by other components
- All state to the API Server
- Embedded Validation: Components cannot store invalid key-value data
- Optimistic Locking: Changes to an object are not overridden
- Notify Clients of their change

Scheduler



- Wait for Pod Events from the API Server as a Watch
- Assign a Node to Each Pod
- But it does not run the Pod
- It merely schedules the pod to run on its node
- It does so by sending the pod schedule to the the API Server

Controller Manager

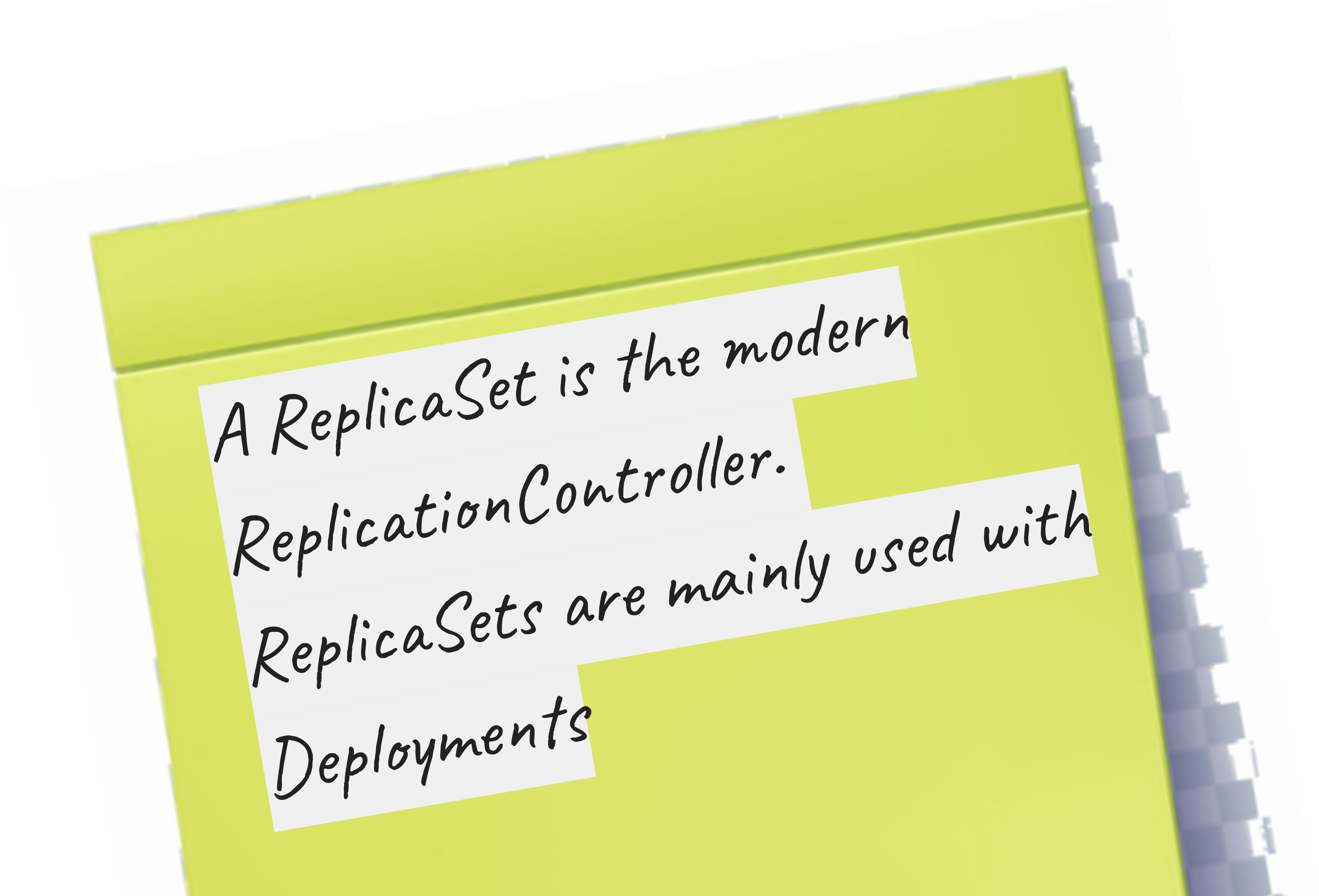


- Component that ensures that the state of the system converges towards the correct state
- Combines multiple *controllers* to perform reconciliation
- Controllers are spawned in multiple processes
- Has the ability to replace with a custom controller

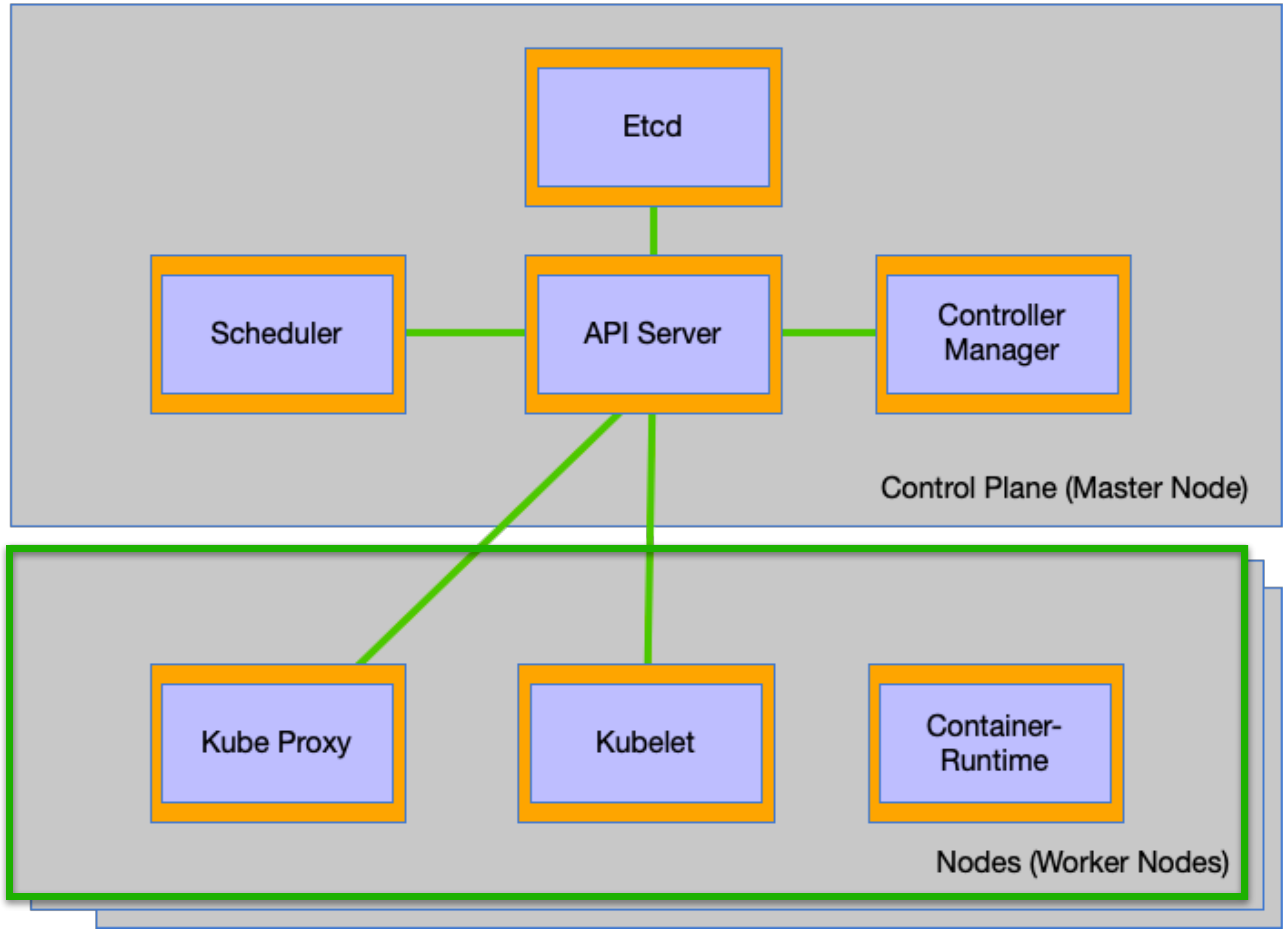
List of Varying Controllers

- Service Controller
- Persistent Volume Controller
- Replica Set Controller
- Daemon Set Controller
- Job Controller
- Stateful Set Controller
- Node Controller

