DC/OS Fundamentals

Day Two - Container Orchestration with Marathon

MESOSPHERE

Agenda - Day Two

Container Orchestration

Marathon

Container Orchestrator Capabilities

Scheduling	Resource Management	Service Management
● Placement	● Memory	● Labels
● Replication/Scaling	● CPU	● Groups/Namespaces
● Resurrection	● GPU	● Dependencies
● Rescheduling	● Volumes	● Health Checks
● Rolling Deployment	● Ports	● Readiness Checks
● Upgrades	● IP Addresses	
● Downgrades	● Images/Artifacts	
● Collocation		

Overview

- Marathon is a DC/OS component that is installed on each master node
- In multi-master clusters, there is one leading Marathon master and the others are followers
- It is designed to launch and manage long running services, such as:
  - Web servers
  - Application servers
  - DB servers
  - API servers
- Marathon exposes a REST API that allows users/systems to send requests to perform different actions:
  - `http://<master_ip>/marathon/v2/apps`
- Also referred to as a meta-framework in Mesos, in that it is able to run and manage other Mesos frameworks (Cassandra, Kafka, Kubernetes, etc.)

Marathon Services

Services

- A unit of deployment in Marathon
- Service specification is declared in a JSON document
- Specification document has some required fields, but most are optional
- The service specifications match what the Marathon API (`http(s)://<master_ip>/marathon/v2/apps`) endpoint returns:
  - Documentation
- REST API
  - Marathon has a powerful API that can be used to build your PaaS, integrate with your CI/CD pipeline, and much more
  - Documentation

Marathon CLI

```
$ dcos marathon --help

Description:
     Deploy and manage applications to DC/OS.
Usage:
     dcos marathon --config-schema
     dcos marathon --help
     dcos marathon --info
     dcos marathon about
     dcos marathon app add [<app-resource>]
     dcos marathon app list [--json]
     dcos marathon app remove [--force] <app-id>
     dcos marathon app restart [--force] <app-id>
     dcos marathon app show [--app-version=<app-version>] <app-id>
     dcos marathon app start [--force] <app-id> [<instances>]
     dcos marathon app stop [--force] <app-id>
     dcos marathon app kill [--scale] [--host=<host>] <app-id>
     dcos marathon app update [--force] <app-id> [<properties>...]
     dcos marathon deployment list [--json <app-id>]
     dcos marathon deployment rollback <deployment-id>
     dcos marathon deployment stop <deployment-id>
```

Commonly Used Commands

- `$ dcos marathon app add <service_spec>`
    - Install and execute a service as defined in `<service_spec>` JSON file
- `$ dcos marathon app update <service_spec> < <updated_service_spec>`
    - Update an installed service to match the specification provided in `<updated_service_spec>` JSON
- `$ dcos marathon app remove <service_id>`
    - Remove `<service_id>` from the cluster
- `$ dcos marathon app stop <service_id>`
    - Scale `<service_id>` down to 0 instances
- `$ dcos marathon app start <service_id>`
    - Scale `<service_id>` up to 1 instance
- `$ dcos marathon app list`
    - List services that exist on the cluster

Example Service Specification

```
{
    “id”: “/hello-world”,
    “cpus”: 0.1,
    “mem”: 32,
    “cmd”: “while [ true ]; do echo ‘Hello Marathon’; sleep 10; done”,
    “instances”: 2
}
```

Basic Service Spec Parameters

Name	Description
`id`	Unique identifier for the service. Consists of a series of names separated by forward slashes.
`cpus`	The CPU allocation for each instance (container). Can be a whole integer or a fractional value.
`mem`	The RAM allocation for each instance (container). Specified in MB.
`cmd`	The command that is executed. This value is wrapped by Mesos via `/bin/sh -c ${cmd}`. Either cmd or args must be supplied.
`container`	Additional data that is to be passed to the containerizer on container launch. These consist of a type, zero or more volumes, network port mappings, and additional type specific options.
`instances`	Number of instances (tasks) to launch

Basic Service Spec Parameters

Name	Description
`fetch`	(Array) List of URIs that will be fetched, cached, and optionally extracted by Mesos before executing the cmd
`env`	(Object) Set of key-value pairs that are passed into the container as environment variables
`constraints`	(Array) List of placement constraints
`AcceptedResourceRoles`	(Array) List of roles from which offers will be accepted for this service
`labels`	(Object) Arbitrary set of key-value pairs which can be used to tag a service
`healthChecks`	(Array) List of health checks to perform on instances of the service

Fetching Artifacts

- Provide the ability to retrieve data from a remote location and make it present within a container
- By default, each URI is downloaded directly into a container
- When caching is enabled, each URI is downloaded and written to disk on an agent, and then sent into a container
    - Subsequent requests for the same URI result in the artifact being downloaded from the cache
- Use-cases:
    - Applications and/or dependencies
    - Ops tooling
    - Static media
    - Authentication data
- Supports the following protocols:
    - `file:///`, `http://`, `https://`, `ftp://`, `ftps://`, `hdfs://`, `s3://`, `s3a://`, `s3n://`
- Can extract archives based off file extension:
    - `.tgz`, `.tar.gz`, `.tbz2`, `.tar.bz2`, `.txz`, `.tar.xz`, `.zip`

Lab 5

Launch a Marathon Service

Software Containers

Container Basics

- Core functionality is implemented in the Linux kernel, made up of two concepts:
    - `cgroups`
    - `namespaces`
- Control groups (`cgroups`) provide fine-grained resource allocation and accounting to a hierarchy of processes
- `namespaces` provide isolation between the host and a container, as well as between containers. On Linux, there are 6 namespace types that can be created:
    - pid
    - network
    - mount
    - ipc
    - uts
    - user

Universal Container Runtime (UCR)

- Supports both Mesos based containers along with the Docker container image format
- If no format is specified, Mesos containers are the default
- Docker and Mesos containers both use the same underlying containerization technologies (`cgroups` and `namespaces`)
- Supports execution of binary executables and Docker container images
- Has no dependency on Docker engine
- When launching a task, makes system calls to the kernel to make the appropriate `cgroup` and `namespace(s)`

Docker Container Runtime

Docker-specific features in Marathon and Mesos

1. Dynamic port-mapping for bridge networking, with service ports (for integration with Marathon-LB)
2. Mesos sandbox is mounted inside the container at `${MESOS_SANDBOX}` path

Lab 6

Launch a Docker Based Marathon Service

Placement Constraints

Overview

- Constraints are a concept in Marathon that leverage agent attributes in Mesos which provide a powerful language useful for specifying where services may or may not run in the cluster
- Use-cases:
  - Pinning services to hosts
  - Distributing containers into separate failure domains
  - Ensuring multiple instances of a service don’t run on the same agent
  - Ensuring some services run on agent nodes with different hardware capabilities (e.g. SSDs)
- Limitations:
  - Constraints can not match resources, only attributes
  - Constraints are distinct from *affinity*, a feature not yet implemented in Marathon

Setting Attributes on Agents

- During install, use configuration management to populate this type of information for you
- To setup manually, SSH to an agent node which you would like to add an attribute to and perform the following:
  - Create/modify the file at `/var/lib/dcos/mesos-slave-common`
  - Add attributes following the pattern below:
    `MESOS_ATTRIBUTES=key1:value1;key2:value2,...`
  - Remove the current resource/attribute file:
    `rm -f /var/lib/mesos/slave/meta/slaves/latest`
  - Restart the mesos-slave daemon:
    `systemctl restart dcos-mesos-slave` (private agents)
    `systemctl restart dcos-mesos-slave-public` (public agents)
  - Deploy new services using the new attributes through specifying constraints in your service spec

Resource Roles

Resources are the fundamental abstraction in Mesos. Agent nodes advertise available resources when they communicate with a master.

Resources can be grouped by roles to partition cluster resources. Example use cases:

- Agents that are public-facing versus internal
- Delineate between different environments (dev versus prod)

By default, resources are allocated with the * role

See Mesos docs for further information:

http://mesos.apache.org/documentation/latest/attributes-resources/

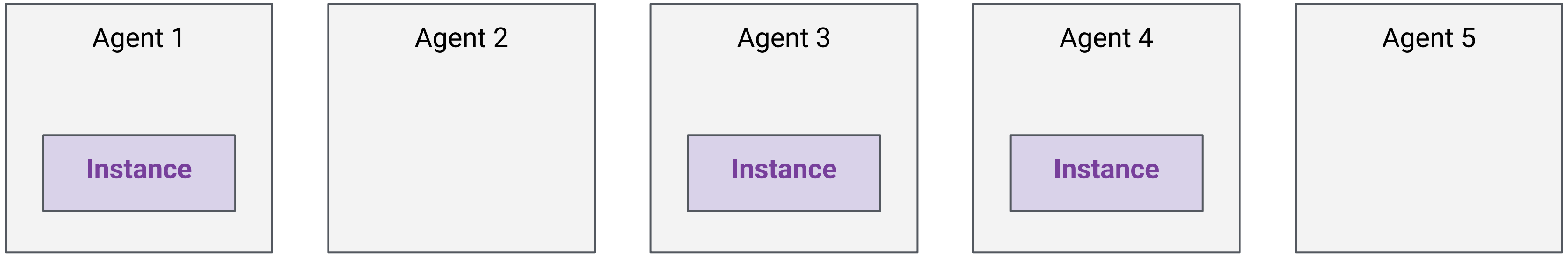
Operators

Operator	Description
`UNIQUE`	All instances of the service must have a unique value for this attribute (e.g. don't allow multiple instances of a service to run on a single agent)
`GROUP_BY`	All service instances will be 'grouped' evenly around this attribute, according to the number of groupings specified. This can be thought of as similar to the SQL 'GROUP BY' clause, with a limit (e.g. spread evenly by rack)
`CLUSTER`	All service instances will be run on agents that share this attribute (e.g. only run on agents with SSDs)
`LIKE`	A regular expression constraint that is applied to the attribute's value. Service instances will only run on agents that match the pattern (e.g. only run on m4.2xlarge and m4.4xlarge)
`UNLIKE`	Opposite of LIKE operator, avoids running workloads on certain agents. (e.g. don't run on m4.2xlarge and m4.4xlarge)
`MAX_PER`	Specify a maximum of instances for a given service that can run on a single agent

