Advanced Kubernetes

CONCEPTS, MANAGEMENT, MIDDLEWARE

Agenda

Intro / Prep Environments

Day 1: Docker Deep Dive

Day 2: Kubernetes Deep Dive

Day 3: Advanced Kubernetes: Concepts, Management, Middleware

Recap

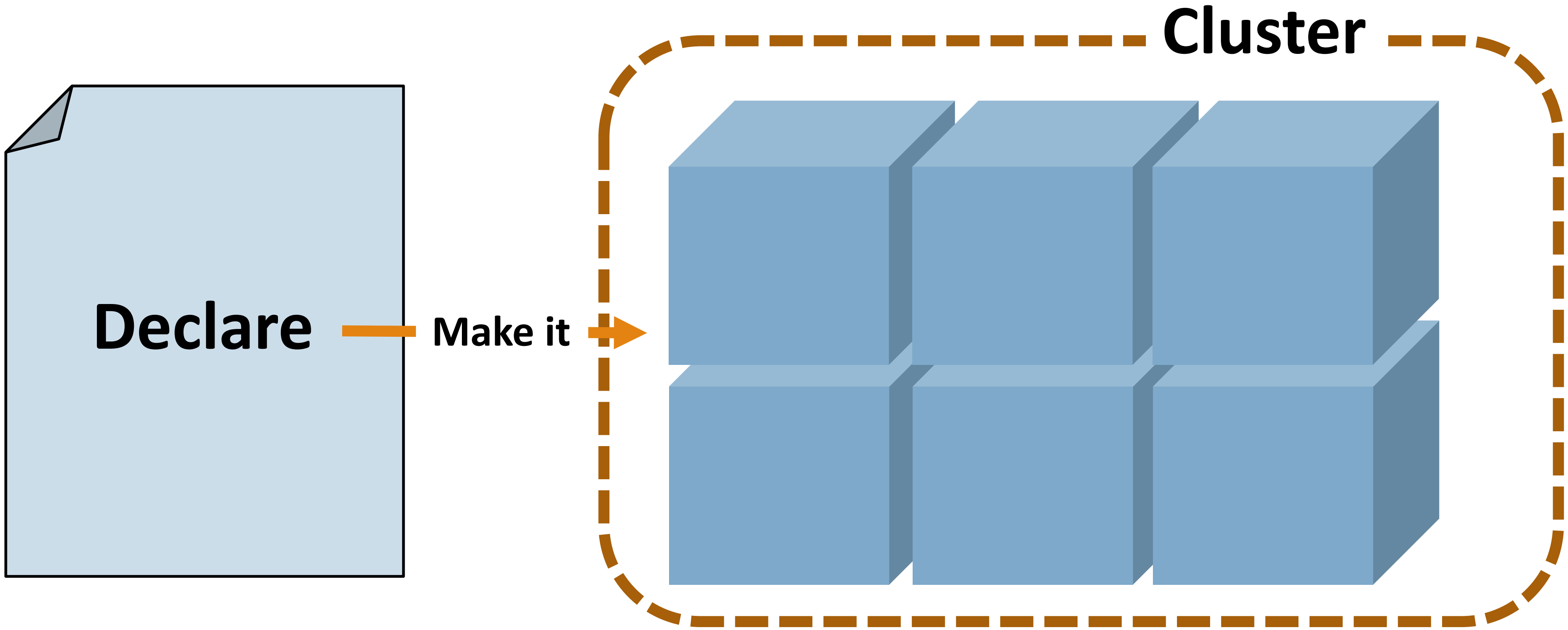
Recap Docker

- Containers run on **single** Docker host
- Containers are **ephemeral**
- Nothing watchdogs the containers
- Containers can have external persistence
- Containers do not contain
- Operating system **matters**

Why is it Important

- Managing containers by hand is harder than VMS - which won’t scale
- Automate the boilerplate stuff
- Runbooks → Scripts → Config Management → Scale
- Decouple application from machine
- Applications run on “resources"
- Kubernetes manages this interaction of applications and resources
- Manage applications and not machines
- What about **legacy apps?**

Reconciliation of End State



Kubernetes Core Concepts

- Simplicity, Simplicity, Simplicity
- Pods
- Labels / Selectors
- Replication Controllers
- Services
- API

Why You Win with Docker and Kubernetes

- Immutable infrastructure
- DevOps
- CI/CD

Kubernetes Namespaces

- Divide cluster across uses, tiers, and teams
- Unique within a namespace; not across multiple namespaces
- Very powerful when combined with Labels
- Example: `qa/dev/prod` can be implemented with Namespaces

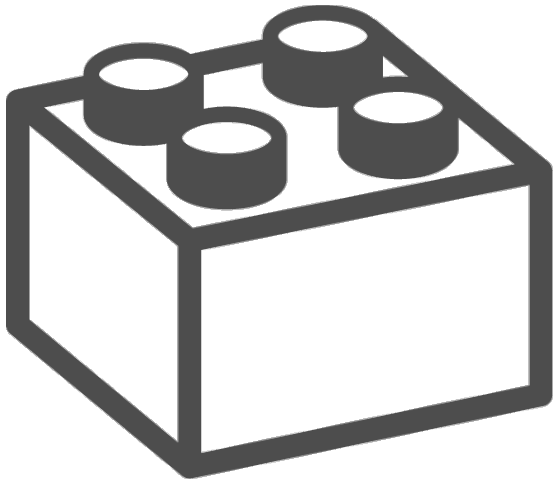
Kubernetes Namespaces

- List the namespaces available to the cluster:

```
kubectl get namespaces
```

- List all the pods across all the namespaces

```
kubectl get pods --all-namespaces
```



Kubernetes Namespaces

- Let’s create a new namespace for our `guestbook` application:

```
 curl -s -L https://raw.githubusercontent.com/christian-
posta/docker-kubernetes-
workshop/master/demos/guestbook/namespace.yaml | kubectl
```

- Let’s list the pods in the `guestbook` namespace:

```
kubectl get pods --namespace=guestbook
```

  - NOTE: There shouldn’t be any at the moment.

Kubernetes Contexts / Namespaces

- You can log into multiple Kubernetes clusters with the same client and switch between clusters/contexts at the command line.
- You can also specify which namespaces to use when pointing to specific clusters.

- For example, to view the current cluster context:

```
kubectl config view
```

Kubernetes Contexts / Namespaces

- **Output:**

```
- context:
    cluster: master-fuse-osecloud-com:8443
    namespace: microservice
    user: admin/master-fuse-osecloud-com:8443
  name: microservice/master-fuse-osecloud-com:8443/admin
- context:
    cluster: vagrant
    user: vagrant
  name: vagrant
current-context: vagrant
kind: Config
preferences: {}
users:
- name: admin/master-fuse-osecloud-com:8443
  user:
    token: kZ_L5Oj5sJ8nJUVJD4quq813Q1pRv4yZWhOjuJEw79w
- name: vagrant
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
    password: vagrant
    username: vagrant
```

Setting and Using Context/Namespaces

- We can create a new context that points to our vagrant cluster:

```
kubectl config set-context guestbook --
namespace=guestbook --user=vagrant --cluster=vagrant
```

- Now, let's switch to use that context so we can put any new pods/RCs into this new namespace:

```
kubectl config  use-context guestbook
```

- Now double check we're in the new context/namespace:

```
kubectl config view | grep current-context | awk '{print
```

Setting and Using Context/Namespaces

- Now let's deploy a replication controller:

```
curl -s -L https://raw.githubusercontent.com/christian-posta/docker-kubernetes-workshop/master/demos/guestbook/frontend-controller.yaml | kubectl create -f -
```

- Now, let's see how many pods we have:

```
kubectl get pods
```

- Output:

```
NAME              READY      STATUS     RESTARTS    AGE
frontend-juz6j    0/1        Pending    0           5s
```

Removing Components Namespaces

- We have two good ways to group components for development purposes and then clean them up when you want to start over.
  - Kubernetes labels
  - Namespaces
- You can delete all resources in a namespace like this:

```
kubectl config use-context vagrant
```

```
kubectl delete namespace guestbook
```

Removing Components Labels

- The Namespace approach works fine for local development and grouping.
- However, in shared environments the best approach is to properly label your components (services, RCs, pods, etc) and delete them using labels.
- That would look something like this:

```
kubectl delete all -l "label=value"
```

Not all Objects in a Namespace

- Most objects are in a namespace
  - Pods
  - Replication controllers
  - Services
- Namespaces themselves not in namespace
- Nodes, PersistentVolumes

Resource Quotas

- If the API Server has `ResourceQuota` passed to the `kube-apiserver`'s `--admission_control` argument, then a namespace can set a **ResourceQuota** object to limit resources.
- Example from the vagrant/master:

```
root       6055  0.0  0.0   3172    48 ?        Ss   00:04   0:00 /bin/sh -c
/usr/local/bin/kube-apiserver --address=127.0.0.1 --
etcd_servers=http://127.0.0.1:4001 --cloud_provider=vagrant  --
runtime_config=api/v1 --
admission_control=NamespaceLifecycle,NamespaceExists,LimitRanger,SecurityCon
textDeny,ServiceAccount,ResourceQuota --service-cluster-ip-
range=10.247.0.0/16 --client_ca_file=/srv/kubernetes/ca.crt --
basic_auth_file=/srv/kubernetes/basic_auth.csv  --cluster_name=kubernetes --
tls_cert_file=/srv/kubernetes/server.cert --
tls_private_key_file=/srv/kubernetes/server.key --secure_port=443 --
token_auth_file=/srv/kubernetes/known_tokens.csv --bind-address=10.245.1.2 -
-v=2   --allow_privileged=False 1>>/var/log/kube-apiserver.log 2>&1
```

Resource Quotas

- Pods must use Resource Limits or will fail to accept the Pod
  - Can use a **LimitRange** to add default limits
- Admin creates a ResourceQuota for the namespace
- If a Pod would cause the Resource Limits to breach, the pod is rejected
- If the aggregate Resource Limits are set higher than actual available resources, first-come first-serve

Use Labels … for Nodes Too!

- You can organize your Nodes based on classifications, tiers, and resource types.
- For example, for some data-intensive applications you may wish to request that the scheduler put those pods on nodes that have SSD storage/PV support:

```
kubectl label nodes node-foo disktype=ssd
```

Use Labels … for Nodes Too!