- Service Controller
- Endpoints Controller
- Namespace Controller
- Replication Controller Controller

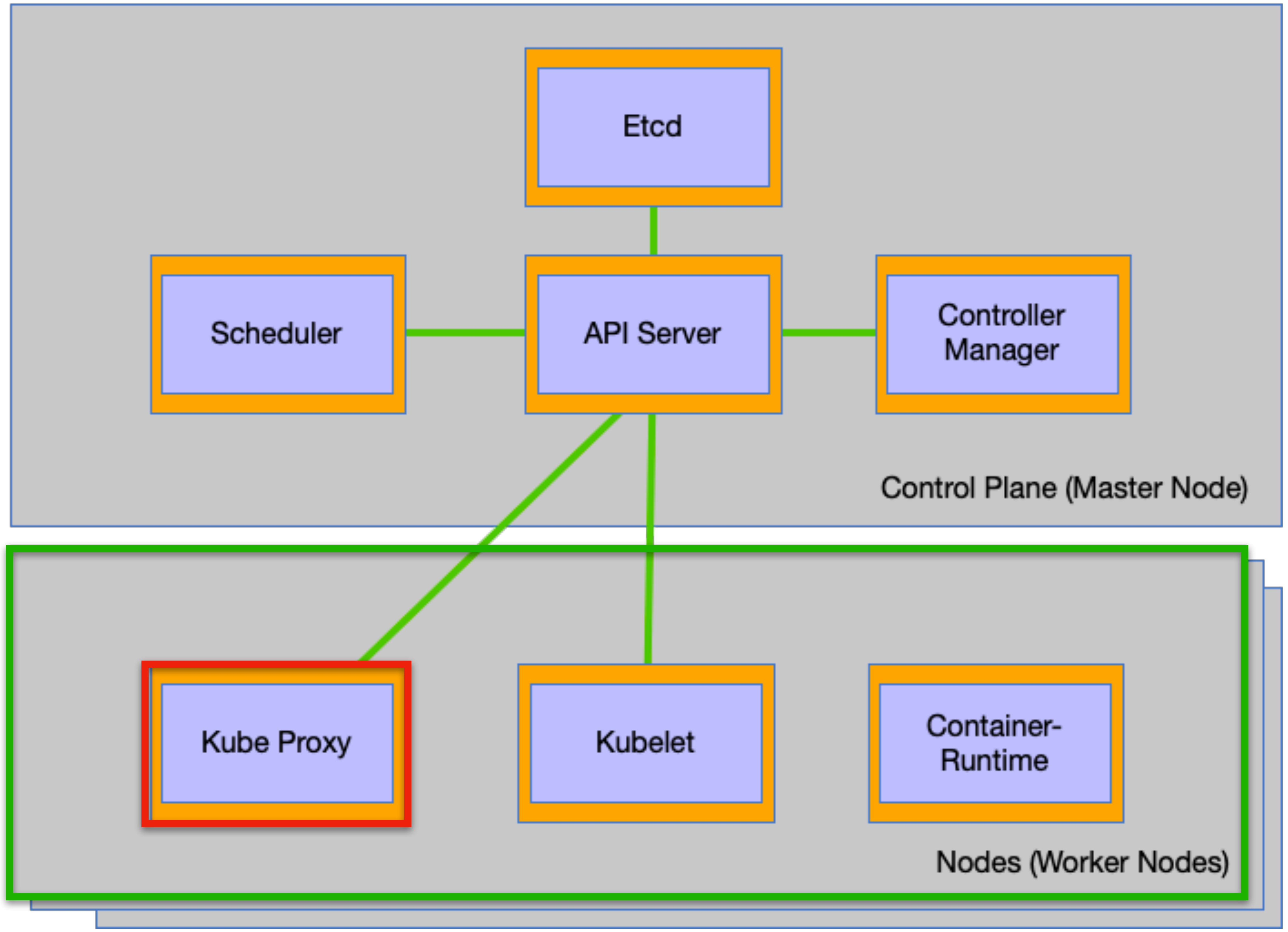


Node



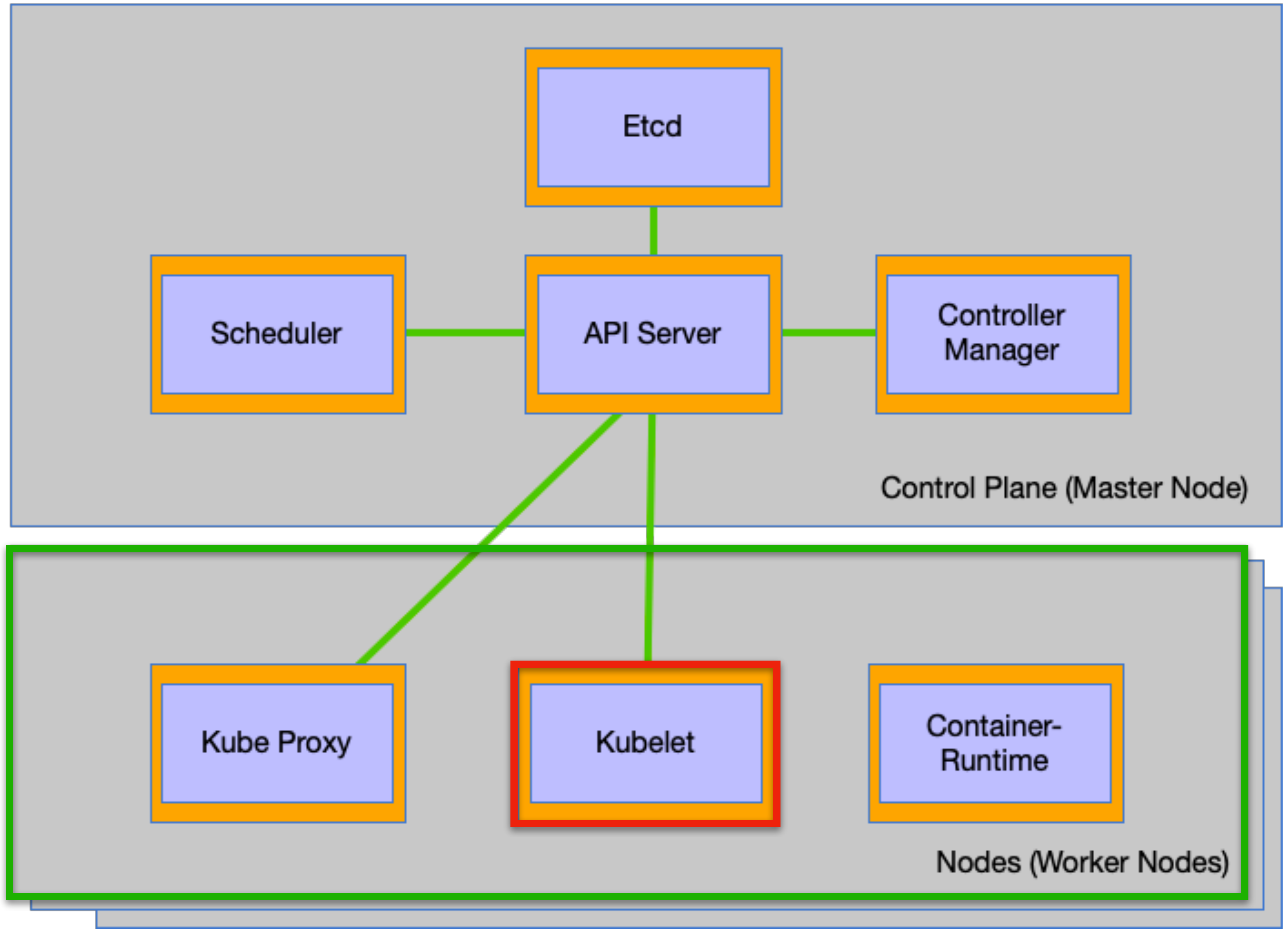
- A node may be a virtual or physical machine
- Each node contains the services necessary to run Pods
- Contains
  - Kube Proxy
  - Kubelet
  - Container Runtime