Operator: UNIQUE

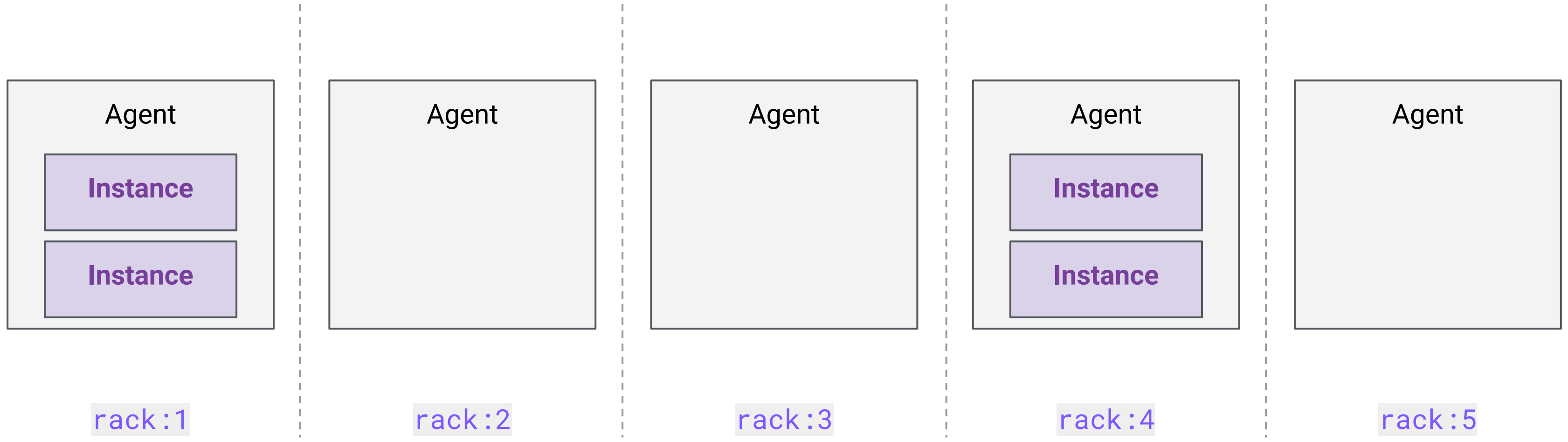
```
“constraints”: [[“hostname”, “UNIQUE”]],
```

```
“instances”: 3
```



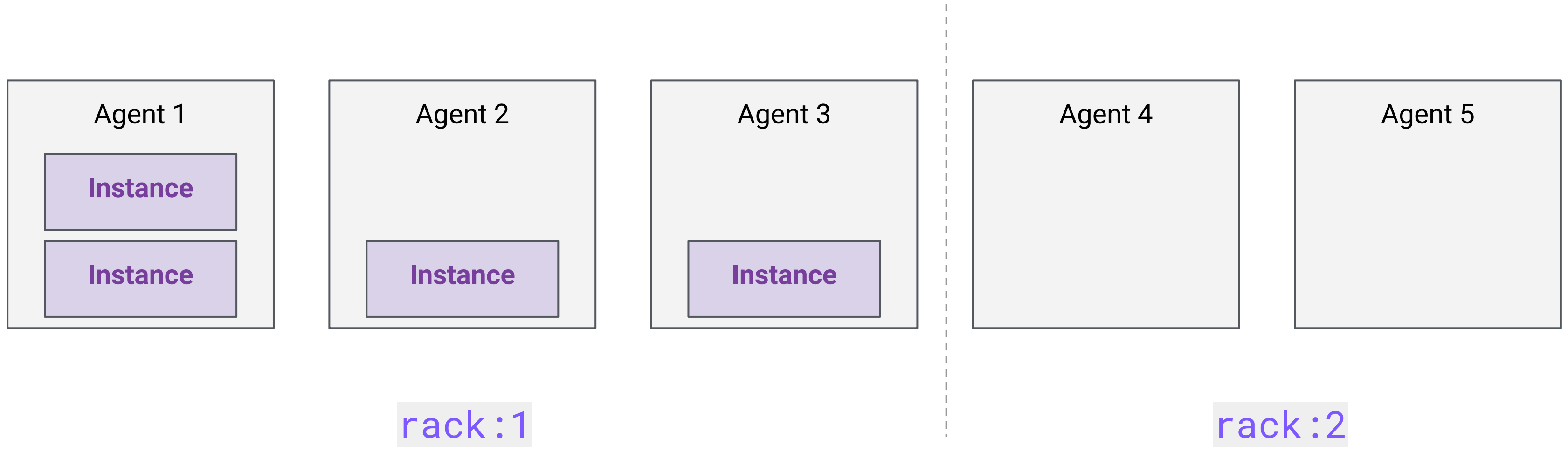
Operator: GROUP_BY

```
“constraints”: [[“rack”, “GROUP_BY”, 2]],
“instances”: 4
```



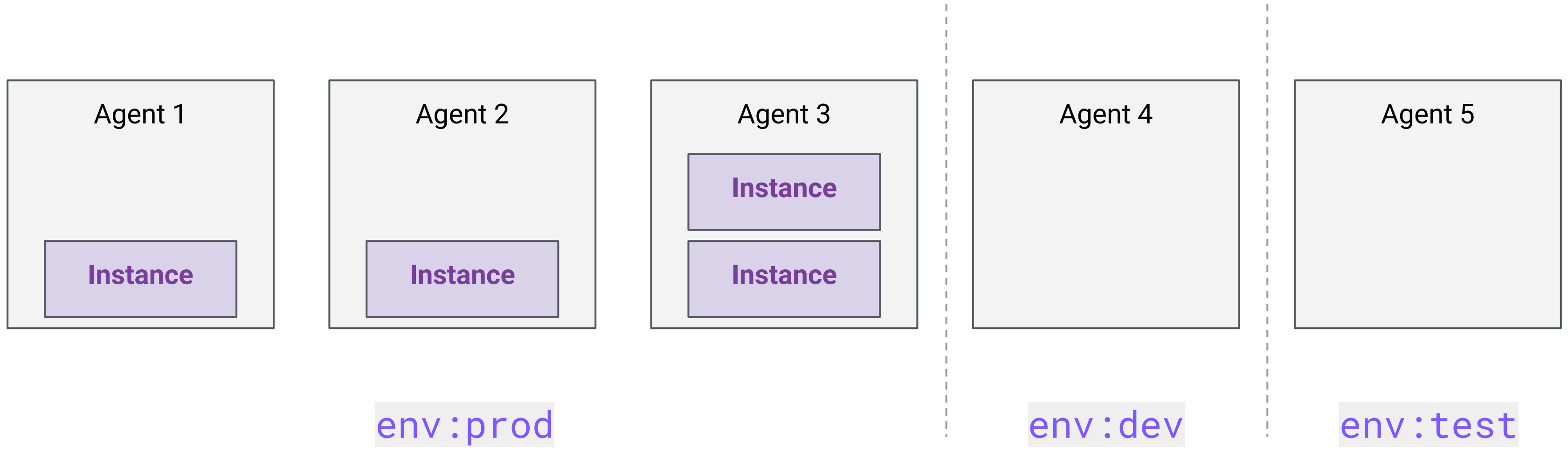
Operator: CLUSTER

```
“constraints”: [[“rack”, “CLUSTER”, “1”]],
“instances”: 4
```



Operator: LIKE

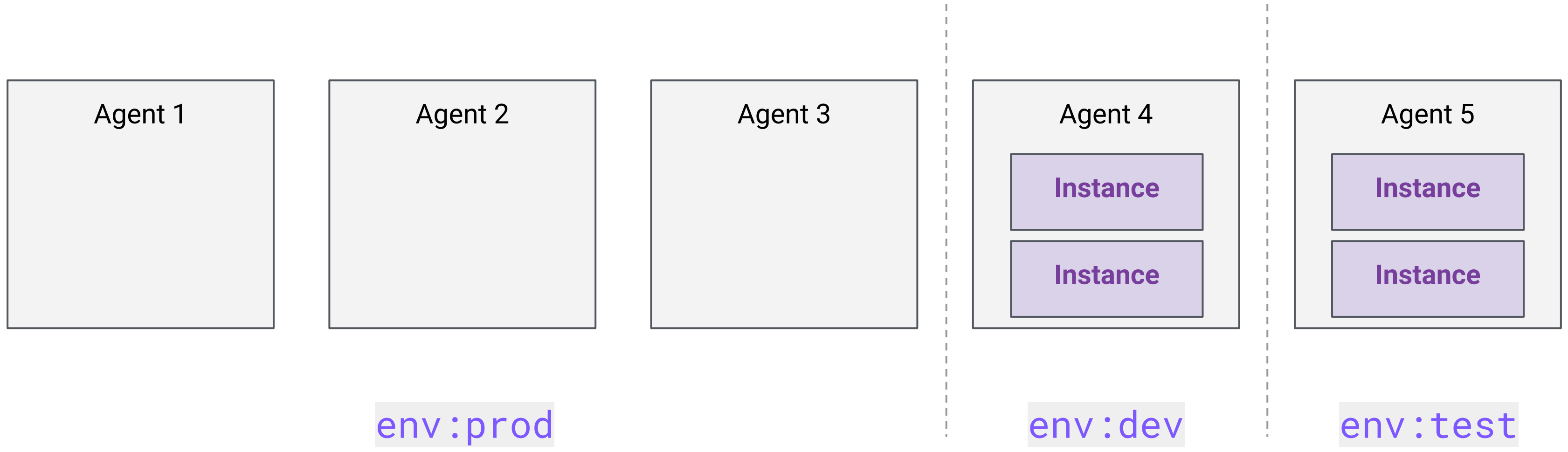
```
“constraints”: [[“env”, “LIKE”, “[Pp]rod”]],
“instances”: 4
```



Operator: UNLIKE

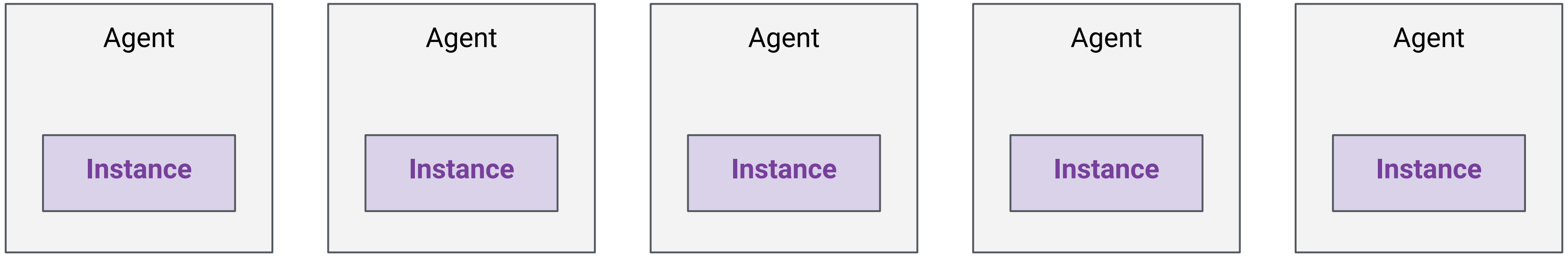
```
“constraints”: [[“env”, “UNLIKE”, “[Pp]rod”]],
```

```
“instances”: 4
```