- Now if you add a node selector section to your Pod, the pod will only end up on nodes with the `disktype=ssd` label.
- Example:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

Kubernetes: Security

Security Goals

- Appropriate boundaries between cluster, pods, users who manage cluster/application developers
- Appropriate boundaries enforced between containers and hosts
  - via `docker/linux cap/selinux/apparmor/etc`
- Ability to delegate administrative functions to users where it makes sense
- Hide credentials/keys/passwords from others

Security Roles

- Administration/Full Authority
- Project/Namespace Admin
- Developer

Securing the API Server

- Used to allow authentication via client certificates
  - `--client_ca_file`

- Allow authentication via tokens; tokens are long-lived and cannot be refreshed (atm)
  - `--token_auth_file`

- HTTP basic httpswd file
  - `--basic_auth_file`

Attribute Based Access Control (ABAC)

- The **four** attributes that apply to authorization measures:
  - The user (as authenticated already)
  - Read only/Write — GET commands are read-only
  - The resource in question (pod/RC/service,etc)
  - The namespace

Specify Policies

- Specifying policies: when starting the API server, pass a single-line JSON file to `--authorization_policy_file`

  - `{"user":"ceposta"}`
  - `{"user":"ceposta", "resource": "pods", "readonly": true}`
  - `{"user":"ceposta", "resource": "events"}`
  - `{"user":"ceposta", "resource": "pods", "readonly": true, "ns": "projectBalvenie"}`

  - NOTE: This file is only reloaded when restarting API server

Service Accounts Intro

- Service accounts vs User accounts
  - User accounts for humans; service accounts for services within Pods
  - Service accounts are "namespaced"
  - Service account creation is much simpler/lightweight vs User creation
  - Allow services to access the Kubernetes API

Service Accounts Admission

- Acts as part of the API server, decorates pods with Service Account information:
  - Will assign `default` Service Account if one not specified
  - Will reject a Service Account if it specified and does not exist
  - Add ImagePullSecrets - for private repos
  - Adds volume for token-based API access - secret
  - Runs synchronously when pods are created

Secrets

- Image secrets
- Secret volumes
- Service accounts actually use secrets to pass API tokens
- Can pass sensitive data
  - Passwords
  - Keys
  - Certificates

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  password: dmFsdWUtMg0K
  username: dmFsdWUtMQ0K
```

Pod Using a Secret

- Example:

```
apiVersion: "v1"
  kind: "Pod"
  metadata:
    name: "mypod"
    namespace: "myns"
  spec:
    containers:
      -
        name: "mypod"
        image: "redis"
        volumeMounts:
          -
            name: "foo"
            mountPath: "/etc/foo"
            readOnly: true
    volumes:
      -
        name: "foo"
        secret:
          secretName: "mysecret"
```

Kubernetes Networking

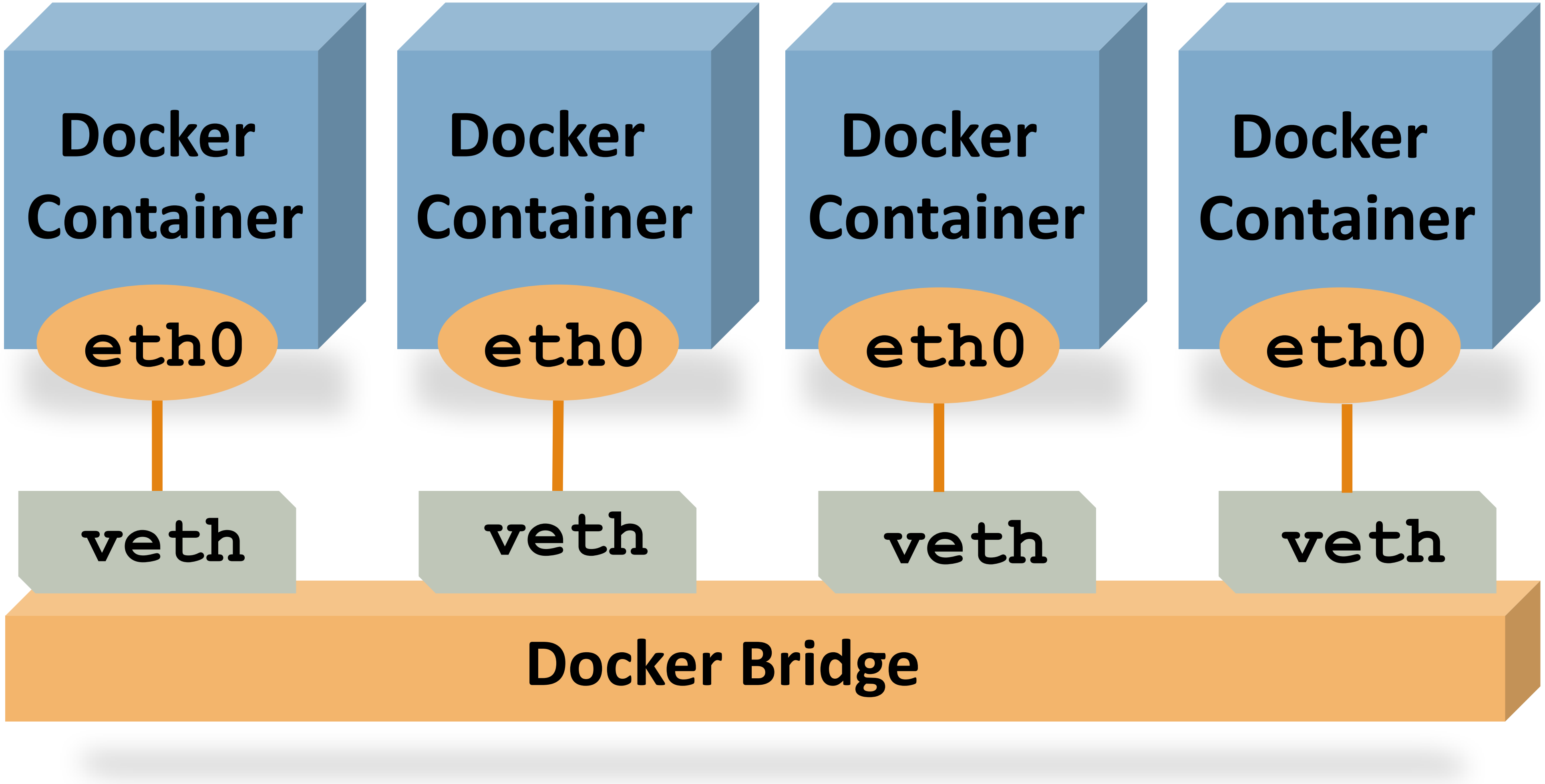
Docker Networking

- Local, host-only bridge (docker0)
- Create new adapters to the bridge (veth) for each container that’s created
- Veth is mapped to eth0 on a container
- Eth0 is assigned an IP from the range dedicated to the virtual bridge
- Result: Docker containers can talk to each other only on the same machine
- Containers on different hosts could have the exact same IP

Docker Networking

- In order for Docker containers to communicate across hosts, they need to allocate ports on the host
- This means containers must coordinate appropriately or allocate dynamically
  - And know when not to run out of ports
- This is difficult to do, doesn’t scale very well
- Dynamic port allocation tricky — now each app MUST take a “port” parameter and configured at runtime

Quickly Understand Default Docker Networking



Kubernetes Networking

- All pods can communicate with other pods w/out any NAT
- All nodes can communicate with pods without NAT
- The IP the pod sees is the same IP seen outside of the pod
- Cannot take Docker hosts out of the box and expect Kubenetes to work
- This is a simpler model
  - Reduces friction when coming from VM environments where this is more or less true

Pod to Pod, Pod to External

- Flat networking space
- So the transition is consistent VM→Pod
- No additional container or application gymnastics /NAT/etc. to have to go through each time you deploy
- Pods have their own “port space” independent of other pods
- Don’t need to explicitly create Docker links between containers
  - Would only work on a single node anyway

Pod to Pod, Pod to External

- Otherwise, dynamic allocation of ports on Host every time a pod needs a port gets very complicated for orchestration and scheduling
  - Exhaustion of ports
  - Reuse of ports
  - Tricky app configuration
  - Watching/cache invalidation
  - Redirection, etc.
  - Conflicts
  - NAT breaks self-registration mechanisms, etc.

Pods Have Single IP Address for all Containers

- IP address visible inside and outside of the container
- Self-registration works fine as you would expect as does DNS
- Implemented as a “pod container” which holds the network namespace (net) and “app containers” which join with Docker’s —net=container:<id>
- In Docker world, the IP inside the container is NOT what an entity outside of the container sees, even in another container

Container to Container With Pod

- All containers behave as though they're on a single host,
  - i.e. They see the same ports and network, plus they can communicate with each other over localhost
- Simplicity
  - Well known ports, 80, 22, etc.
- Security
  - Ports bound on localhost are only visible within the pod/containers, never outside
- Performance
  - Don't have to take network stack penalties, marshaling, un-marhsaling, etc.

Container to Container With Pod

- Very similar to running multiple processes in a VM host for example
- Con: No container-local ports, could clash, etc. but these are minor inconveniences at the moment and workarounds are being implemented
- However, pods come with the premise of shared resources (volumes, CPU, memory, etc.) so a reduction in isolation is really expected
- If you need isolation, use Pods not containers to achieve this

Pod to Service

- Service IPs are VIP
- And `kube-proxy` alters `iptables` on the node to trap service IPs and redirect them to the correct backend
- Simple, hi-performance, HA solution

External to Pod

- This gets tricky...
- Need to set up external load balancer to fwd all service IPs and load balance against all nodes
- The `kube-proxy` should trap that IP and send it to service?
- Expose services directly to node hosts? —> suitable for poc type workloads, but not suitable for real prod workloads

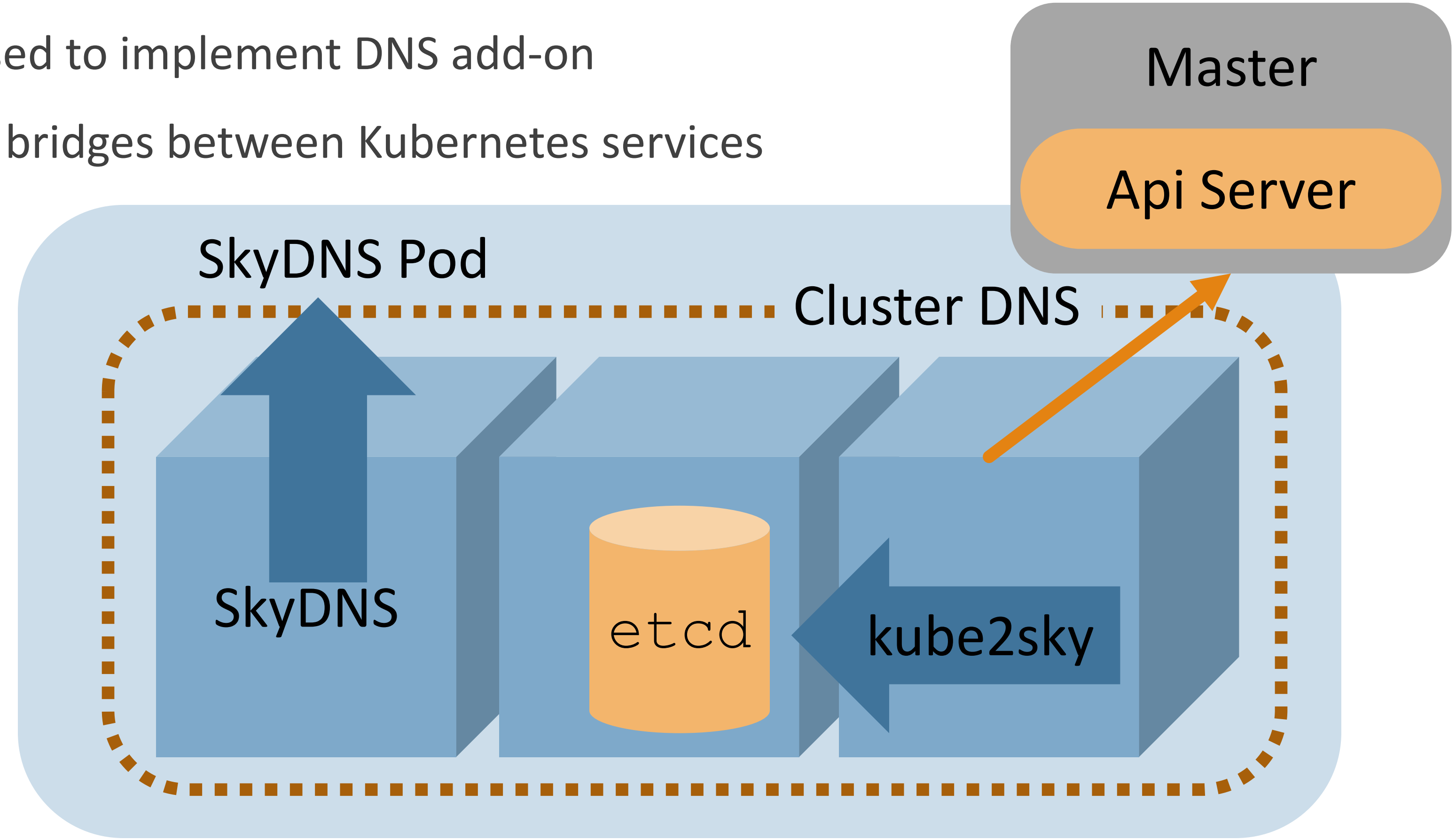
DEMO

Suppose to be a demo here.

Cluster Add-Ons

Cluster DNS

- Add-Ons implemented as Services and Replication Controllers
- Sky DNS used to implement DNS add-on
- A pod that bridges between Kubernetes services and DNS



Cluster DNS

- A Kubernetes service that is the DNS provider
  - i.e. has an VIP, etc.
- Kublet configured to decorate the pods with correct DNS server
  - Can configure the `kubelet` manually if not automatically set up:

```
--cluster_dns=<DNS service ip>
```

```
--cluster_domain=<default local domain>
```

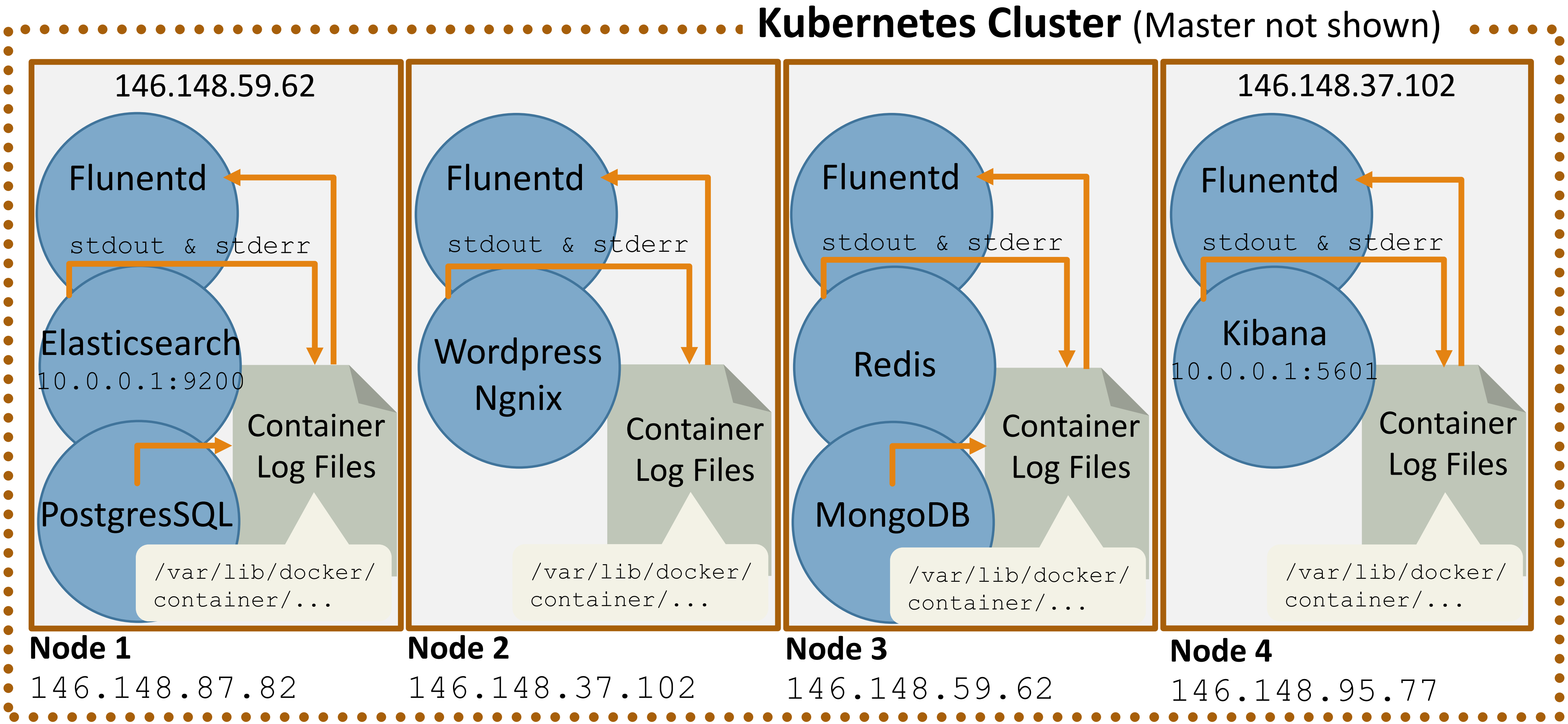
Cluster DNS

- A records are created for services in the form `svc-name.ns-name.svc.cluster.local`
- Headless service - no clusterIP - are DNS round-robin
- SRV records (discovering services and ports) `_my-port-name._my-port-protocol.my-svc.my-namespace.svc.cluster.local`
  - Resolves to the hostname `my-svc.my-namespace.svc.cluster.local` and the port

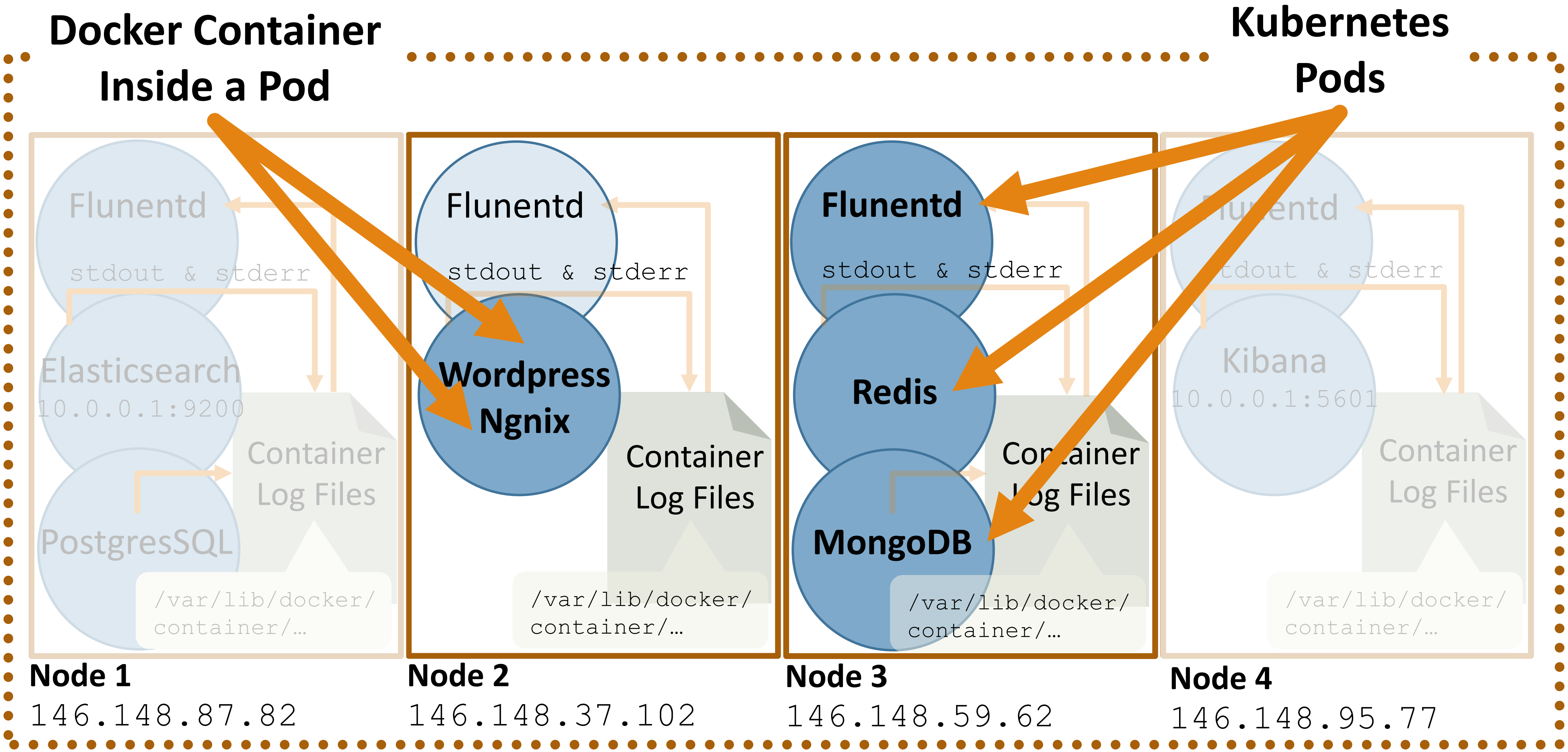
Cluster Logging with Elasticsearch and Fluentd

- Log collector on each node
- Implemented with `fluentd`, as a pod
- Watches all containers' logs on that node and pump them to Elastic search cluster
- Elasticsearch can be queried via Kibana

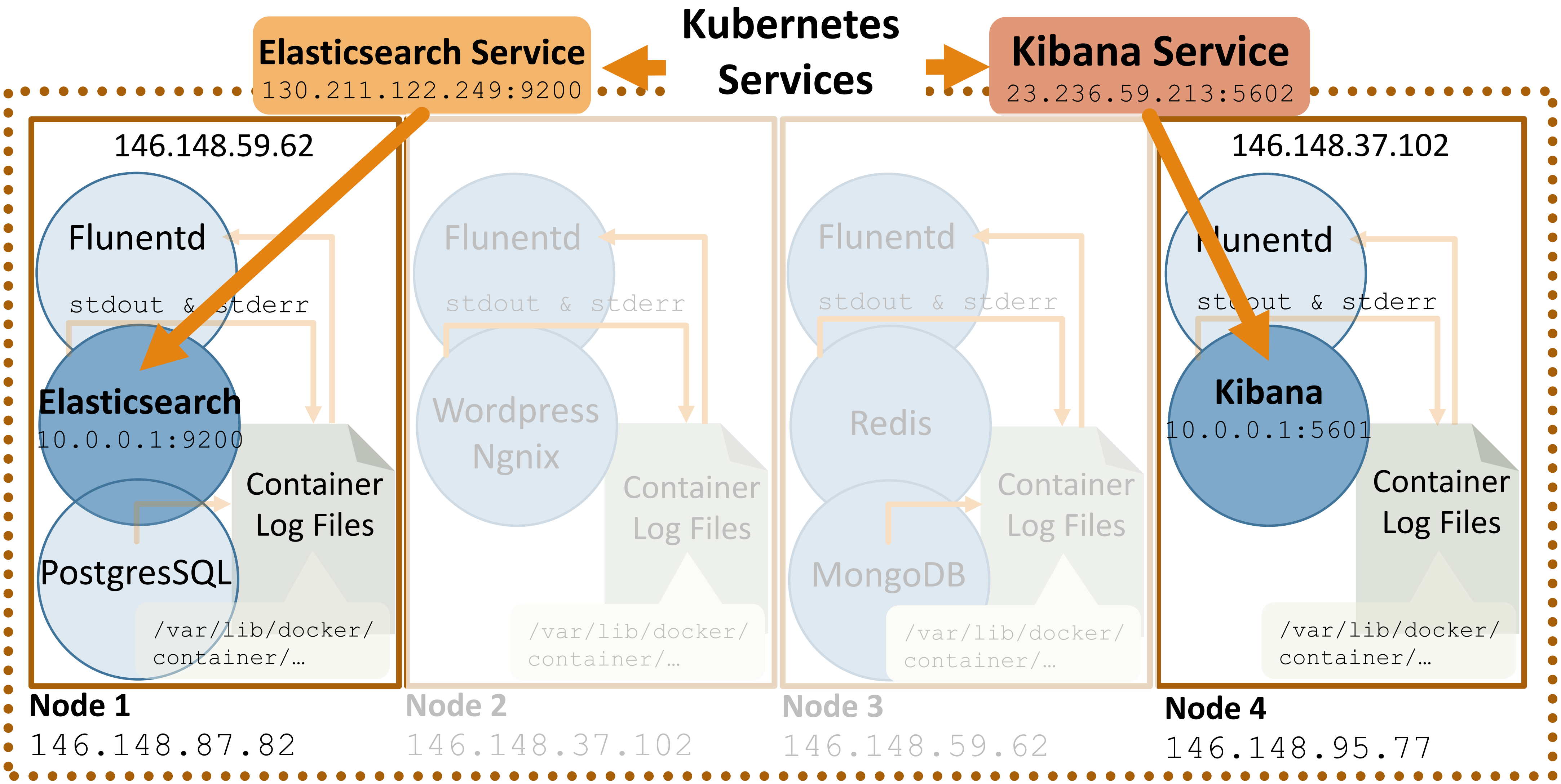
Elasticsearch and Fluentd



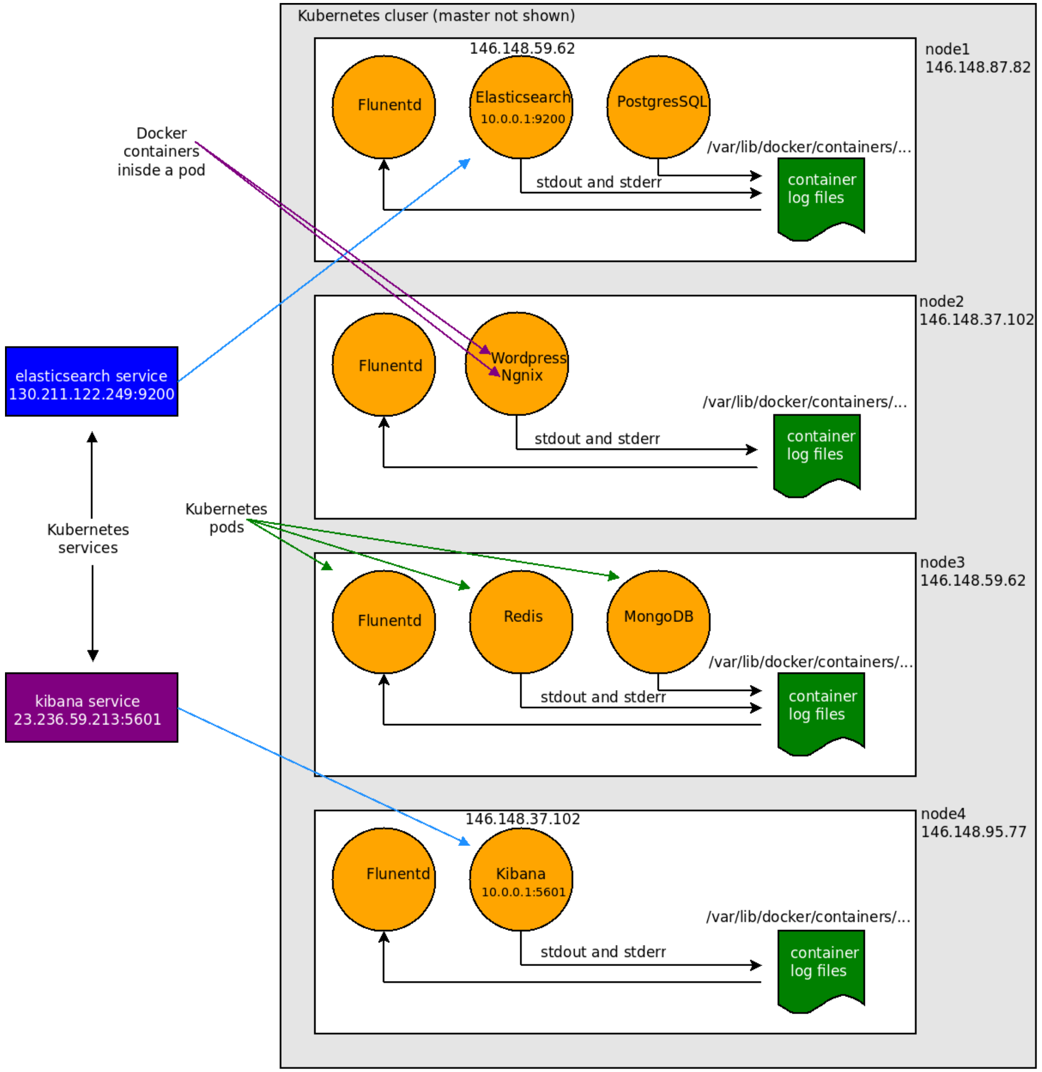
Elasticsearch and Fluentd



Elasticsearch and Fluentd



ORIGINAL GRAPHIC THAT WAS REPLACED



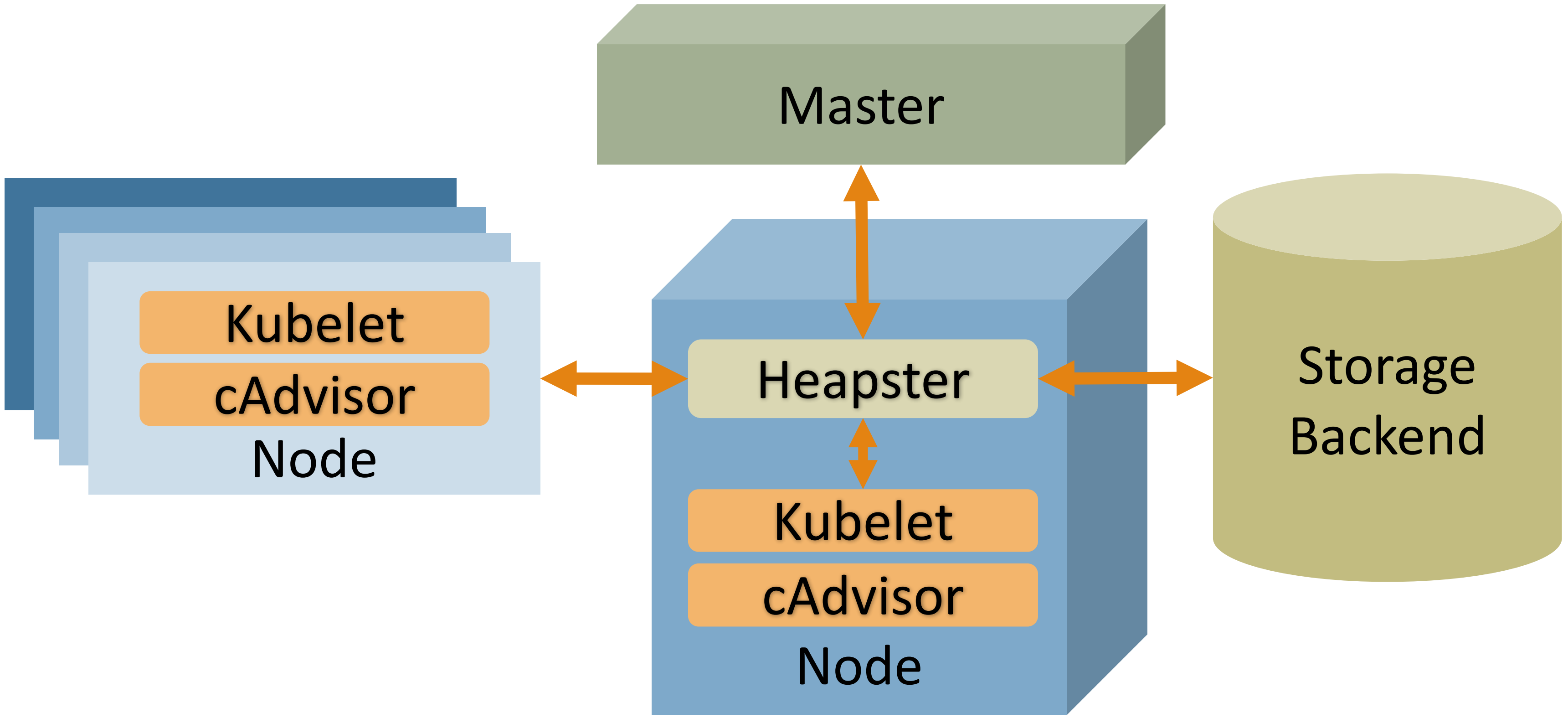
DEMO

- Elasticsearch and fluentd Demo

Container Level Monitoring

- Need visibility into the cluster as an aggregate and individually where appropriate
  - cAdvisor
  - Heapster
  - Influxdb/Prometheus/Graphite
  - Grafana
- You can compare options at:
  - `prometheus.io/docs/introduction/comparison/`

Container Level Monitoring



cAdvisor UI

Usage

Overview

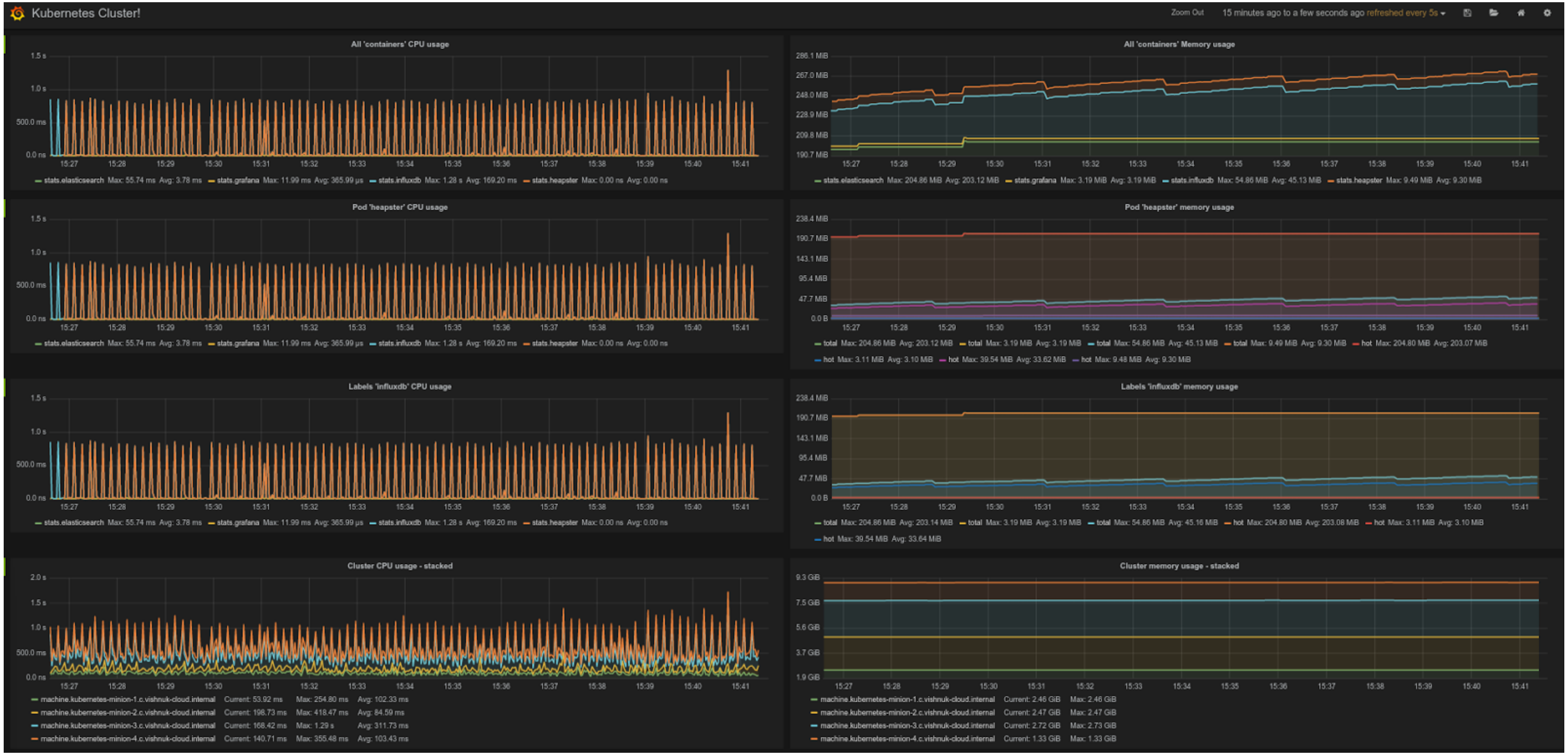
Processes

User	PID	PPID	Start Time	CPU %	MEM %	RSS	Virtual Size	Status	Running Time	Command
www-data	26948	44499	11:40	0.00	0.00	5.34 KiB	169.08 KiB	S	00:00:00	apache2
www-data	26949	44499	11:40	0.00	0.00	5.34 KiB	169.08 KiB	S	00:00:00	apache2