Kube Proxy



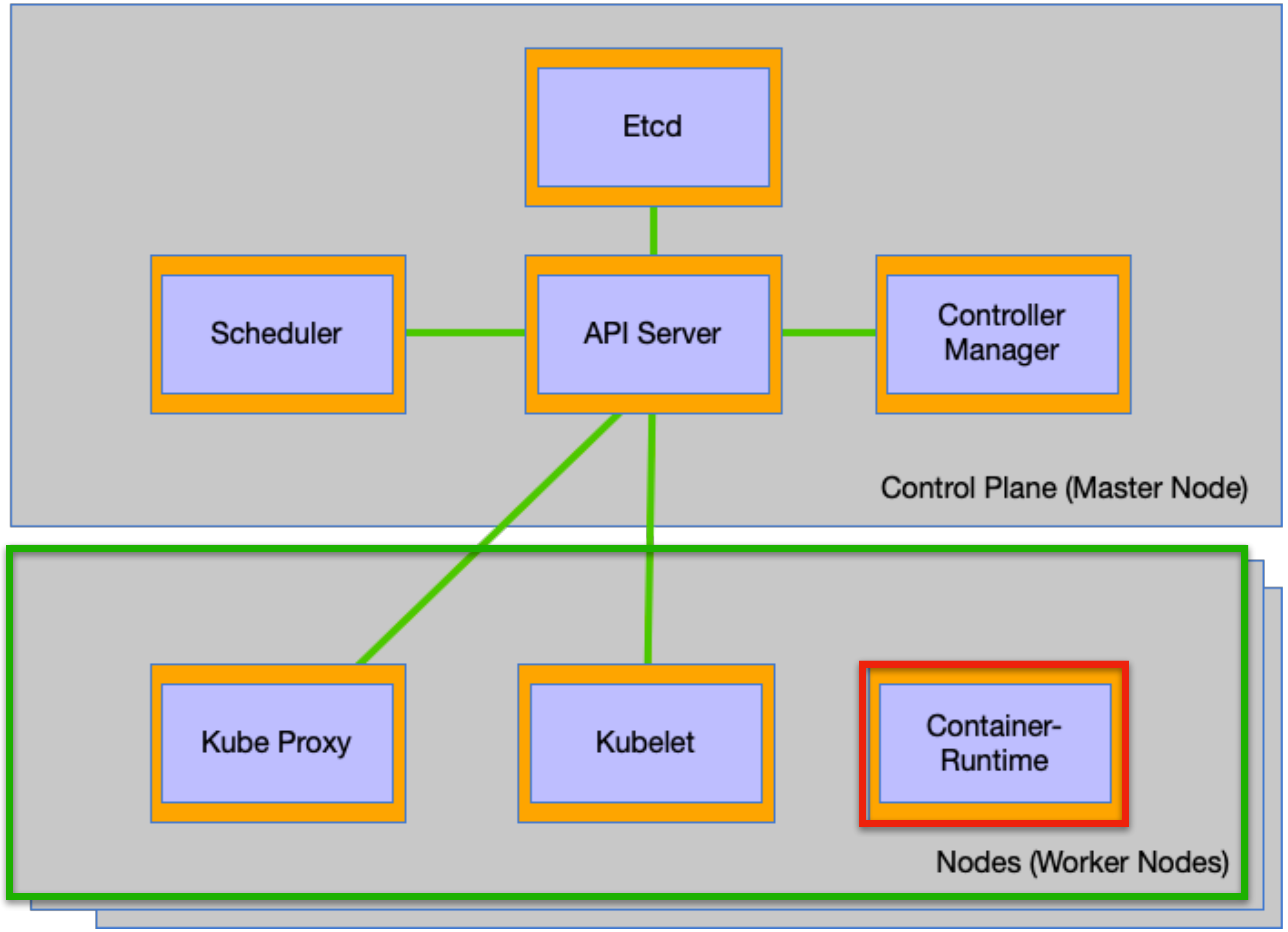
- Network Proxy that runs in each node
- Just like performing https://localhost

Kubelet



- An agent that runs on each node in the cluster.
- Makes sure that containers are running in a Pod
- API Server connects to the Kubelet when fetching logs, attaching, or port-forwarding

Container Runtime



- Software responsible for running the containers
- Supports varying container runtimes:
  - Docker
  - containerd
  - CRI-O
  - Any Implementation of the Kubernetes CRI (Container Runtime Interface)

Understanding your Kube Config

KubeConfig

- Default file: `.kube/config`
- File that maintains and organizes information about clusters, users, namespaces, and authentication mechanisms
- What is used by `kubectl` to find the information it needs to choose a cluster and communicate with the API server of a cluster
- Can be overridden with the `KUBECONFIG` environment variable
- Can use your own config file with kubectl using `--kubeconfig` flag
- Users and their certificates are also maintained within the `.kube/config`
- `kubectl config view` will show the active configuration

Viewing KubeConfig Demo

Showing a KubeConfig Samples

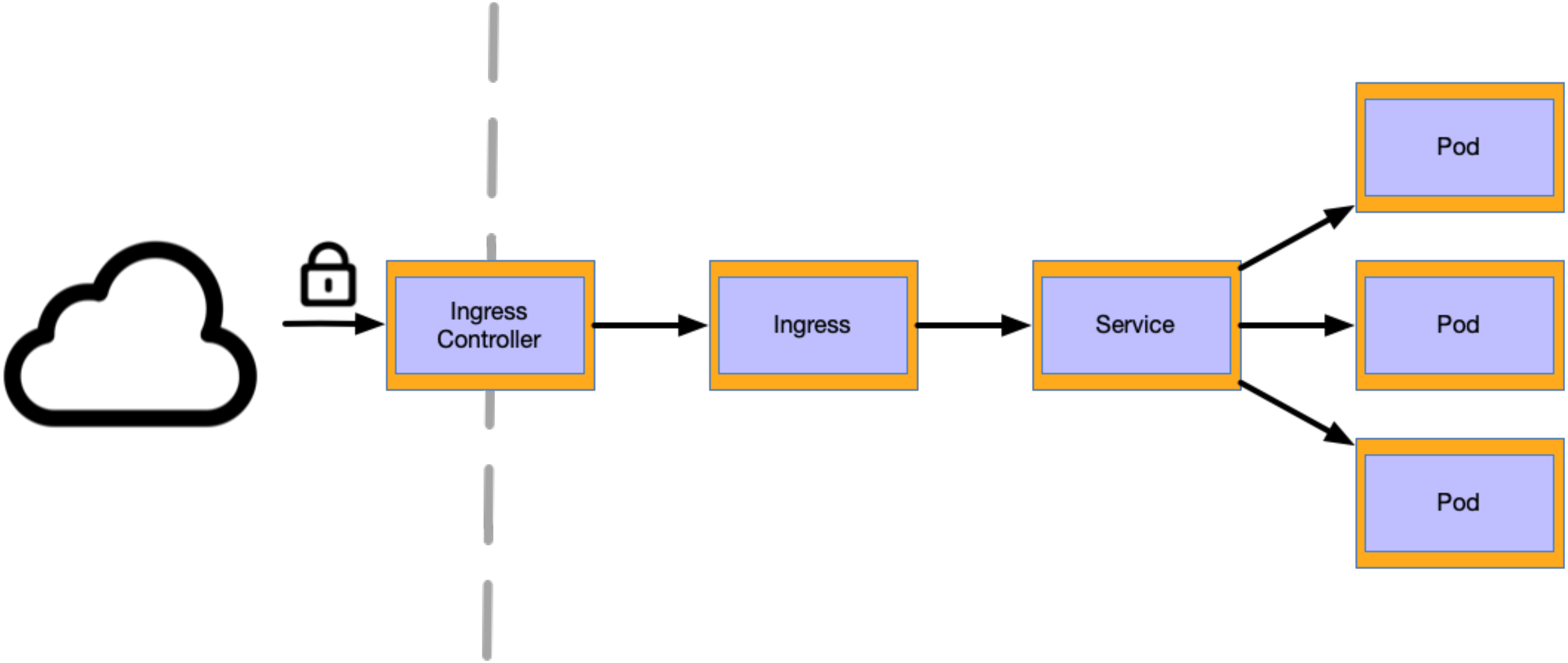
Kubernetes Objects Overview

TLS Termination Ingress

Ingress Controllers

- Ingress Controller's a load balancing service that routes traffic to services within Kubernetes.
- Different Kubernetes environments use different implementations of the controller, but several don't provide a default controller at all
- A popular option would be to use nginx (perhaps via Helm)
- Ingress Controllers can also perform TLS-Termination, meaning that TLS traffic terminates at the controller

```
$ helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
$ helm install ingress-nginx ingress-nginx/ingress-nginx
```