Operator: MAX_PER

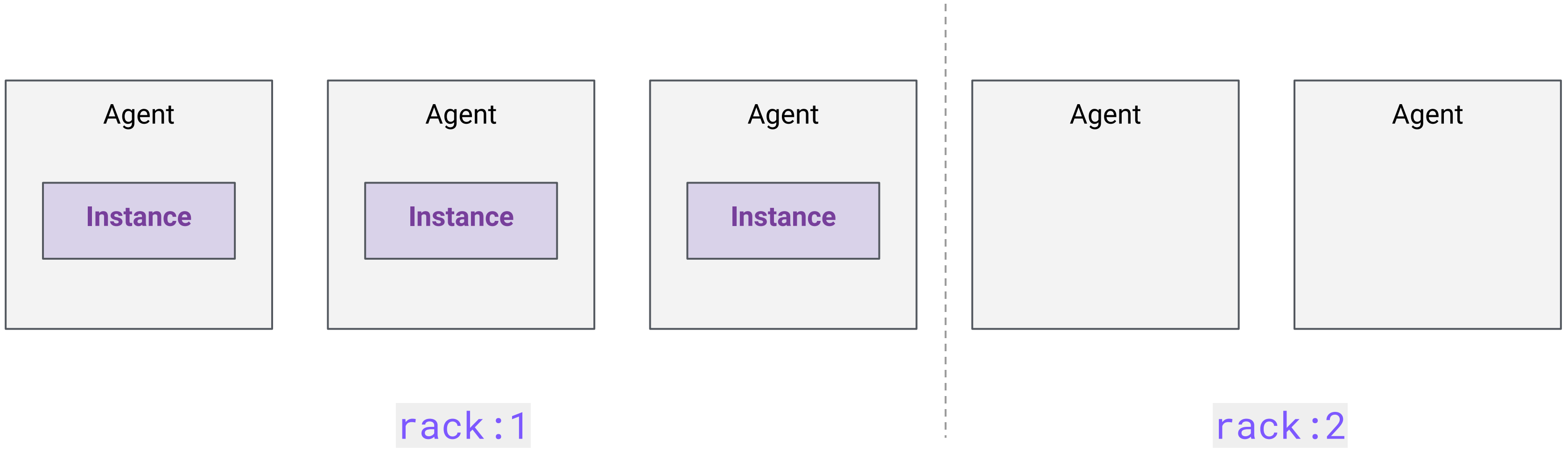
```
“constraints”: [[“hostname”, “MAX_PER”, “1”]],
“instances”: 10
```



Combining Constraints

```
“constraints”: [[“hostname”, “UNIQUE”], [“rack”, “CLUSTER”, “1”]],
```

```
“instances”: 3
```



Lab 7

Placement Constraints

Health Checks

Overview

- Health checks are executed by the container executor that is running on the host where a container runs
- Multiple different types of health checks are supported:
  - Command based health checks where the exit code is used to determine the health
  - HTTP(s) based health checks where a return code between 200 and 399 is considered healthy
  - TCP based health checks where a successfully socket creation is considered healthy
- Interpretation of health checks is left up to the scheduler - in the case of Marathon, a task which fails its health check will be killed and respawned
- Data from health checks may be consumed by external systems in order to collect availability metrics

Health Check Parameters

Parameter	Description
`protocol`	(Default: <>) Protocol of the requests to be performed. One of "`command`", "`MESOS_HTTP`", "`MESOS_HTTPS`", "`MESOS_TCP`" are supported.
`path`	(Default: "/") Path to the endpoint to query
`portIndex`	(Default: 0) Index in the service's `portMappings`/`portDefinitions` to send query to
`portName`	Name of the port in the service's `portMappings`/`portDefinitions` to send query to
`timeoutSeconds`	(Default: 20) Number of seconds after which a health check is considered a failure regardless of the response
`gracePeriodSeconds`	(Default: 15) Health check failures are ignored within this number of seconds of the task being started or until the task becomes healthy for the first time
`intervalSeconds`	(Default: 10) Number of seconds to wait between health check queries
`maxConsecutiveFailures`	(Default: 3) Number of consecutive health check failures after which the unhealthy task should be killed

Readiness Checks

- Readiness checks are similar to health checks - but they run **BEFORE** health checks
  - Way to test that an application has finished initializing before health checks start
  - Prevents slow starting applications from being marked as unhealthy before they have enough time to complete initialization
  - Service will be stuck in a Deploying state until Readiness Checks succeed
- Documentation

Readiness Check Parameters

Parameter	Description
`protocol`	(Default: `HTTP`) Protocol of the requests to be performed. One of “`HTTP`”, or “`HTTPS`”.
`path`	(Default: “/”) Path to the endpoint to query
`portName`	Name of the port in the service’s `portMappings`/`portDefinitions` to send query to
`timeoutSeconds`	(Default: 20) Number of seconds after which a readiness check times out, regardless of the response. This value must be smaller than `intervalSeconds`.
`httpStatusCodesForReady`	(Default: [200]) The HTTP/HTTPS status code to treat as ready.
`intervalSeconds`	(Default: 10) Number of seconds to wait between health check queries
`maxConsecutiveFailures`	(Default: 3) Number of consecutive health check failures after which the unhealthy task should be killed
`preserveLastResponse`	(Default: false) If true, the last readiness check response will be preserved and exposed in the API as part of a deployment.

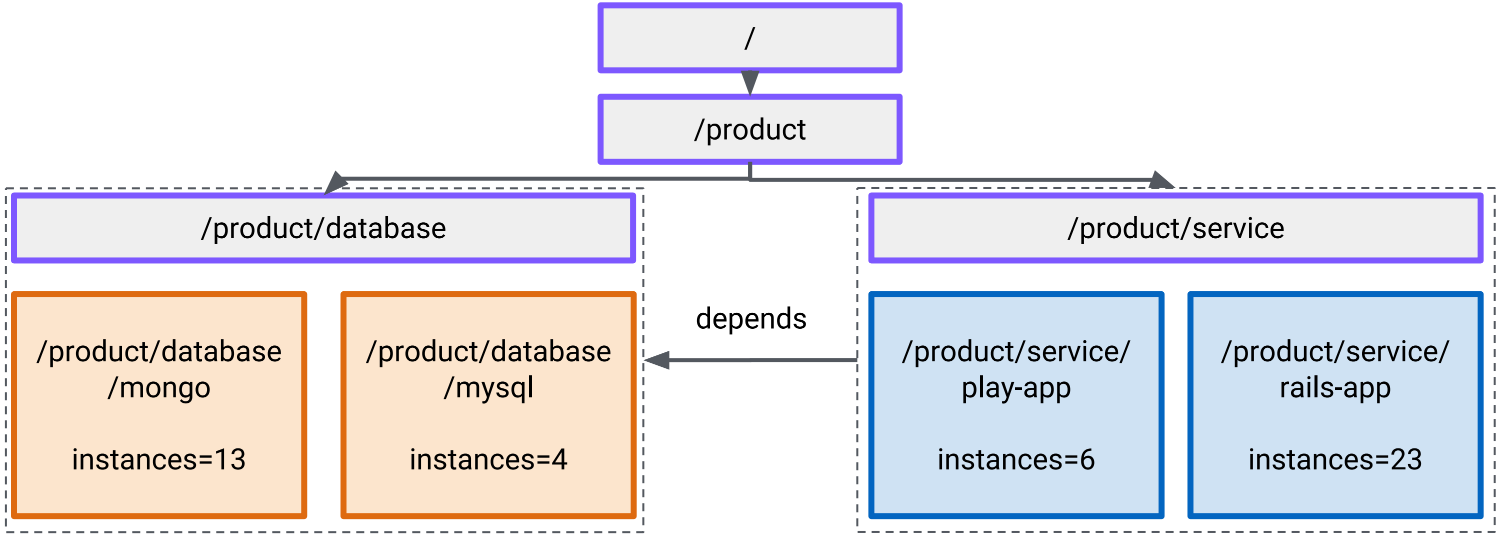
Lab 8

Health Checks

Application Groups

Overview

- Application groups provide an abstraction for groups of applications
- Multiple apps can be partitioned into sets of apps for ease of management
- Some applications have dependencies and need to deployed with ordering
- Interdependent apps can often be modeled as DAGs



Example Group Spec

```
{
     "id": "/product",
     "groups": [{
          "id": "/product/database",
          "apps": [
               { "id": "/product/database/mongo", …},
               { "id": "/product/database/mysql", …}
          ]
     },{
          "id": "/product/service",
          "dependencies": ["/product/database"],
          "apps": [
               { "id": "/product/service/rails-app", …},
               { "id": "/product/service/play-app", …}
          ]
     } ]
}
```

Use Cases

- Allow for multiple layers of groups of applications including common service(s)
- Provide mechanism for intelligent scaling of all, or a subset of applications in a group
- Scaling factor is a multiplier of current deployed instances:
    - To double instances within a group:
        - `dcos marathon group scale <service_id> 2`
    - To reduce the number of instances within a group by half:
        - `dcos marathon group scale <service_id> 0.5`
- Restrict users and groups from creating, viewing, or destroying applications in a group

Pods

Overview

- Grouping of containers
- Each container in a pod shares storage and networking
- Each container in a pod is co-located on the same agent node
- Represented as a single service in Marathon
- Useful for migrating legacy workloads to a containerized platform
- Only supported through UCR

Use Case

- Will this application work on a different host from this other app?
    - e.g.
        - Will Wordpress work on a different host from its MySQL db?
            - YES
        - Will this log scraper that scrapes at logs on a local filesystem work on a different host from the application that it is scraping logs from?
            - NO

Pod Definition

```
“containers”: [ {
    “name”: “business-app”,
    “resources”: {
         “cpus”: 1.0, “mem”: 256 },
         “image”: {
              “kind”: “DOCKER”,
              “id”: “business-app:1.0”,
              “forcePull”: false
         }
    },{
    “name”: “log-scraper”,
    “resources”: {
         “cpus”: 0.1, “mem”: 16 },
         “exec”: {
              “command”: {“shell”: “...”}
         }
    }
 ...
```