www-data	26954	44499	11:40	0.00	0.00	5.34 KiB	169.08 KiB	S	00:00:00	apache2
www-data	26955	44499	11:40	0.00	0.00	5.34 KiB	169.08 KiB	S	00:00:00	apache2
www-data	26956	44499	11:40	0.00	0.00	5.34 KiB	169.08 KiB	S	00:00:00	apache2
root	44499	5093	11:19	0.00	0.00	11.32 KiB	169.05 KiB	Ss	00:00:00	apache2

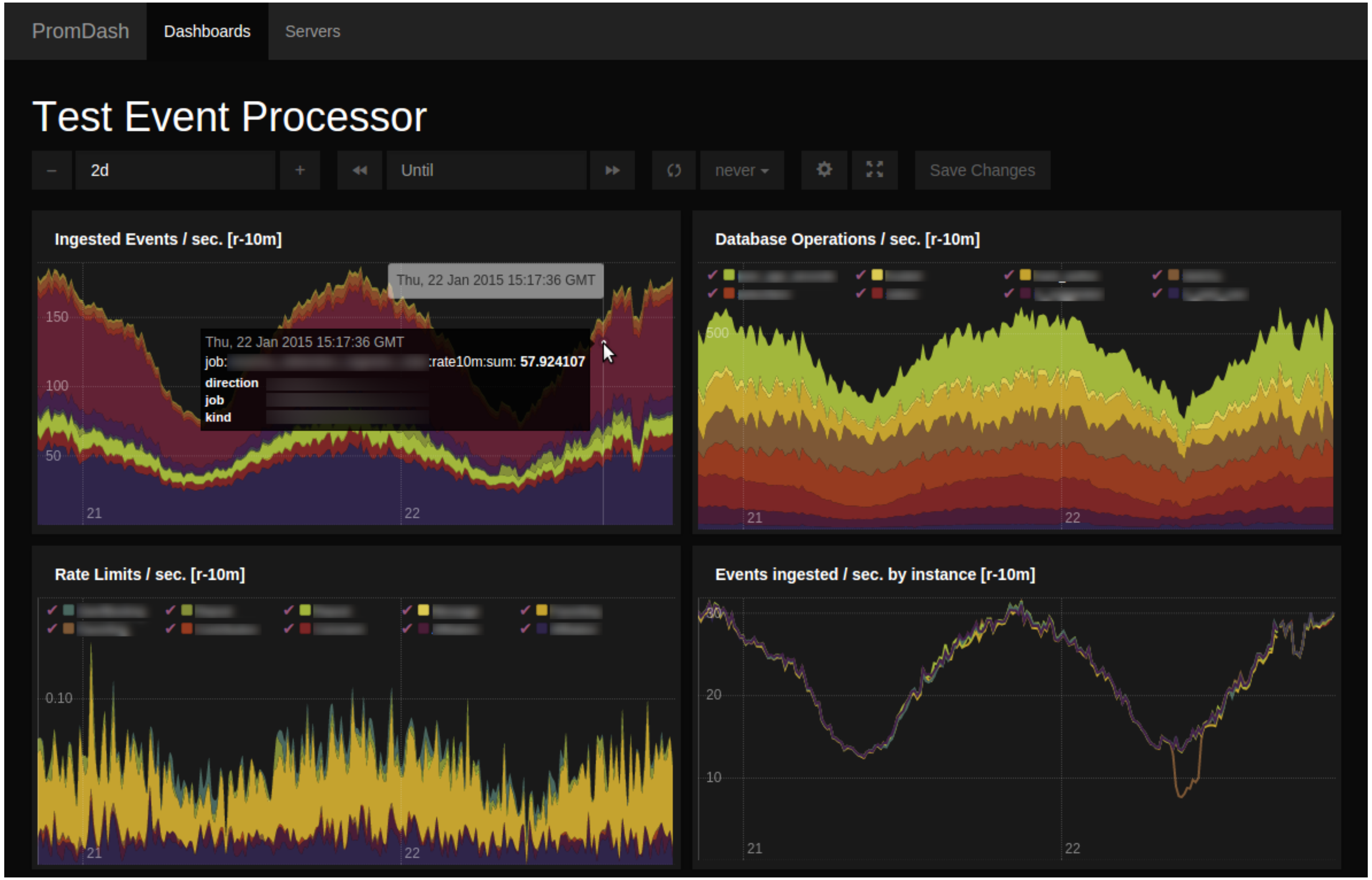
CPU

Total Usage

Influxdb


Kubernetes Cluster!
Zoom Out
15 minutes ago to a few seconds ago refreshed every 5s
All 'containers' CPU usage
stats.elasticsearch Max: 55.74 ms Avg: 3.78 ms
stats.grafana Max: 11.99 ms Avg: 365.99 µs
stats.influxdb Max: 1.28 s Avg: 169.20 ms
stats.heapster Max: 0.00 ns Avg: 0.00 ns
All 'containers' Memory usage
stats.elasticsearch Max: 204.86 MiB Avg: 203.12 MiB
stats.grafana Max: 3.19 MiB Avg: 3.19 MiB
stats.influxdb Max: 54.86 MiB Avg: 45.13 MiB
stats.heapster Max: 9.49 MiB Avg: 9.30 MiB
Pod 'heapster' CPU usage
Pod 'heapster' memory usage
Labels 'influxdb' CPU usage
Labels 'influxdb' memory usage
Cluster CPU usage - stacked
machine.kubernetes-minion-1.c.vishnuk-cloud.internal Current: 53.92 ms Max: 254.80 ms Avg: 102.33 ms
machine.kubernetes-minion-2.c.vishnuk-cloud.internal Current: 198.73 ms Max: 418.47 ms Avg: 84.59 ms
machine.kubernetes-minion-3.c.vishnuk-cloud.internal Current: 168.42 ms Max: 1.29 s Avg: 311.73 ms
machine.kubernetes-minion-4.c.vishnuk-cloud.internal Current: 140.71 ms Max: 355.48 ms Avg: 103.43 ms
Cluster memory usage - stacked
machine.kubernetes-minion-1.c.vishnuk-cloud.internal Current: 2.46 GiB Max: 2.46 GiB
machine.kubernetes-minion-2.c.vishnuk-cloud.internal Current: 2.47 GiB Max: 2.47 GiB