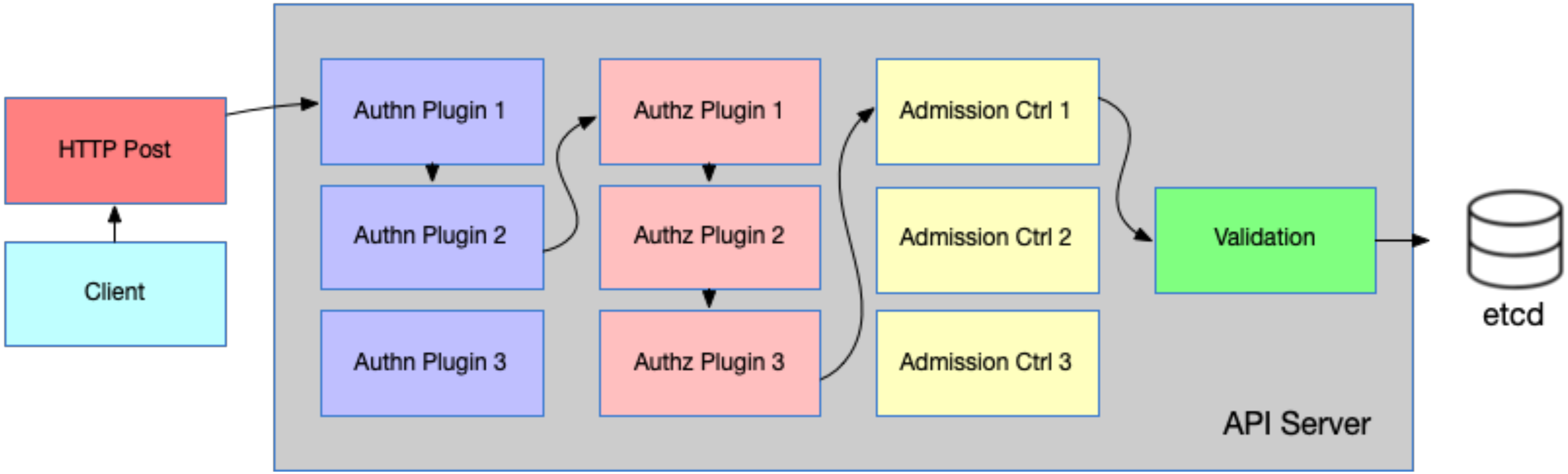


Ingress Controller
Ingress
Service
Pod
Pod
Pod

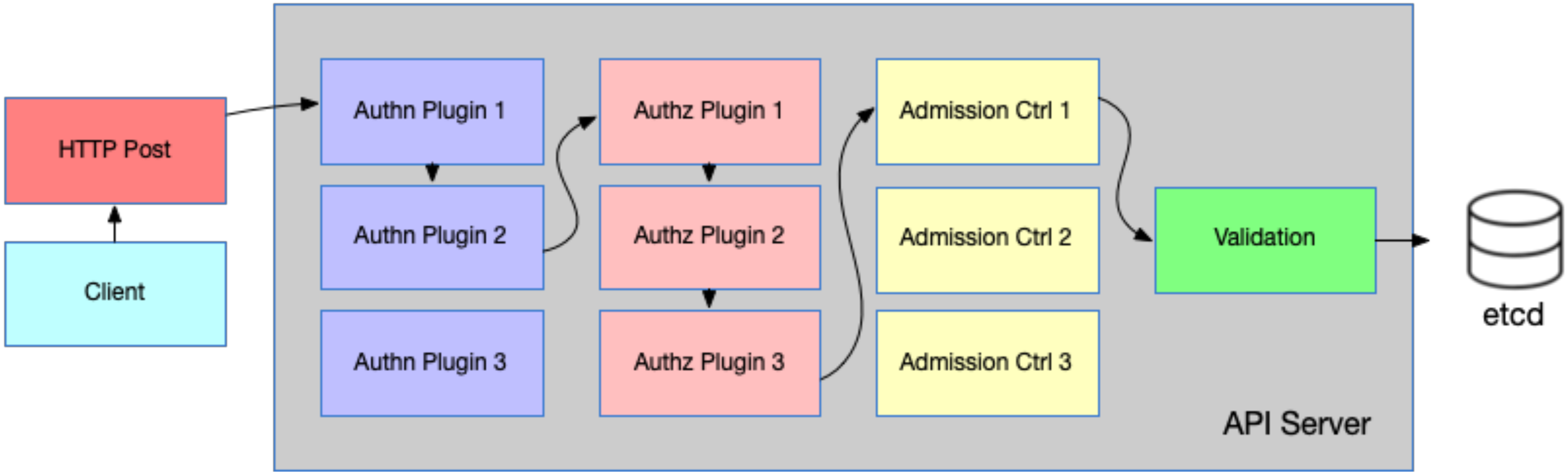
TLS Termination Ingress

Applying an Ingress and Certificate in AWS

Kubernetes Authn & AuthZ



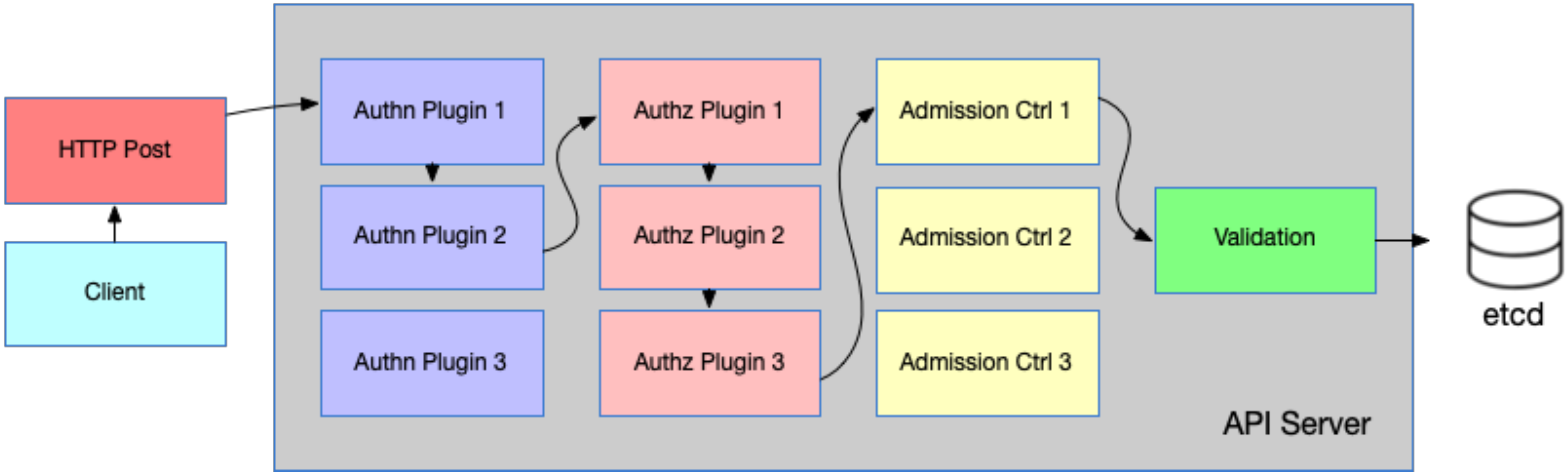
- Client call attempt is made as an HTTP call to the API Server, this includes what we do with `kubectl`
- API Server is RESTful: `get`, `post`, `put`, `patch`, etc.
- The API Authenticates, Authorized, Admit, and Validates before changing etcd



- Authentication is a list of the Authenticated Plugins that identifies who is gaining access to the Kubernetes API
- Authentication Plugins works like a linked list where one will be able to authenticate
- Once established then Authorization Plugins are negotiated