Lab 9

Pods

Rolling Upgrades

Deployment Policies

- Marathon begins a deployment anytime there’s a change to a service specification, which includes:
  - Starting/stopping services
  - Updating service specification
  - Scaling a service
- In the case of updating a service’s specification (which would be used for upgrading and downgrading an application), Marathon can execute a rolling upgrade and leverage health checks to control the process
- Services which are a part of an application group will be deployed in the correct order as specified by their dependencies

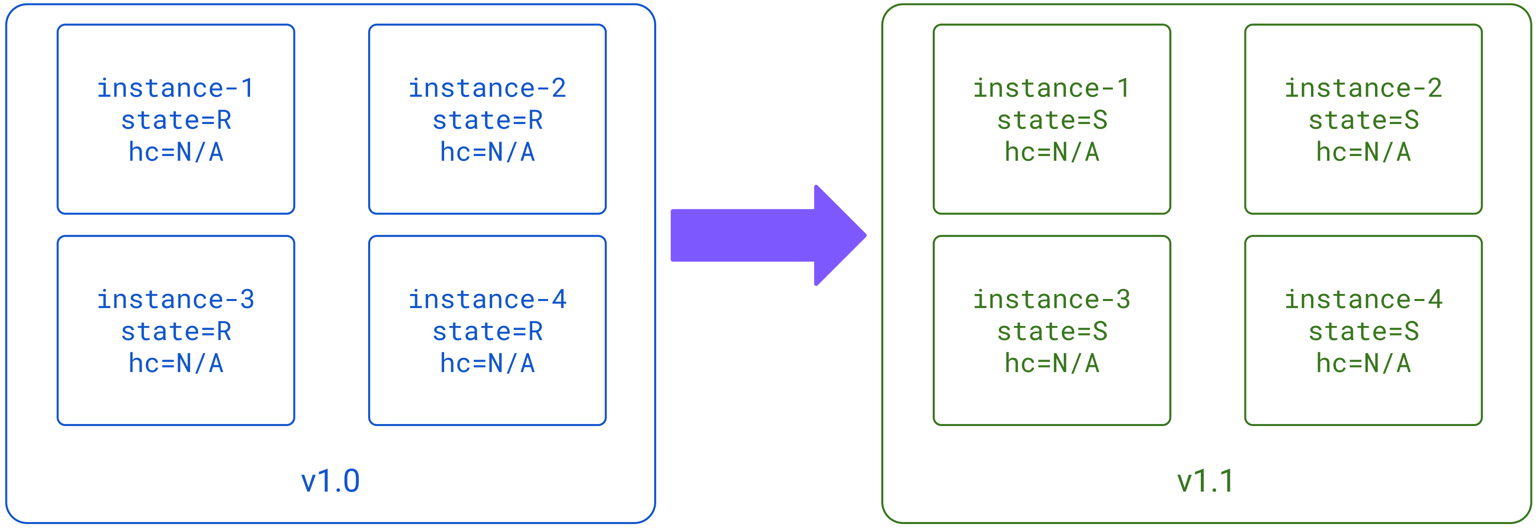
Rolling Upgrades: Strategies & Behaviors

- Default behavior
  - Without health checks
  - With health checks
- Minimal overcapacity
- Blue/Green

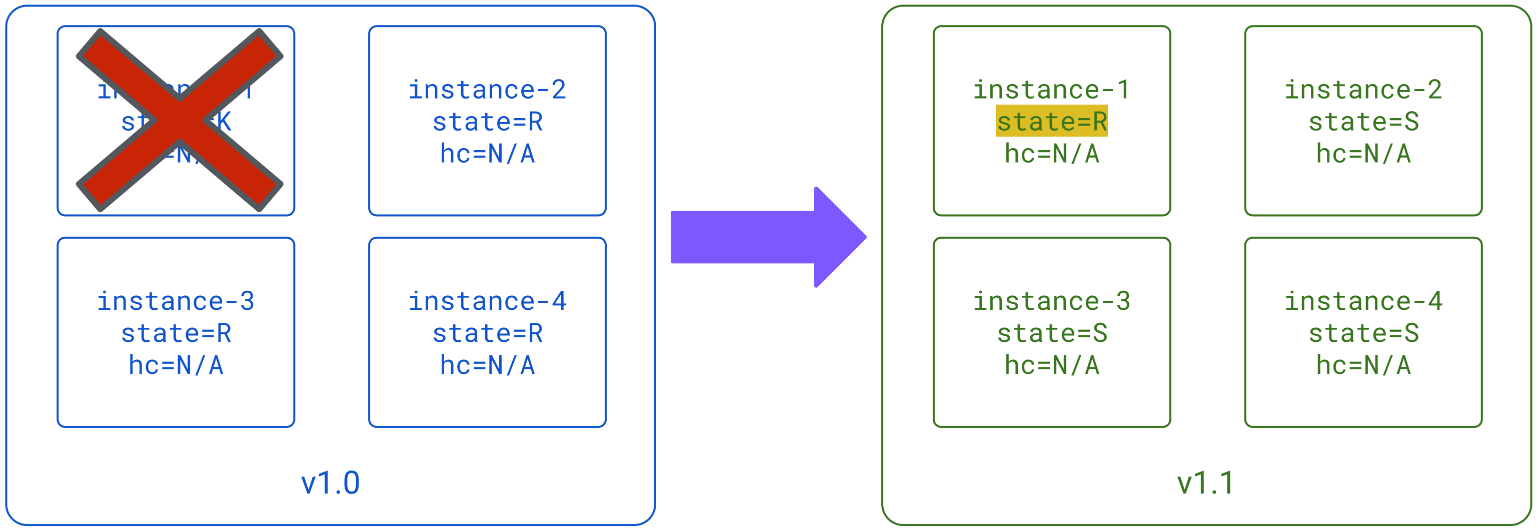
Default: No Health Checks

- All new instances are `staged` and `started`
- As each new instance gets to a `running` state, the oldest instance gets killed
- No guarantee that new instance is fully healthy since no health checks have been defined