machine.kubernetes-minion-3.c.vishnuk-cloud.internal Current: 2.72 GiB Max: 2.73 GiB
machine.kubernetes-minion-4.c.vishnuk-cloud.internal Current: 1.33 GiB Max: 1.33 GiB

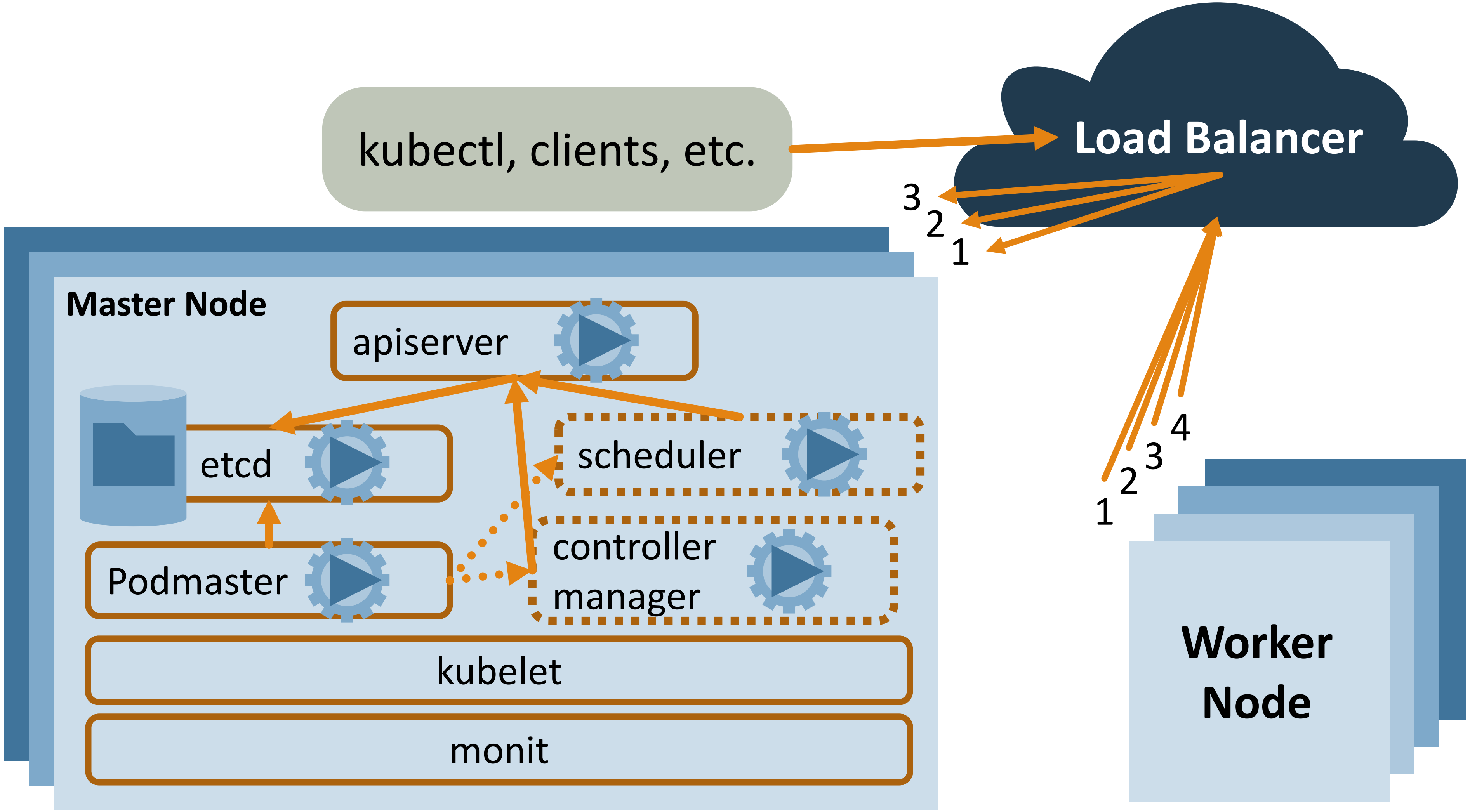
Prometheus



PromDash
Dashboards
Servers
Test Event Processor
2d
Until
never
Save Changes
Ingested Events / sec. [r-10m]
Thu, 22 Jan 2015 15:17:36 GMT
Thu, 22 Jan 2015 15:17:36 GMT
job:
:rate10m:sum: 57.924107
direction
job
kind
150
100
50
21
22
Database Operations / sec. [r-10m]
21
22
Rate Limits / sec. [r-10m]
0.10
21
22
Events ingested / sec. by instance [r-10m]
20
10
21
22

Cluster High Availability

High Availability

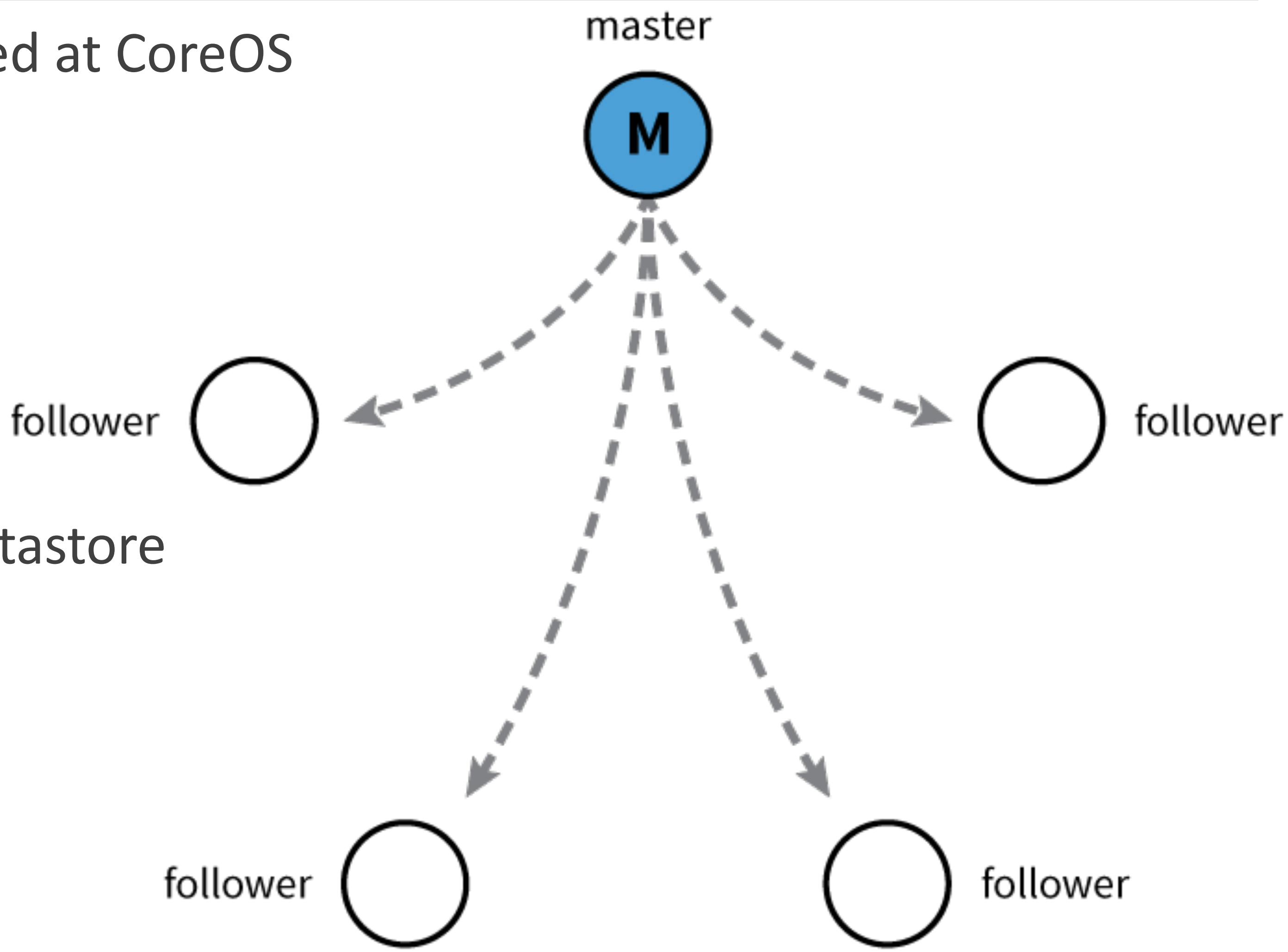


High Availability Components

- HA master nodes
- An `etcd` datastore
- Replicated, load-balanced, API server
- Elected scheduler and controllers

ETCD

- Open source project started at CoreOS
- Distributed database
- CAP Theorem? == CP
- Raft algorithm/protocol
- Watchable
- And `etcd` provides HA datastore



High Availability Components

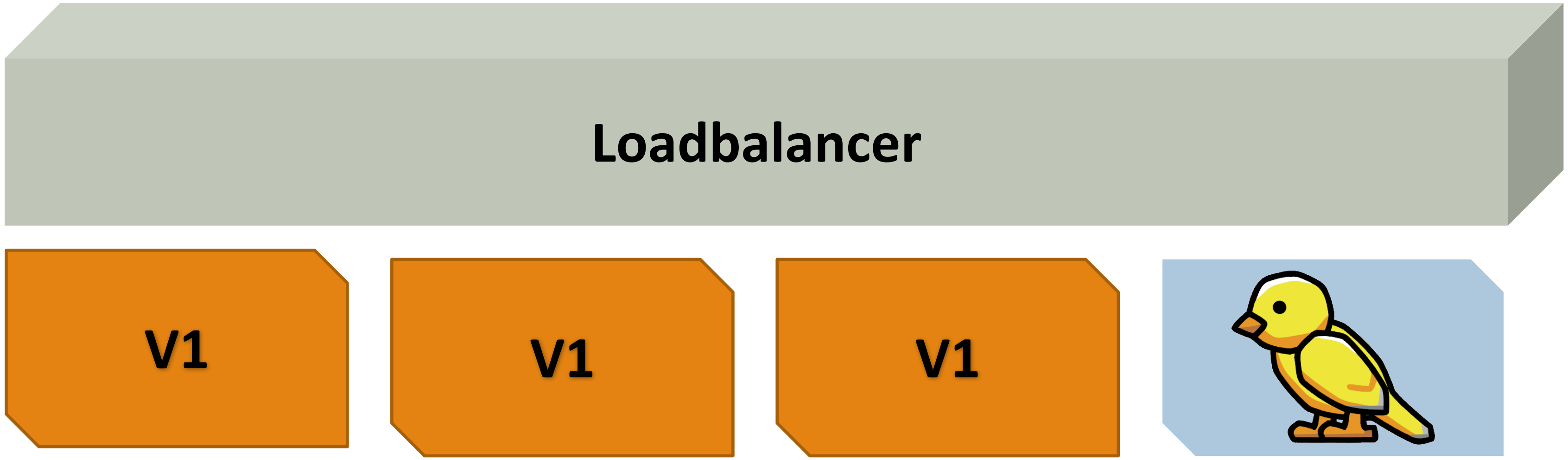
- Run `kubelet` on the master nodes to monitor the API server process and restart on failure
  - So, `systemctl` enables `kubelet` and `systemctl` enables `docker`.
- Replicated `etcd`
- Run shared storage locations for each of the `etcd` nodes
- Network loadbalancers over the API servers
- Run `podmaster` which coordinates a lease-lock election using `etcd`

Updating Applications - Releasing Updates

Release Types

- Canary release
- Blue/green deployment
- A/B testing
- Rolling upgrade/rollback

Canary Release



Canary Release with Kubernetes

- App to deploy:

```
labels:
        app: guestbook
        tier: frontend
        track: canary
```

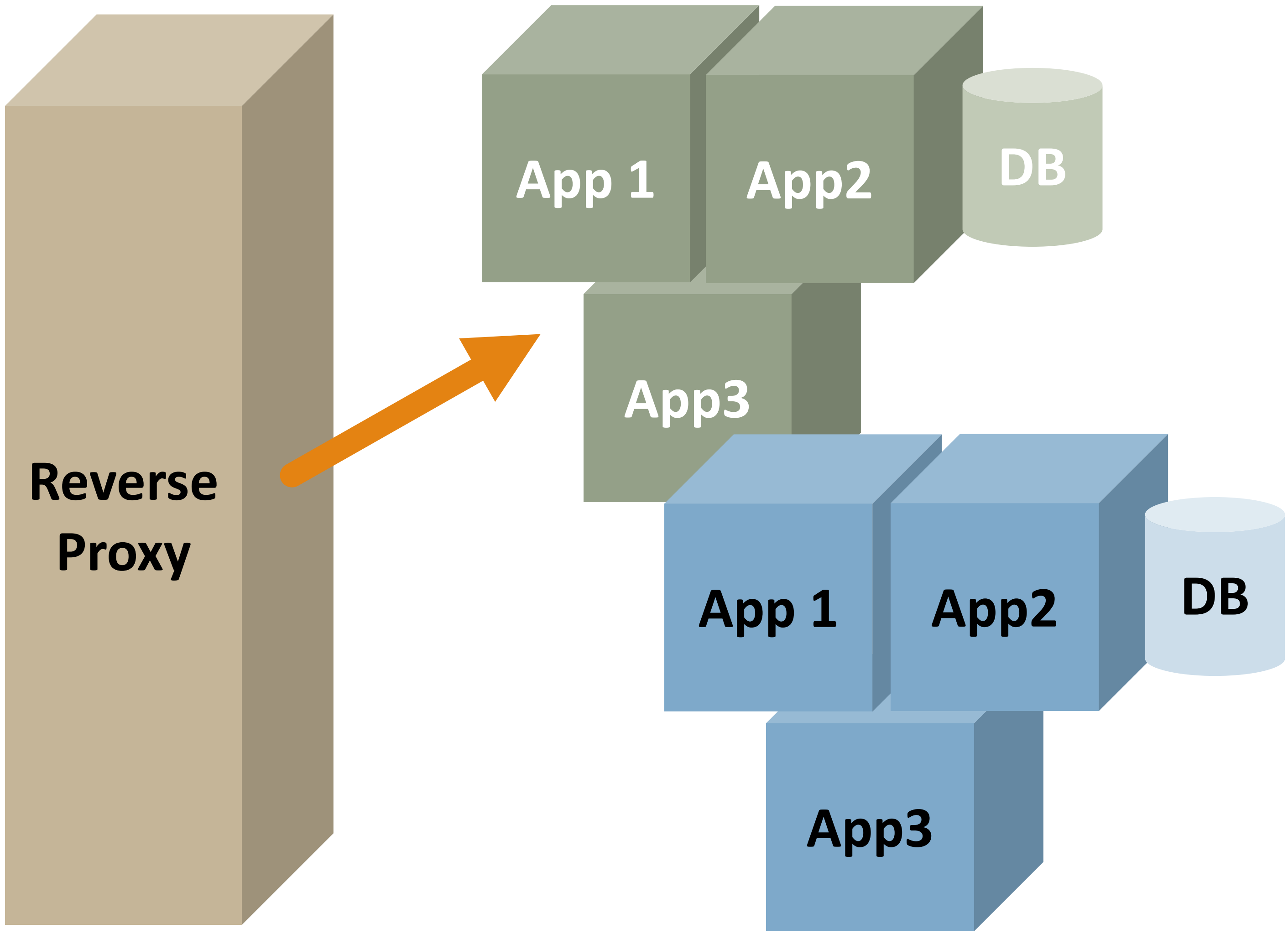
- Existing set of apps:

```
labels:
        app: guestbook
        tier: frontend
        track: stable
```

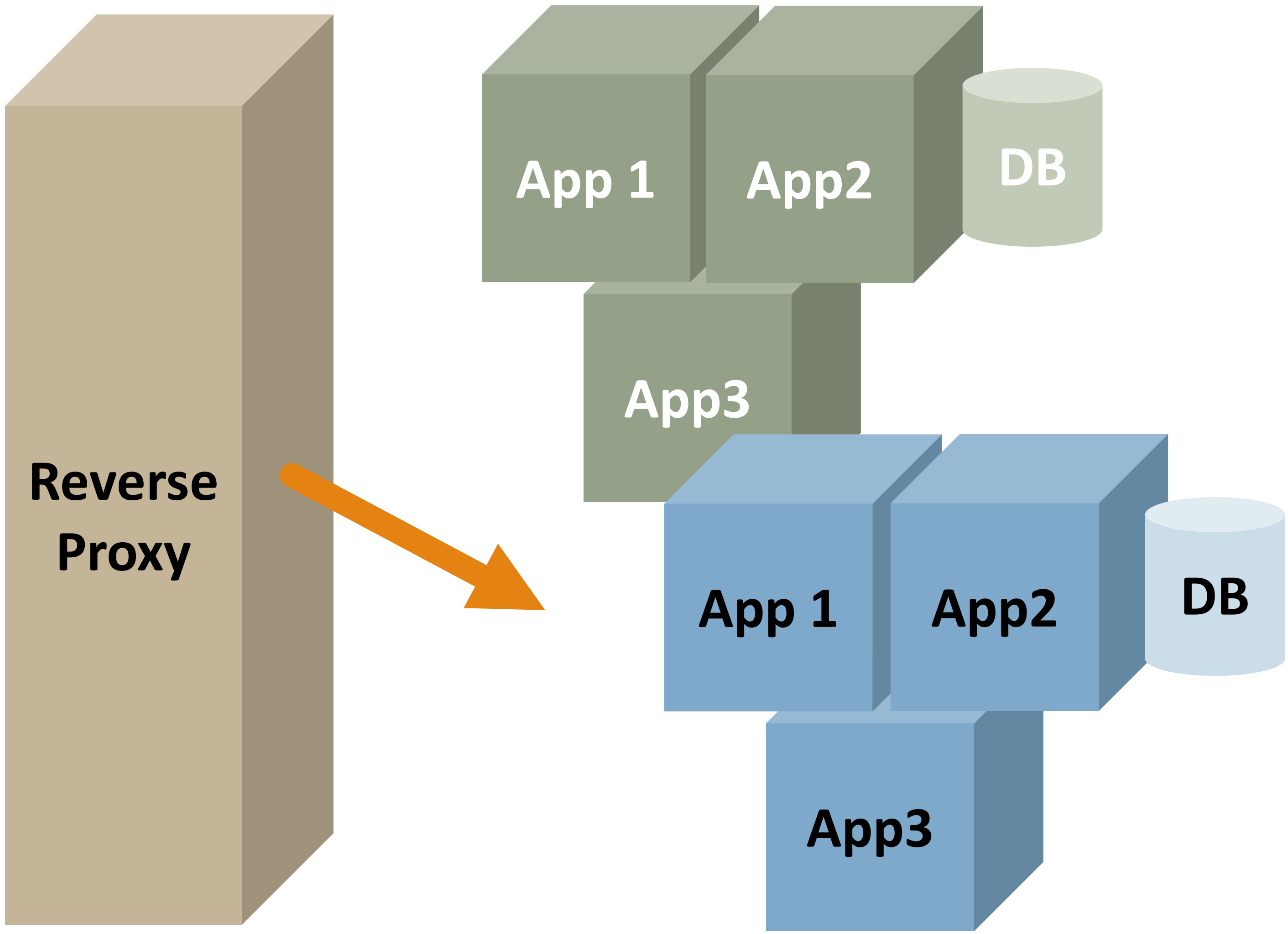
- Service selector:

```
selector:
        app: guestbook
        tier: frontend
```

Blue/Green Deployment



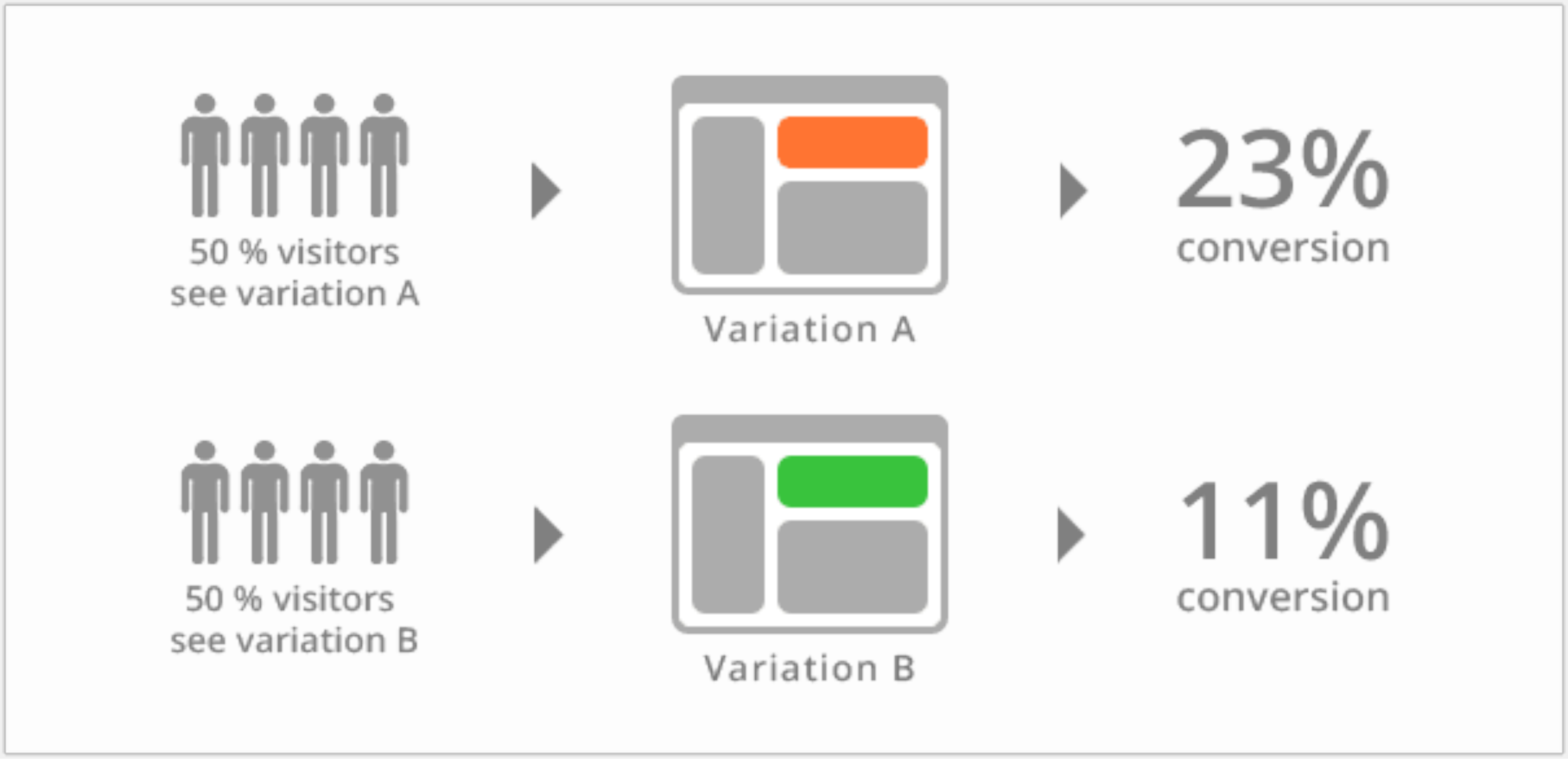
Blue/Green Deployment



Blue/Green Deployment with Kubernetes

- Have two separate replication controllers, one blue, one green
- Have labels "color=green", "color=blue”
- Service selector = "color=green"
- Change selector to "color=blue" to switch
- Can switch back

A/B Testing


50 % visitors
see variation A
Variation A
23%
conversion
50 % visitors
see variation B
Variation B
11%
conversion

Rolling Update

- Bring up a container with the new version, same fleet of containers
- Bring down one of the old version
- Bring up a second container with the new version
- Repeat
- To be aware: potentially scaling while doing Rolling Updates

Rolling Update: How To

- Use replication controllers to control the number of replicas at a given step
- Use `kubectl` rolling-update
- Replaces an old RC with a new RC
- Must be in same namespace
- Share at least one label name, different value

- Example:

```
kubectl rolling-update frontend-v1 -f frontend-v2.json
```

Rolling Update: How To Recover

- What happens if a failure is introduced part of the way through the rolling update?
- Kubernetes keeps track and annotates the RC with info:
- `kubernetes.io/desired-replicas`
  - The number that this replica controller needs to get to
- `kubernetes.io/update-partner`
  - Who's the other half of the replica-set
- Recovery is achieved by running the same command again

Rolling Update: How To Recover

- While size of `foo-next < desired-replicas` annotation on `foo-next`
  - Increase size of `foo-next`
  - If size of `foo > 0` decrease size of `foo`
- Go to Rename

Demo Rolling Update

NEEDS TO BE DETERMINED

Automating Generation

OF KUBERNETES RESOURCES FOR JAVA

Docker Maven Plugin

- Set of maven goals for managing Docker builds and containers
- Can be run as part of a CI/build step in your existing build or CI pipelines
- Requires access to a Docker Daemon for builds
- Can build images, start/stop containers, etc.

- https://github.com/rhuss/docker-maven-plugin

Docker Maven Plugin

`docker:start`	Create and start containers
`docker:stop`	Stop and destroy containers
`docker:build`	Build images
`docker:watch`	Watch for doing rebuilds and restarts
`docker:push`	Push images to a registry
`docker:remove`	Remove images from local Docker host
`docker:logs`	Show container logs

Docker Maven Plugin: Build

- `mvn package docker:build`
- Can build a Docker image as part of `mvn lifecycle`
- Package files from project (build artifacts, configs, etc.) into a Docker image
- Which files are selected using `maven-assembly-plugin`
- Selected files are inserted into base image at specified location
- Default `/maven`
- See the assembly descriptor file format
- Once image is built, can use `maven-failsafe-plugin` to run integration tests

Build Output

```
<configuration>
  <images>
    <image>
      <alias>service</alias>
      <name>jolokia/docker-demo:${project.version}</name>

      <build>
         <from>java:8</from>
         <assembly>
           <descriptor>docker-assembly.xml</descriptor>
         </assembly>
         <ports>
           <port>8080</port>
         </ports>
         <cmd>
            <shell>java -jar /maven/service.jar</shell>
         </cmd>
      </build>
...
```

Build Output

```
        <run>
            <ports>
              <port>tomcat.port:8080</port>
            </ports>
            <wait>
              <url>http://localhost:${tomcat.port}/access</url>
              <time>10000</time>
            </wait>
            <links>
              <link>database:db</link>
            </links>
          </run>
       </image>

       <image>
         <alias>database</alias>
         <name>postgres:9</name>
         <run>
           <wait>
             <log>database system is ready to accept connections</log>
             <time>20000</time>
           </wait>
         </run>
       </image>
     </images>
   </configuration>
```

Docker Maven Plugin: Watch

- Can watch for changes in project and rebuild
- Rebuild Docker image
- Re-start existing running container
- Fast development feedback/loop

- Examples:

```
mvn package docker:build docker:watch -Ddocker.watchMode=build
```

```
mvn docker:start docker:watch -Ddocker.watchMode=run *
```

Docker Maven Plugin: Watch

```
<configuration>
   <!-- Check every 10 seconds by default -->
   <watchInterval>10000</watchInterval>
   <!-- Watch for doing rebuilds and restarts -->
   <watchMode>both</watch>
   <images>
      <image>
         <!-- Service checks every 5 seconds -->
         <alias>service</alias>
         ....
         <watch>
            <interval>5000</interval>
         </watch>
      </image>
      <image>
         <!-- Database needs no watching -->
         <alias>db<alias>
         ....
         <watch>
            <mode>none</mode>
         </watch>
      </image>
      ....
   </images>
</configuration>
```

Fabric8 Maven Plugin

- `fabric8:json`
- `fabric8:apply`
- `fabric8:rolling`
- `fabric8:devops`
- `fabric8:create-routes`
- `fabric8:recreate`

Fabric8 Maven Plugin: JSON

- Generates `kubernetes.json` file based on Maven settings
  - Can generate ReplicationController, Services, Pods
- Attaches `kubernetes.json` and versions as part of the build
  - Will be included in the artifacts uploaded to artifact repo

JSON Options

- Options:
  - Hand-generate your own file and let `mvn` coordinates be applied
  - Use default `mvn` properties and let `fabric8:json` generate the JSON file
  - Use annotation processors and typesafe DSL builders directly
  - Enrich the generated JSON with additional stuff

Fabric8 Maven Plugin: JSON

- Example:

```
<project>
...
  <properties>
    <fabric8.env.FOO>bar</fabric8.env.FOO>
    ...
  </properties>
...
</project>
```

Fabric8 Maven Plugin: JSON

```
docker.image

fabric8.combineDependencies

fabric8.container.name

fabric8.containerPrivileged

fabric8.env.FOO = BAR

fabric8.extra.json

fabric8.generateJson

fabric8.iconRef

fabric8.iconUrl

fabric8.iconUrlPrefix
```

Fabric8 Maven Plugin: JSON

```
fabric8.iconBranch
fabric8.imagePullPolicy
fabric8.imagePullPolicySnapshot
fabric8.includeAllEnvironmentVariables
fabric8.includeNamespaceEnvVar
fabric8.label.FOO = BAR
fabric8.livenessProbe.exec
fabric8.livenessProbe.httpGet.path
fabric8.livenessProbe.httpGet.port
fabric8.livenessProbe.httpGet.host
fabric8.livenessProbe.port
fabric8.namespaceEnvVar
```

Fabric8 Maven Plugin: JSON

```
fabric8.parameter.FOO.description
fabric8.parameter.FOO.value
fabric8.port.container.FOO = 1234
fabric8.port.host.FOO = 4567
fabric8.provider
fabric8.readinessProbe.exec
fabric8.readinessProbe.httpGet.path
fabric8.readinessProbe.httpGet.port
fabric8.readinessProbe.httpGet.host
fabric8.readinessProbe.port
fabric8.replicas
fabric8.replicationController.name
```

Fabric8 Maven Plugin: Apply

- Takes the `kubernetes.json` from `fabric8:json` and "applies" it to Kubernetes
- Synonymous with `kubectl create -f <resource`
- Can be applied part of `mvn build/mvn` lifecycle
- Just configure these environment variables
  - `KUBERNETES_MASTER` - The location of the Kubernete Master
  - `KUBERNETES_NAMESPACE` - The default Namespace used on operations

Fabric8 Maven Plugin: Apply

- Example:

```
mvn fabric8:apply -Dfabric8.recreate=true \ -
Dfabric8.domain=foo.acme.com -Dfabric8.namespace=cheese
```

- `fabric8.apply.create`
- `fabric8.apply.servicesOnly`
- `fabric8.apply.ignoreServices`
- `fabric8.apply.createRoutes`
- `fabric8.domain`
- `fabric8.namespace`

Lab

End of Chapter