Authentication Plugins

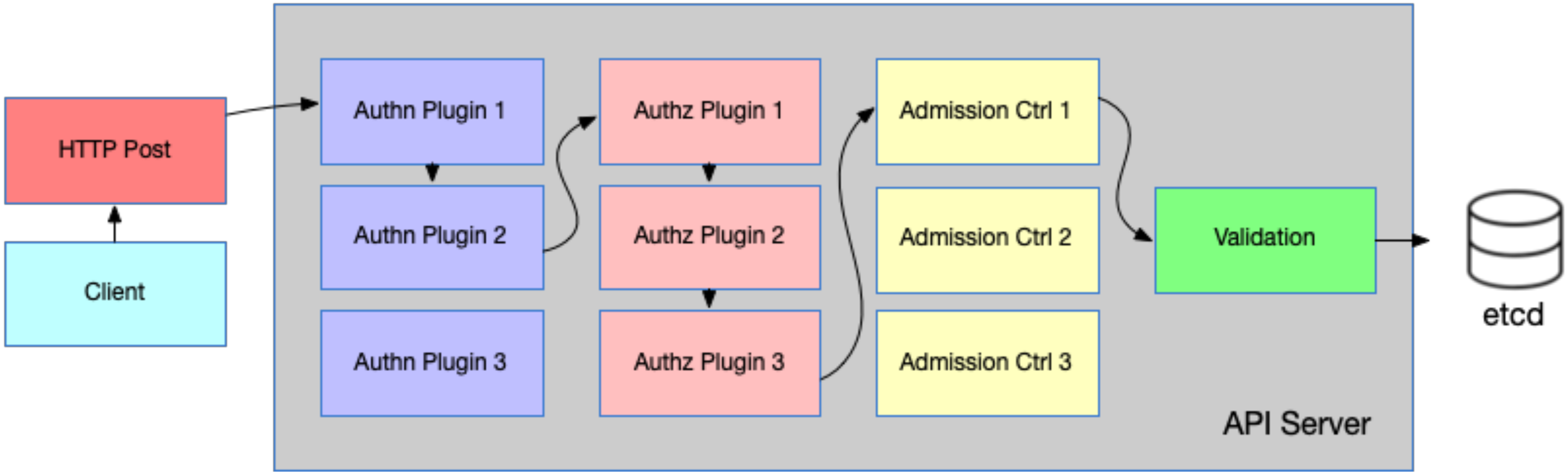
- X509 Client Certs
- Static Token File
- Bearer Tokens
- Bootstrap Tokens
- Service Account Tokens
- Open ID Connect Tokens
- Webhook Token Authentication



- Authorization Plugins works like a linked list where they have plugins
- Given the action that the user wishes to perform, the plugins will determine if the user is allowed to do so.
- As soon as a plugin says the user can perform the action, the API server progresses to the next stage

Authorization Plugins

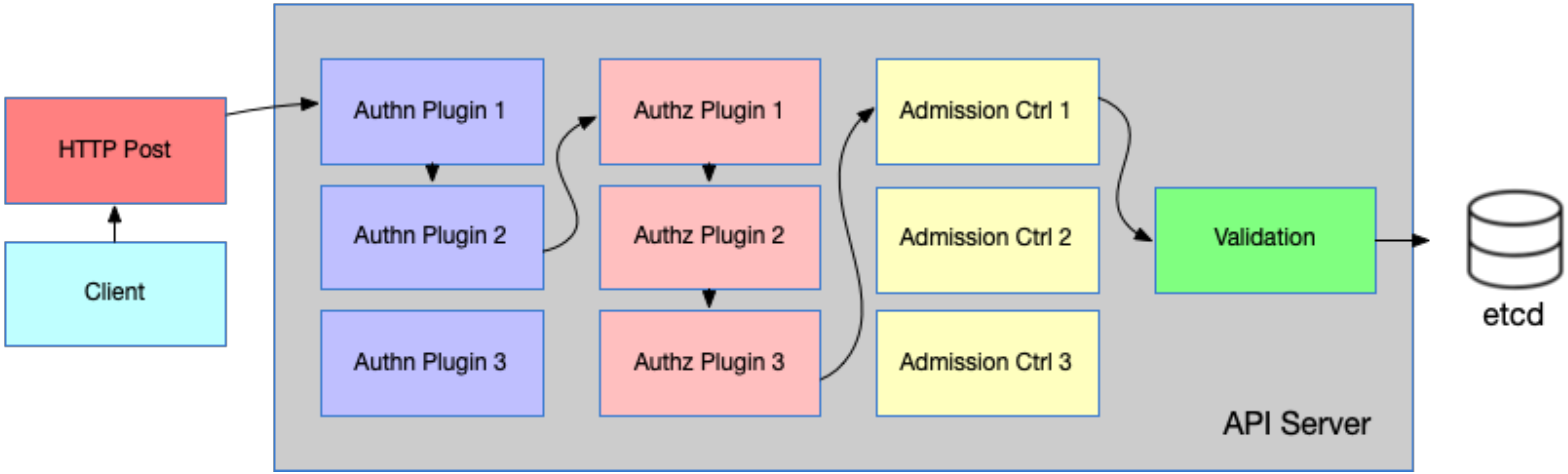
- RBAC Plugin - Role Based Access Control, checks whether an action is allowed to be performed by the user requesting the action
- ABAC Plugin - Attribute Based Access Control, defines an access control paradigm whereby access rights are granted to users through the use of policies which combine attributes together
- Node Plugin - Node authorization is a special-purpose authorization mode that specifically authorizes API requests made by kubelets
- Webhook Plugin - WebHook is an HTTP callback mode that allows you to manage authorization using a remote REST endpoint



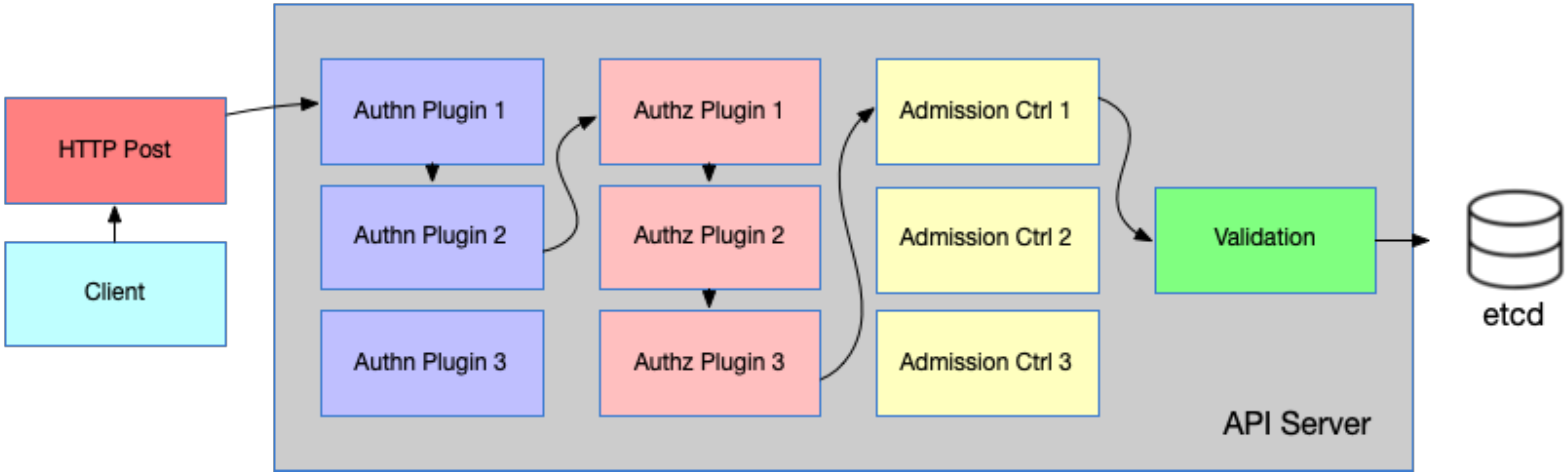
- Admission Control plugins can modify the resource for different reasons
- They may initialize fields missing from the resource specification to the configured default values or even override them
- They may even modify other related resources, which aren't in the request, and can also reject a request for whatever reason
- GET (Read) calls don't go through the admission control plugins

Admission Control Plugins

- `AlwaysPullImages`—Overrides the pod's `imagePullPolicy` to `Always`, forcing the image to be pulled every time the pod is deployed.
- `ServiceAccount`—Applies the default service account to pods that don't specify it explicitly.
- `NamespaceLifecycle`—Prevents creation of pods in namespaces that are in the process of being deleted, as well as in non-existing namespaces.
- `ResourceQuota`—Ensures pods in a certain namespace only use as much CPU and memory as has been allotted to the namespace.