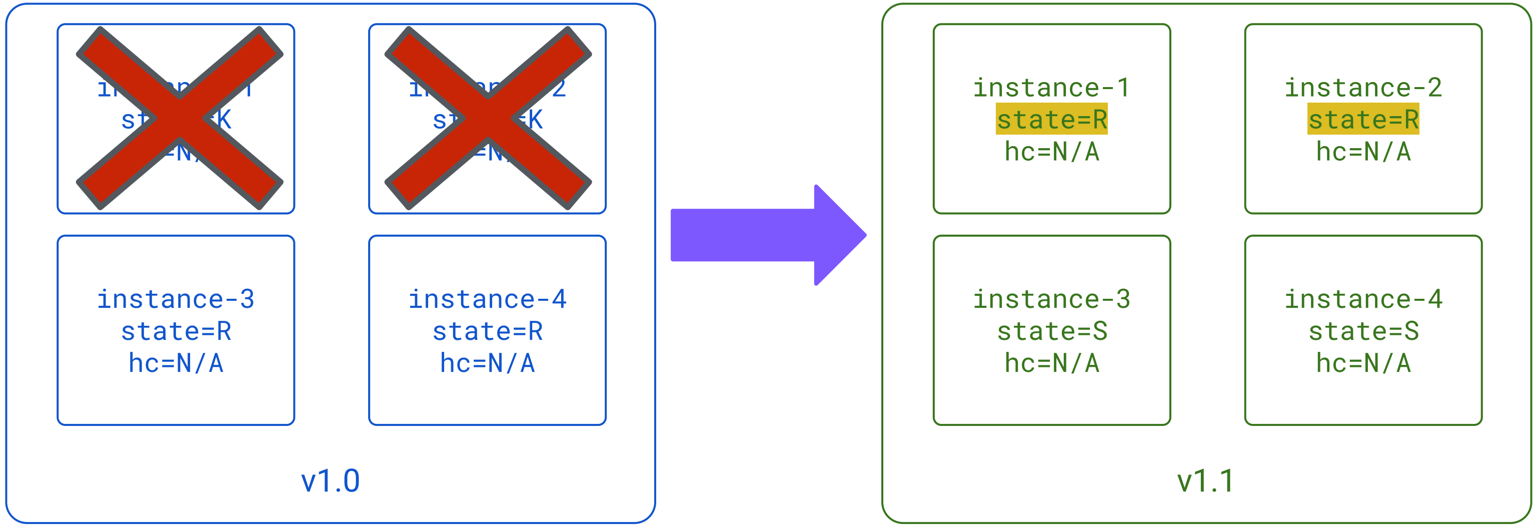
Default: No Health Checks



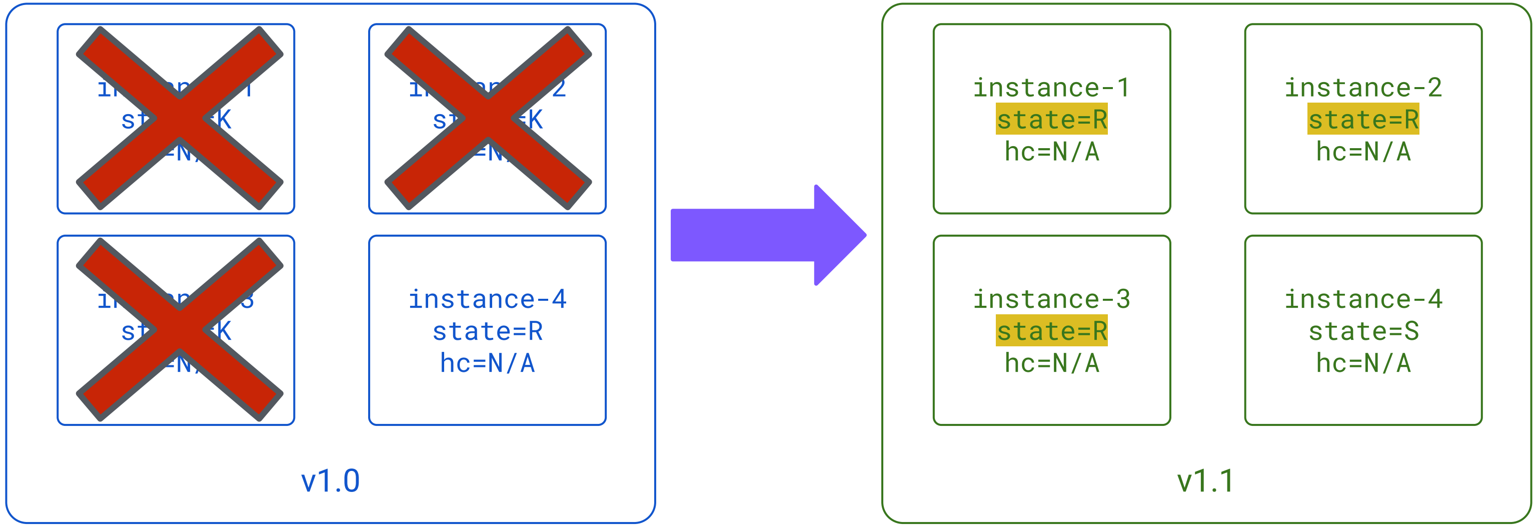
Default: No Health Checks



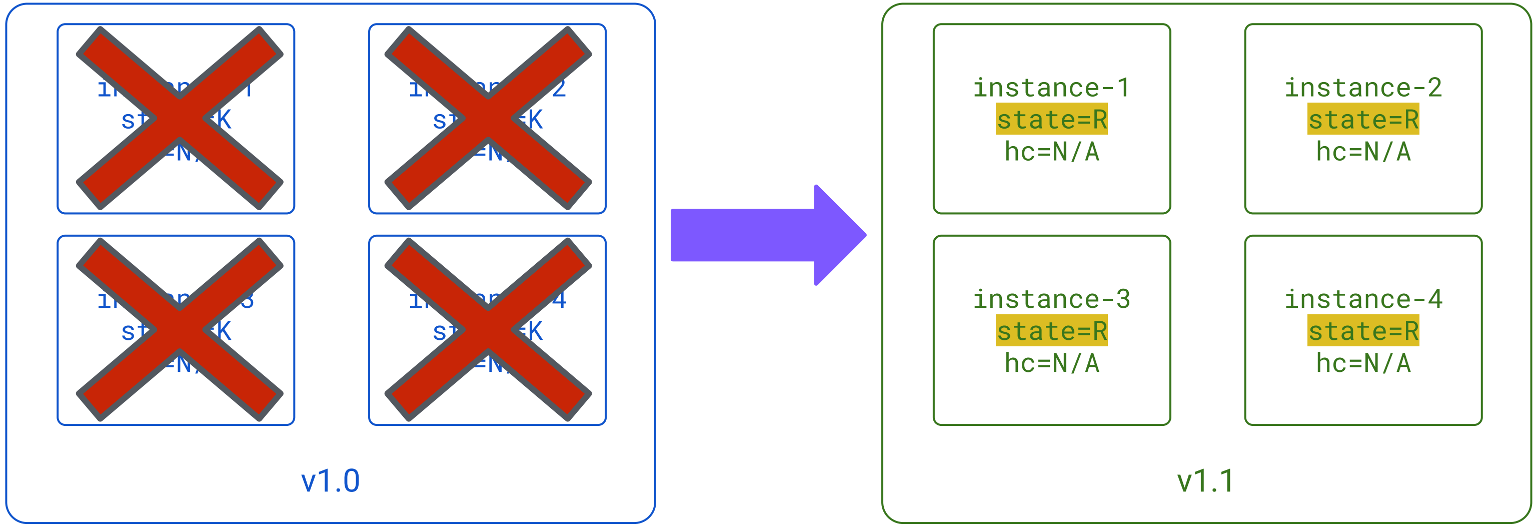
Default: No Health Checks



Default: No Health Checks



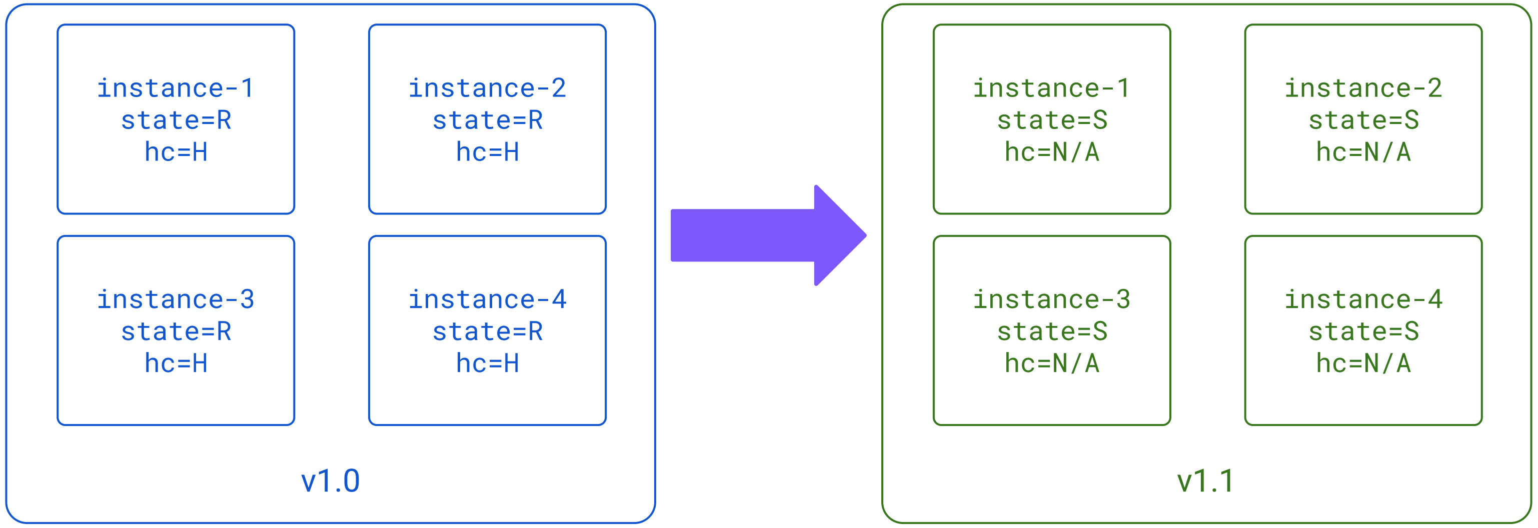
Default: No Health Checks



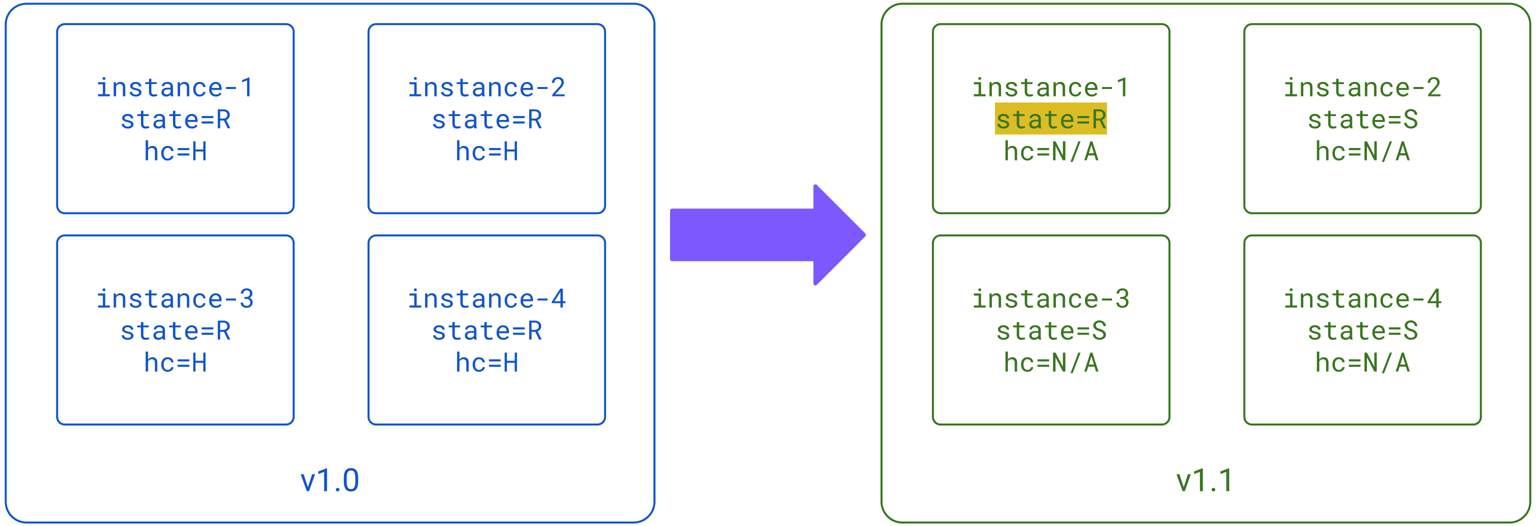
Default: With Health Checks

- All new instances are `staged` and `started`
- As each new instance gets to a `running` and `healthy` state, the oldest instance gets killed
- Requires double the capacity on your cluster while the upgrade is ongoing, for example:
    - Number of instances at old version = 4
    - Number of instances at new version = 4
    - Total number of instances running = 8 (100% over requested instance/resource allotment)