- Admission Control plugins can modify the resource for different reasons
- They may initialize fields missing from the resource specification to the configured default values or even override them
- They may even modify other related resources, which aren't in the request, and can also reject a request for whatever reason
- GET (Read) calls don't go through the admission control plugins



- API server then validates the object, stores it in etcd, and returns a response to the client.
- Validation ensures that the data structure is correct before placing into etcd

Service Accounts

Kubernetes Types Of Accounts

- Humans
    - Meant to be Authenticated by External System
- Pods
    - Meant to be Authenticated by Service Account
    - Stored in the API Server as a Resource

Service Accounts

- API Server authorizes requests coming from a pod based on a Service Account
- All pods by default are associated with the default Service Account
- Service Accounts
  - Can be associated with a namespace or cluster
  - Bound to Roles with a Role Binding
- Token Stored in each pod:
  `/var/run/secrets/kubernetes.io/serviceaccount/token`

Service Account Demo

Gaining Access to the Kubernetes API using Service Accounts

User and Groups

Users aren't your Standard Users

- There is no such user database in Kubernetes
- Users can be managed outside of Kubernetes (e.g. LDAP, SSO)
- No API Calls to add to Users
- Groups are also not defined in Kubernetes this is established as an outside concern

Kubernetes Groups

- Users and Service Accounts belong to one or more Groups
- Authentication Plugins returns groups with username and user ID
- Groups are used to grant permission to many people at once
- Kubernetes has automatically created group that are returned with the following formats:
  - `system:unauthenticated` - unauthenticated clients
  - `system:authenticated` - user authenticated successfully
  - `system:serviceaccounts` - all service accounts
  - `system:serviceaccounts:<namespace>` - service accounts in a namespace

Creating Users & Groups Demo

Establish a new user to access your Kubernetes Cluster

RBAC

RBAC

- Role Based Access Control
- General Availability as of Kubernetes 1.8
- Prevent unauthorized users from viewing or modifying the cluster state via the Kubernetes API
- Uses user roles as the key factor in determining whether the user may perform an action
- A *subject* (users, service account, or groups thereof) is associated with one or more roles
- A role is allowed to perform certain verbs (`GET`, `POST`, `PUT`, `PATCH`, `DELETE`)

Http Methods and Verbs

What do they mean when consulting the API

HTTP Method	Verb Single	Verb Plural
GET, HEAD	get (and watch)	list (and watch)
POST	create	
PUT	update	
PATCH	path	
DELETE	delete	deletecollection

https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.19/#-strong-read-operations-pod-v1-core-strong-

Role Bindings

Role

- Always sets permissions within a particular namespace
- Must add a namespace that it belongs
- Roles define what can be done

Defining a Role

Establishing a Role within Kubernetes

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: data-engineering
  name: pod-reader
rules:
- apiGroups: [""] # Core API
  verbs: ["get", "list"]
  resources: ["pods"]
```

https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.19/#-strong-read-operations-pod-v1-core-strong-

Defining a Role Binding to Service Account

Establishing a RoleBinding within Kubernetes

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: test
  namespace: foo
  ...
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: service-reader
subjects:
- kind: ServiceAccount
  name: default
  namespace: foo
```

https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.19/#-strong-read-operations-pod-v1-core-strong-

Role Binding Demo

Allow Users and Service Account a little more free reign within your Kubernetes Cluster

Cluster Bindings

ClusterRole

- Role that represents a non-namespaced resources
- Resources that are not namespaced includes Nodes, PersistentVolumes, and Namespaces
- The Kubernetes API can expose some URLs that don't represent resources, like the `/healthz` endpoint
- Namespaced resources (like Pods), across all namespaces For example: you can use a ClusterRole to allow a particular user to run kubectl get pods --all-namespaces

ClusterRole YAML Example

ClusterRoles have no namespaces

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  # "namespace" omitted since ClusterRoles are not namespaced
  name: secret-reader
rules:
- apiGroups: [""]
  #
  # at the HTTP level, the name of the resource for accessing Secret
  # objects is "secrets"
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

ClusterRoleBindings

- Binds together the cluster role with subject (User, Group, ServiceAccount)
- Kubernetes comes with a default set of ClusterRoles and ClusterRoleBindings
- Cluster Role Bindings Updated everytime the API starts