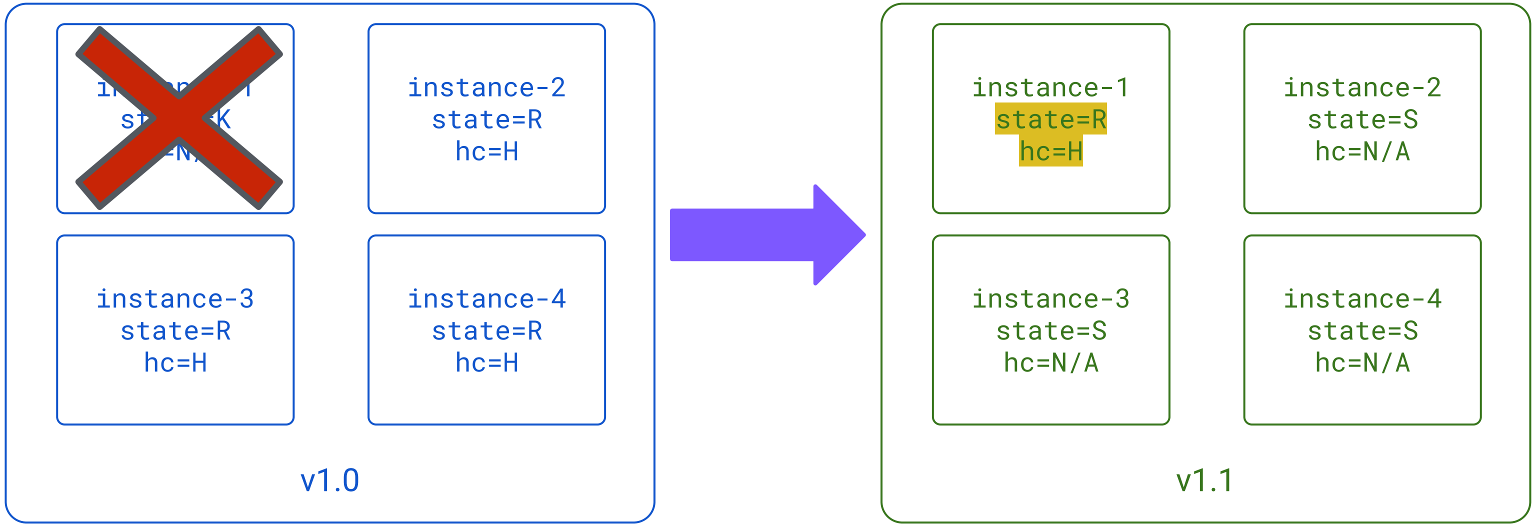
Default: With Health Checks



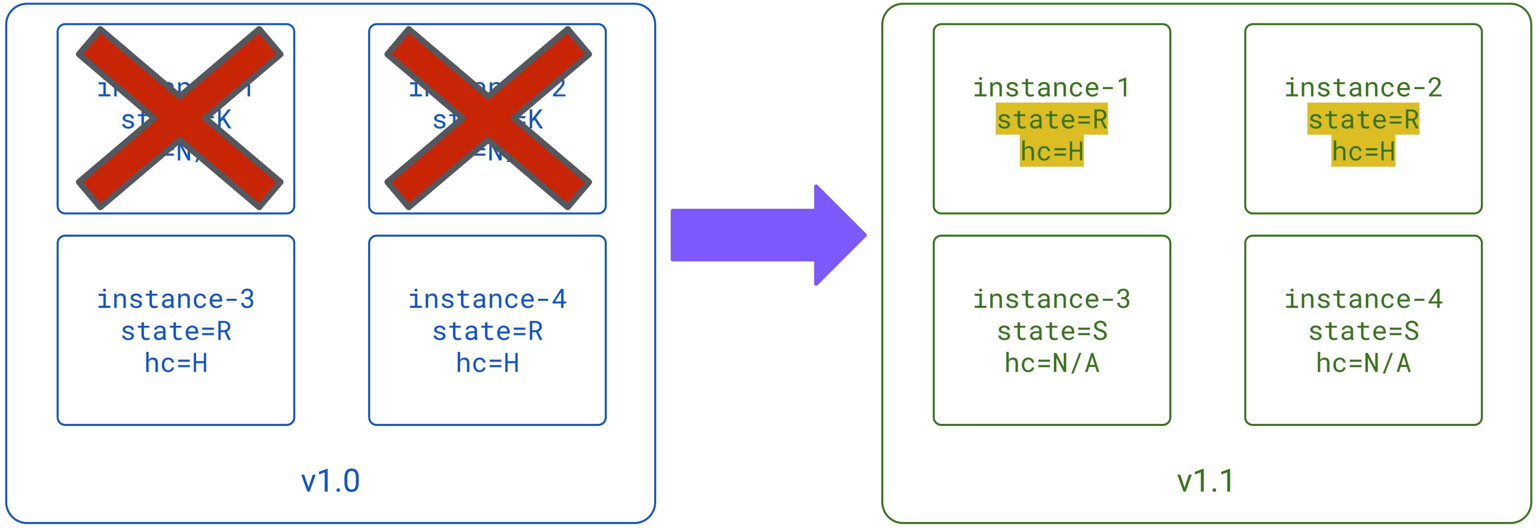
Default: With Health Checks



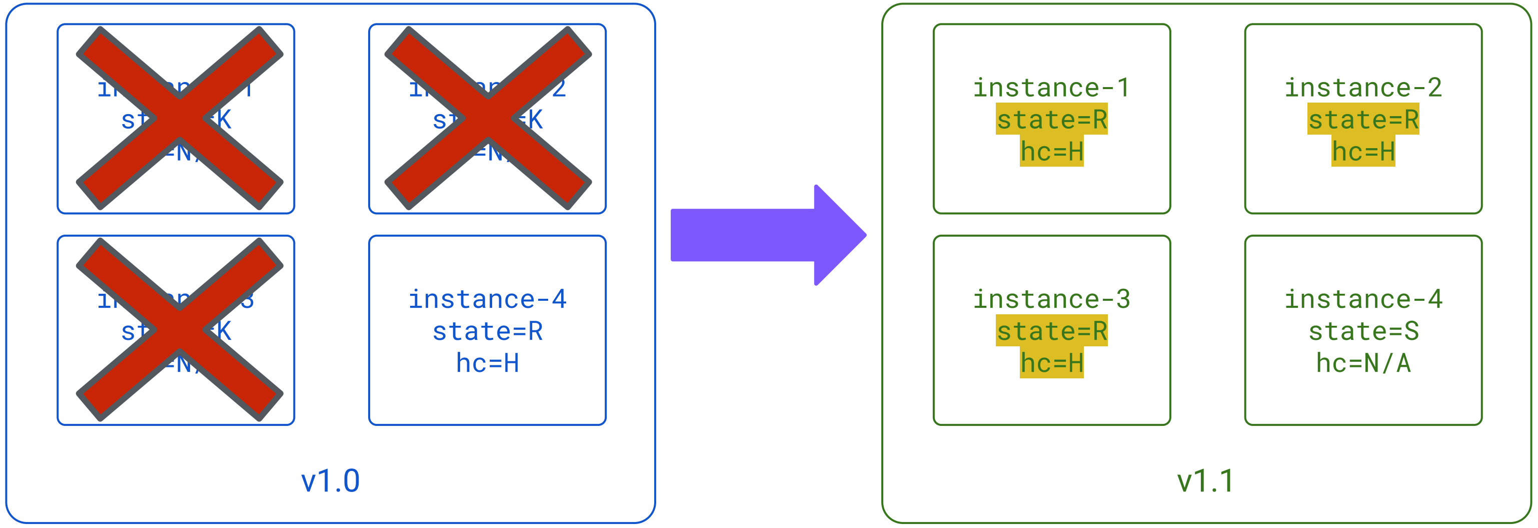
Default: With Health Checks



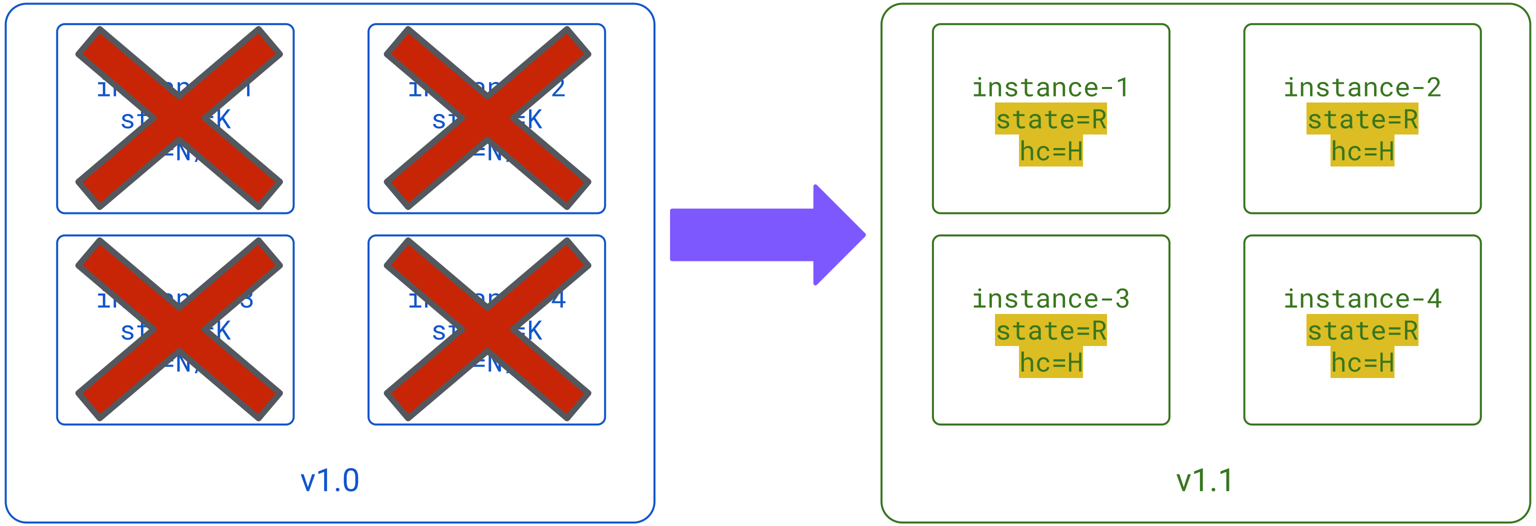
Default: With Health Checks



Default: With Health Checks



Default: With Health Checks



Deployment Policy Configuration

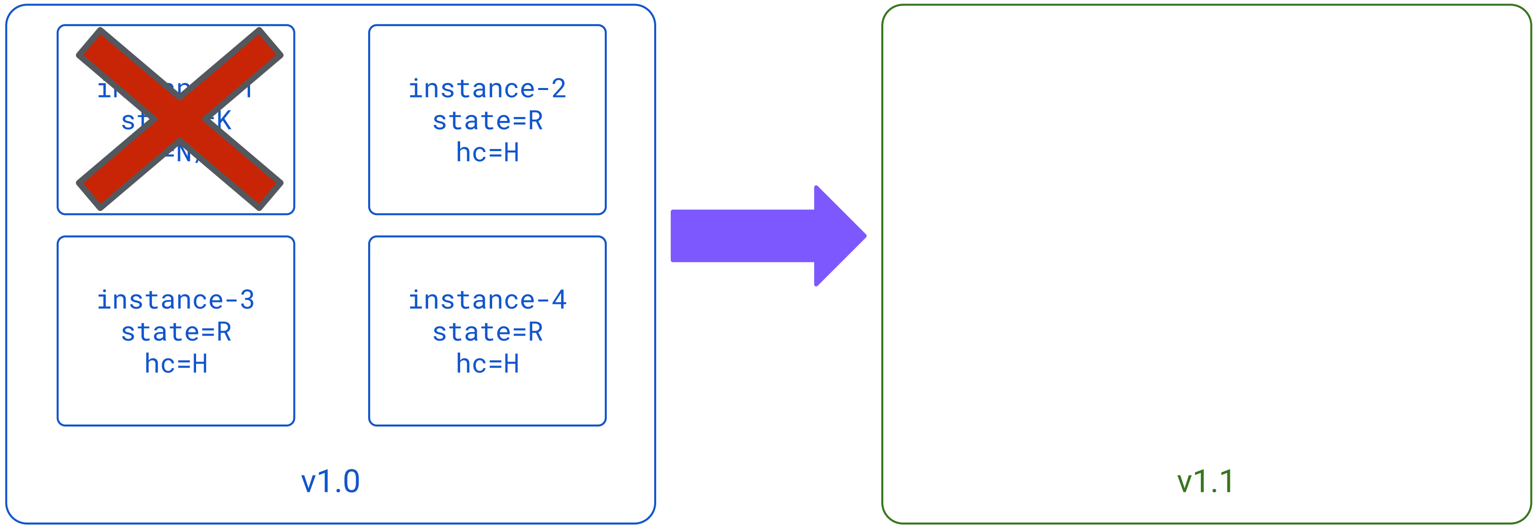
- To perform an upgrade more granularly for scenarios where you may not have sufficient capacity in your cluster to be 100% over your requested instance/resource allotment, there are two parameters that can be placed in your service specification:
    - `minimumHealthCapacity`:
        - A value between 0 and 1 which specifies the percentage of requested instances which must be in a `healthy` state while performing a deployment
    - `maximumOverCapacity`:
        - A value between 0 and 1 which specifies the percentage of requested instances that we can be above during a deployment