ClusterRoleBinding YAML Example

ClusterRoleBinding to either a User, Group, or ServiceAccount

```yaml
apiVersion: rbac.authorization.k8s.io/v1
# This cluster role binding allows anyone in the "manager" group to read secrets in any namespace.
kind: ClusterRoleBinding
metadata:
  name: read-secrets-global
subjects:
- kind: Group
  name: manager # Name is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```

Cluster Role Binding Demo

Give a user access to do more to your Kubernetes Cluster

Securing Pods

Networking Policies

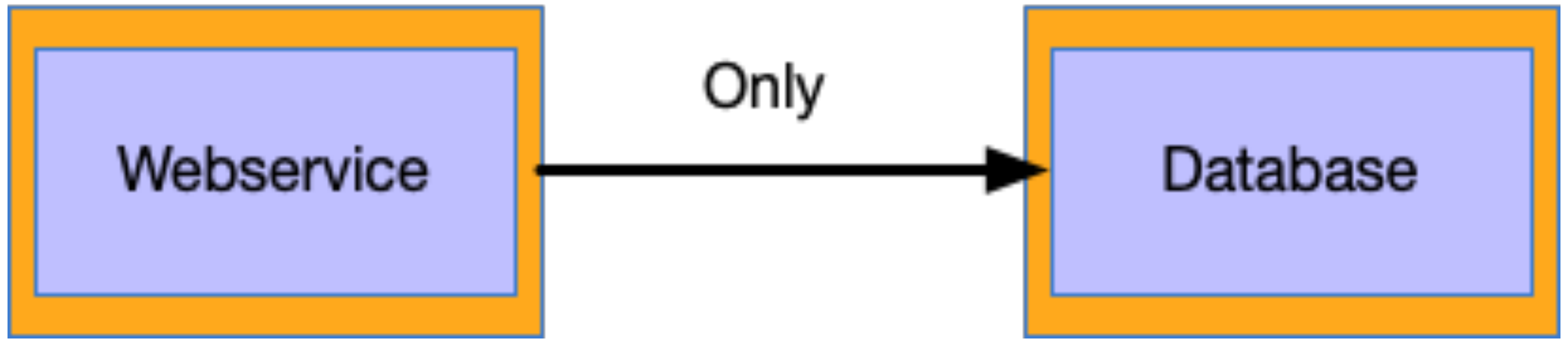
Network Policies

- NetworkPolicy applies to pods that match its label selector
- Specifies either which sources can access the matched pods or which destinations can be accessed from the matched pods
- Operated with ingress or egress **rules**. Ingress here has nothing to do with the ingress as we talked about previously
- Matching pods can be performed using:
  - A pod selector
  - A namespace selector
  - CIDR Notation (`192.168.1.1/24`)

Network Policy by Pod Selector

Selecting the traffic allowed inwards, ingress

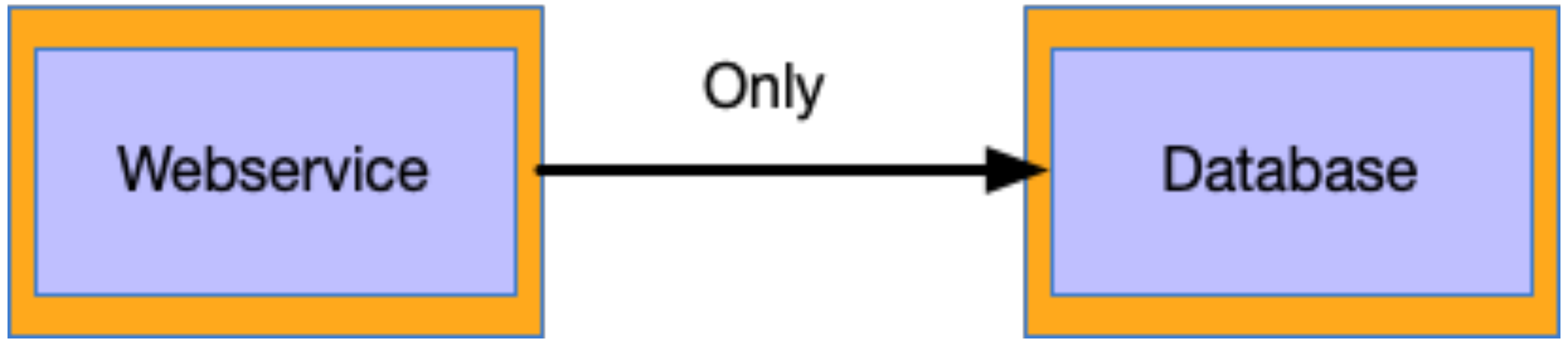
```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: database-netpolicy
spec:
  podSelector:
    matchLabels:
      app: database
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: webserver
    ports:
    - port: 3066
```



Network Policy by Pod Selector

Selecting the traffic allowed outwards, egress

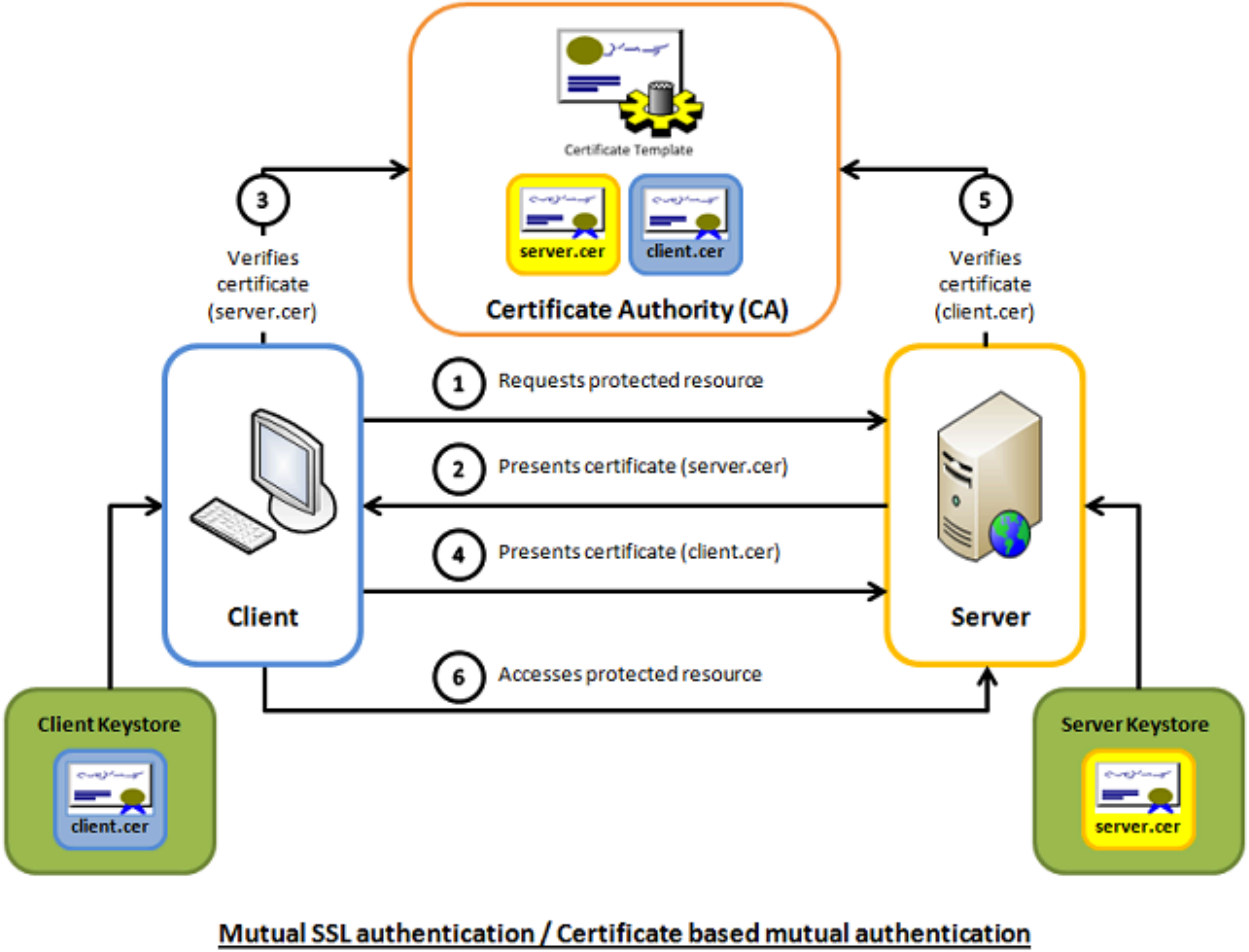
```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: webserver-netpolicy
spec:
  podSelector:
    matchLabels:
      app: webserver
  egress:
  - from:
    - podSelector:
        matchLabels:
          app: database
```



Outer Network mTLS

mTLS Defined

Mutual TLS (mTLS) - Authentication ensures that traffic is both secure and trusted **in both directions** between a client and server.



Mutual Authentication flow from [codeproject.com]

Step for mTLS

- Setup an Ingress Controller
- Create Certificates
- Create Kubernetes Secrets
- Deploy your Application

https://awkwardferny.medium.com/configuring-certificate-based-mutual-authentication-with-kubernetes-ingress-nginx-20e7e38fdfca

Creating the Certificates

- **CommonName(CN):** Identifies the hostname or owner associated with the certificate.
- **Certificate Authority(CA):** A trusted 3rd party that issues Certificates. Usually you would obtain this from a trusted source, but for this example we will just create one. The CN is usually the name of the issuer.
- **Server Certificate:** A Certificate used to identify the server. The CN here is the hostname of the server. The Server Certificate is valid only if it is installed on a server where the hostname matches the CN.
- **Client Certificate:** A Certificate used to identify a client/user. The CN here is usually the name of the client/user.

https://awkwardferny.medium.com/configuring-certificate-based-mutual-authentication-with-kubernetes-ingress-nginx-20e7e38fdfca

Creating the mTLS Certificates

```
# Generate the CA Key and Certificate
$ openssl req -x509 -sha256 -newkey rsa:4096 -keyout ca.key -out ca.crt -days 356 -nodes
-subj '/CN=Fern Cert Authority'
# Generate the Server Key, and Certificate and Sign with the CA Certificate
$ openssl req -new -newkey rsa:4096 -keyout server.key -out server.csr -nodes -subj '/
CN=meow.com'
$ openssl x509 -req -sha256 -days 365 -in server.csr -CA ca.crt -CAkey ca.key -set_serial
01 -out server.crt
# Generate the Client Key, and Certificate and Sign with the CA Certificate
$ openssl req -new -newkey rsa:4096 -keyout client.key -out client.csr -nodes -subj '/
CN=Fern'
$ openssl x509 -req -sha256 -days 365 -in client.csr -CA ca.crt -CAkey ca.key -set_serial
02 -out client.crt
```

Storing CA and Server CRT

```
$ kubectl create secret generic my-certs --from-file=tls.crt=server.crt --from-file=tls.key=server.key --from-file=ca.crt=ca.crt

$ kubectl get secret my-certs
NAME          TYPE      DATA    AGE
my-certs      Opaque    3       1m
```

- Deploy your application as normal with Services and Deployments
- We will configure the Ingress to use my-certs as to where the certificate authority and server certificates will reside

Establishing Ingress Certificates

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    nginx.ingress.kubernetes.io/auth-tls-verify-client: \"on\"
    nginx.ingress.kubernetes.io/auth-tls-secret: \"default/my-certs\"
  name: meow-ingress
  namespace: default
```

- TLS is enabled and it is using the `tls.key` and `tls.crt` provided in the my-certs secret.
- The `nginx.ingress.kubernetes.io/auth-tls-secret` annotation uses `ca.crt` from the `my-certs` secret.

https://awkwardferny.medium.com/configuring-certificate-based-mutual-authentication-with-kubernetes-ingress-nginx-20e7e38fdfca

Testing mTLS

```
$ curl https://meow.com/ -k
...
<center><h1>400 Bad Request</h1></center>
<center>No required SSL certificate was sent</center>
....
```

- -k is insecure, don't consult with a certificate verification
- In the following, a client certifcation and client key is used to get the payload

```
$ curl https://meow.com/ --cert client.crt --key client.key -k
...
ssl-client-issuer-dn=CN=Fern Cert Authority
ssl-client-subject-dn=CN=Fern
ssl-client-verify=SUCCESS
user-agent=curl/7.54.0
...
```

https://awkwardferny.medium.com/configuring-certificate-based-mutual-authentication-with-kubernetes-ingress-nginx-20e7e38fdfca

Inner Network TLS

TLS The Hard Way

```
cat <<EOF | cfssl genkey - | cfssljson -bare server
{
  "hosts": [
    "my-svc.my-namespace.svc.cluster.local",
    "my-pod.my-namespace.pod.cluster.local",
    "192.0.2.24",
    "10.0.34.2"
  ],
  "CN": "system:node:my-pod.my-namespace.pod.cluster.local",
  "key": {
    "algo": "ecdsa",
    "size": 256
  },
  "names": [
    {
      "O": "system:nodes"
    }
  ]
}
EOF
```

Create Certificate Signing Request

```
cat <<EOF | kubectl apply -f -
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: my-svc.my-namespace
spec:
  request: $(cat server.csr | base64 | tr -d '\n')
  signerName: kubernetes.io/kubelet-serving
  usages:
  - digital signature
  - key encipherment
  - server auth
EOF
```

Creating the Secret

```
apiVersion: "v1"
kind: "Secret"
metadata:
  name: "nginxsecret"
  namespace: "default"
type: kubernetes.io/tls
data:
  tls.crt: "LS0tL..."
  tls.key: "LS0tL..."
```

- Apply the TLS.crt and TLS.key as a Kubernetes Secret
- This can be used when defining the Service

Creating the Secret

```
spec:
  volumes:
  - name: secret-volume
    secret:
      secretName: nginxsecret
    ...
    containers:
    - name: nginxhttps
      image: bprashanth/nginxhttps:1.0
      ports:
      ...
      volumeMounts:
      - mountPath: /etc/nginx/ssl
        name: secret-volume
```

- Create a Secret Volume which refers to the Secret established in the previous slide
- Bind the SSL to the directory where in this case NGINX is requiring the certificates

Istio/Service Mesh and mTLS

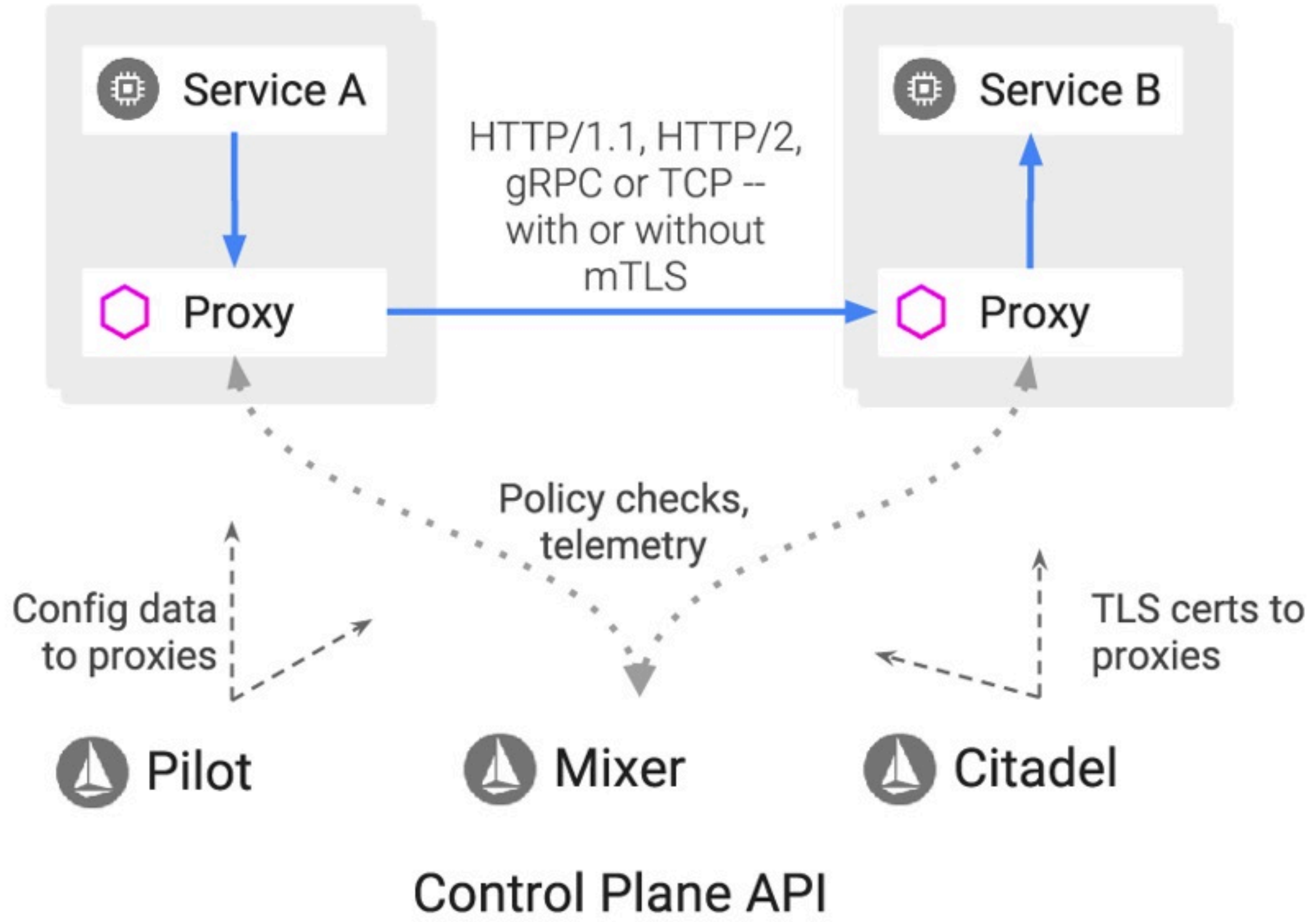
Service Meshes

- A service mesh manages all service-to-service communication within a distributed (potentially microservice-based) software system
- Use of “sidecar” proxies that are deployed alongside each service through which all traffic is transparently routed.
- OSI Layer 7 = Communication using HTTP, now any underlying layer like packets TCP, etc
- Dynamic service discovery and traffic management
- Traffic Shadowing for Testing, Traffic Spitting for Canary
- Including but not limited to **Security Enforcement**!
- Linkerd, Istio, Consul, Kuma, Maesh, AWS App Mesh

OSI Layer Model

7	Application Layer	Human-computer interaction layer, where applications can access the network services	HTTP, FTP, IRC, SSH, DNS	
6	Presentation Layer	Ensures that data is in a usable format and is where data encryption occurs	SSL, SSH,IMAP, FTP	Data Layer
5	Session Layer	Maintains connections and is responsible for controlling ports and sessions	Sockets, Winsock	
4	Transport Layer	Transmits data using transmission protocols including TCP and UDP	TCP, UDP	Segment Layer
3	Network Layer	Decides which physical path the data will take	IP, ICMP, IPSec	Packet Layer
2	Data Link Layer	Defines the format of data on the network	Frames, Ethernet, PPP	Frame Layer
1	Physical Layer	Transmits raw bit stream over the physical medium	Coax, Fiber, Wireless	Bit Layer

Istio


Service A
Service B
HTTP/1.1, HTTP/2, gRPC or TCP -- with or without mTLS
Proxy
Proxy
Policy checks, telemetry
Config data to proxies
TLS certs to proxies
Pilot
Mixer
Citadel
Control Plane API

Istio Components

- **Pilot** - Responsible for configuring the Envoy and Mixer at runtime.
- **Proxy / Envoy** - Sidecar proxies per microservice to handle ingress/egress traffic between services in the cluster and from a service to external services.
- **Mixer** - Create a portability layer on top of infrastructure backends. Enforce policies such as ACLs, rate limits, quotas, authentication, request tracing and telemetry collection at an infrastructure level.
- **Citadel / Istio CA -** Secures service to service communication over TLS. Providing a key management system to automate key and certificate generation, distribution, rotation, and revocation.
- **Ingress/Egress** - Configure path based routing for inbound and outbound external traffic.
- **Control Plane API** - Underlying Orchestrator such as Kubernetes or Hashicorp Nomad.

Mutual TLS Authentication

- TLS Communication and Setup performed through Envoy Proxies
- How Istio handles that traffic:
    - Istio re-routes the outbound traffic from a client to the client's local sidecar Envoy.
    - The client side Envoy starts a mutual TLS handshake with the server side Envoy. During the handshake, the client side Envoy also does a secure naming check to verify that the service account presented in the server certificate is authorized to run the target service.
    - The client side Envoy and the server side Envoy establish a mutual TLS connection, and Istio forwards the traffic from the client side Envoy to the server side Envoy.
    - After authorization, the server side Envoy forwards the traffic to the server service through local TCP connections.

https://istio.io/latest/docs/concepts/security/

Thank You

- Email: dhinojosa@evolutionnext.com
- Github: https://www.github.com/dhinojosa
- Twitter: http://twitter.com/dhinojosa
- Linked In: http://www.linkedin.com/in/dhevolutionnext