Custom: With Health Checks

```
“instances”: 4
“upgradeStategy”: {
      “minimumHealthCapacity”: 0.75,
      “maximumOverCapacity”: 0.25
}
```

```
min. healthy = 0.75 * 4 = 3
max. over = 0.25 * 4 = 1 (5 instances)
```

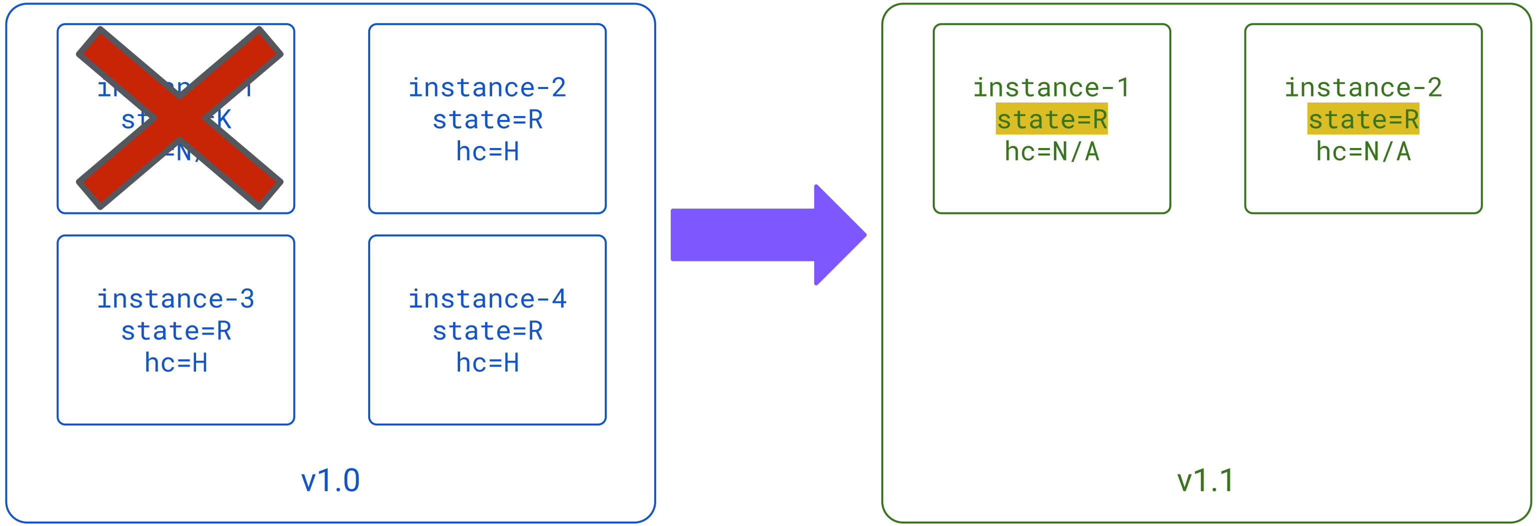
Custom: With Health Checks



```
"instances": 4
"upgradeStategy": {
      "minimumHealthCapacity": 0.75,
      "maximumOverCapacity": 0.25
}
```

```
min. healthy = 0.75 * 4 = 3
max. over = 0.25 * 4 = 1
```

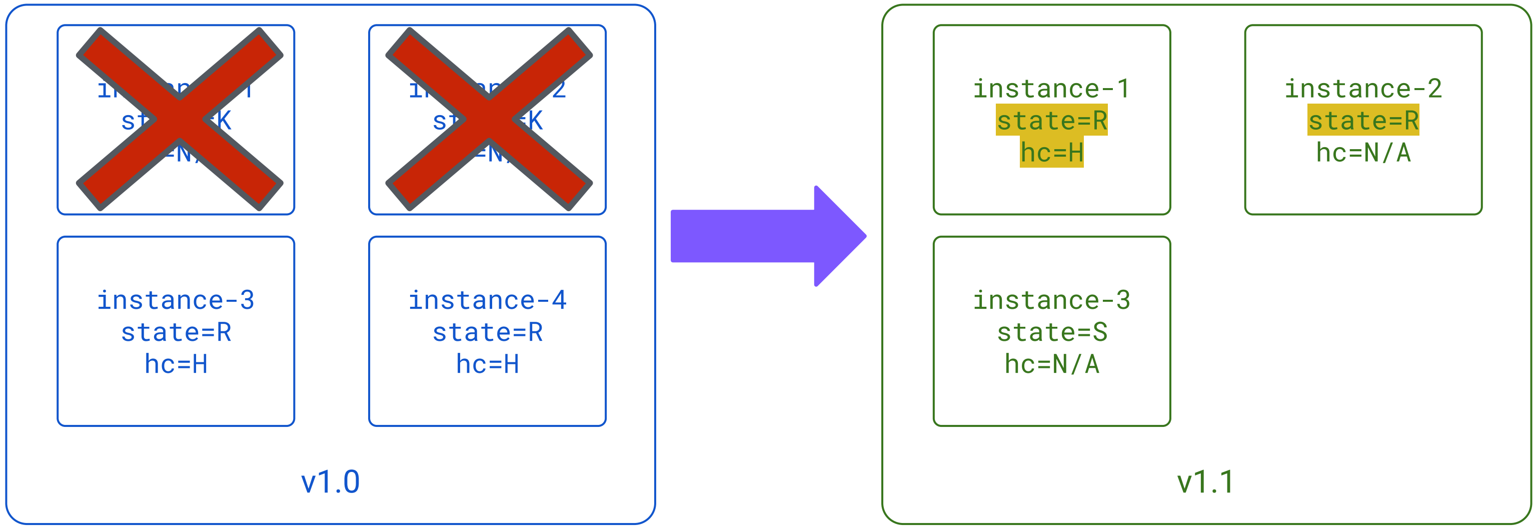
Custom: With Health Checks



```
"instances": 4
"upgradeStategy": {
        "minimumHealthCapacity": 0.75,
        "maximumOverCapacity": 0.25
}
```

```
min. healthy = 0.75 * 4 = 3
max. over = 0.25 * 4 = 1
```

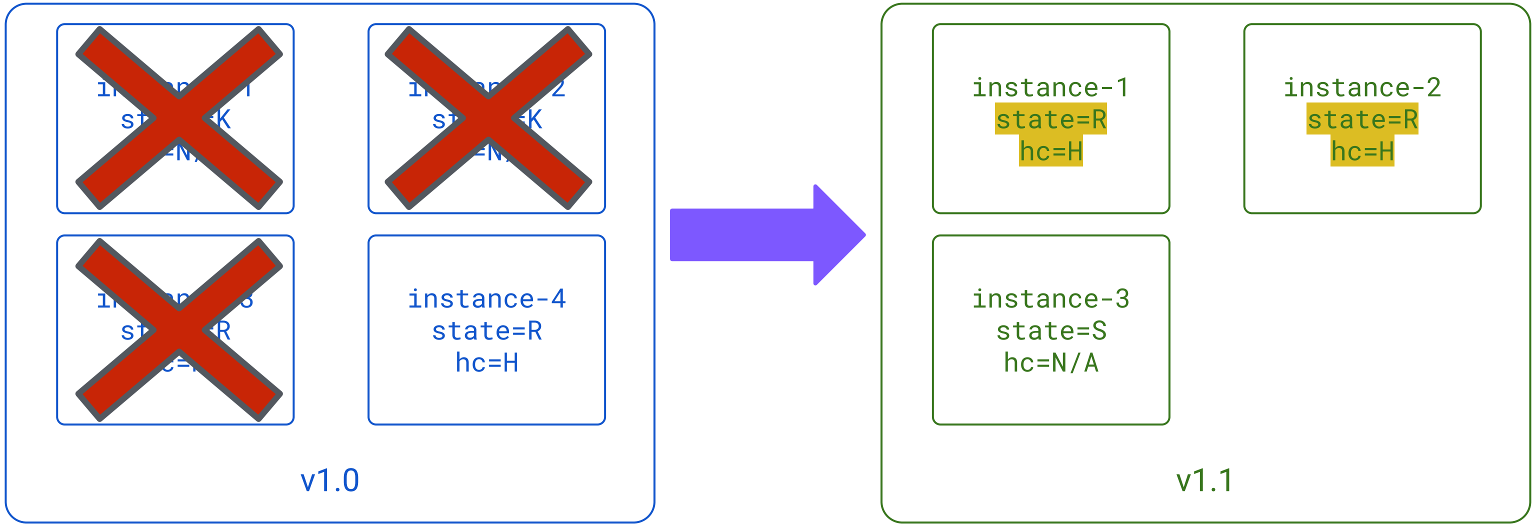
Custom: With Health Checks



```
“instances”: 4
“upgradeStategy”: {
        “minimumHealthCapacity”: 0.75,
        “maximumOverCapacity”: 0.25
}
```

```
min. healthy = 0.75 * 4 = 3
max. over = 0.25 * 4 = 1
```

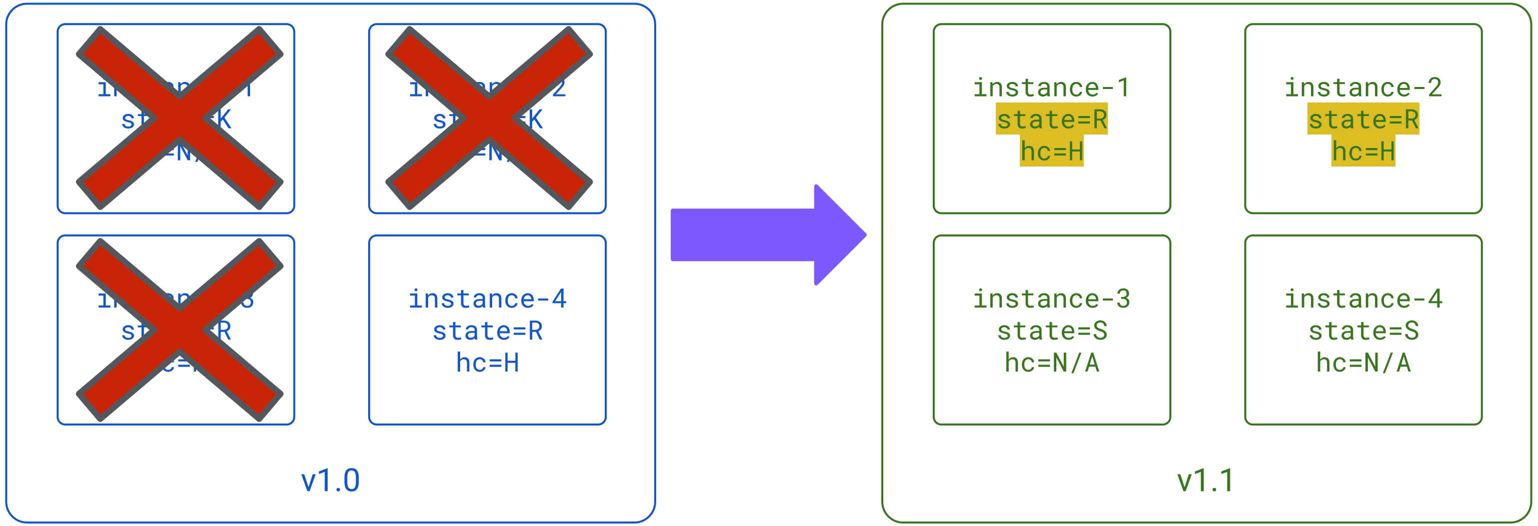
Custom: With Health Checks



```
"instances": 4
"upgradeStategy": {
      "minimumHealthCapacity": 0.75,
      "maximumOverCapacity": 0.25
}
```

```
min. healthy = 0.75 * 4 = 3
max. over = 0.25 * 4 = 1
```

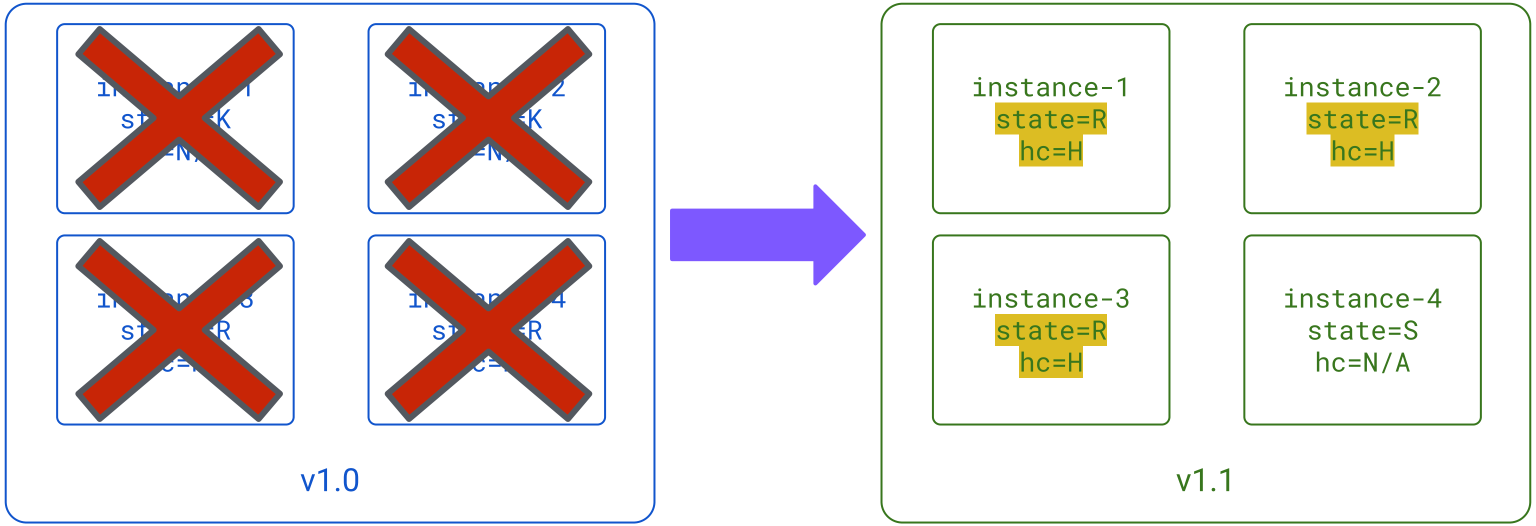
Custom: With Health Checks



```
“instances”: 4
“upgradeStategy”: {
      “minimumHealthCapacity”: 0.75,
      “maximumOverCapacity”: 0.25
}
```

```
min. healthy = 0.75 * 4 = 3
max. over = 0.25 * 4 = 1
```

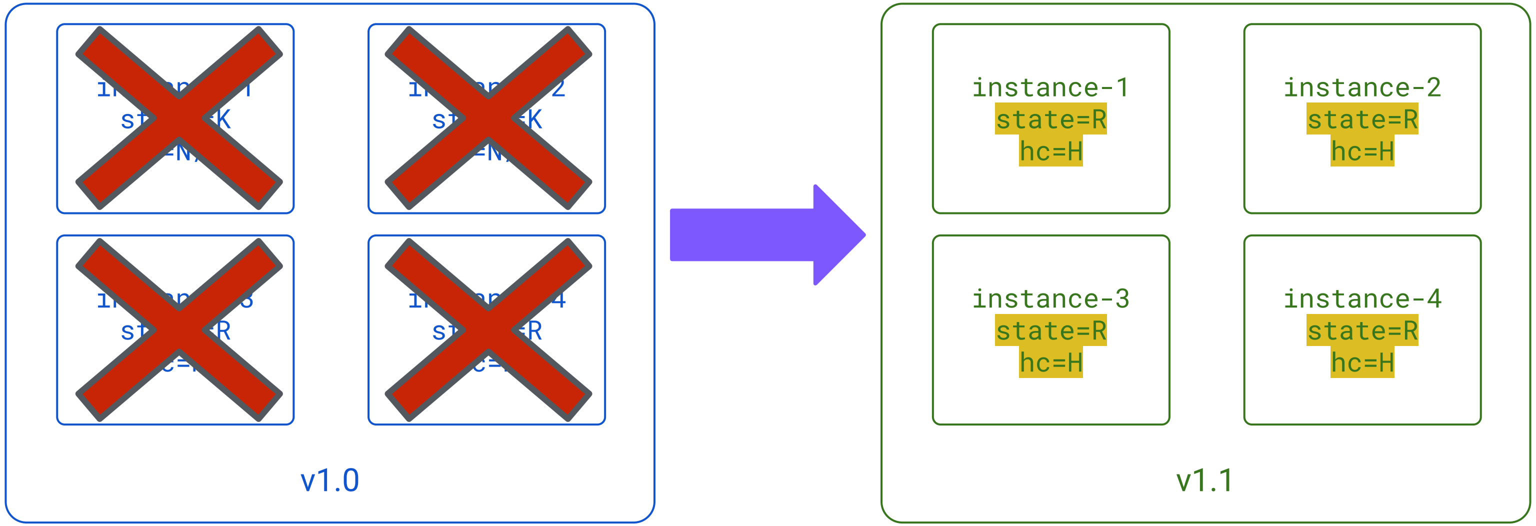
Custom: With Health Checks



```
“instances”: 4
“upgradeStategy”: {
      “minimumHealthCapacity”: 0.75,
      “maximumOverCapacity”: 0.25
}
```

```
min. healthy = 0.75 * 4 = 3
max. over = 0.25 * 4 = 1
```

Custom: With Health Checks



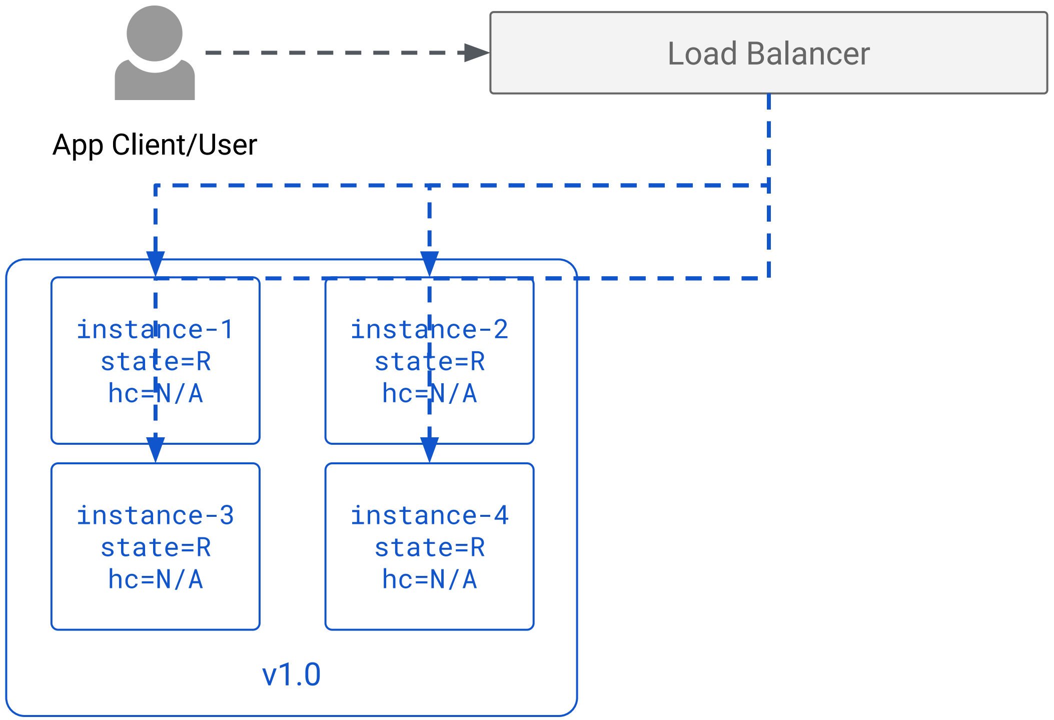
```
"instances": 4
"upgradeStategy": {
      "minimumHealthCapacity": 0.75,
      "maximumOverCapacity": 0.25
}
```

```
min. healthy = 0.75 * 4 = 3
max. over = 0.25 * 4 = 1
```

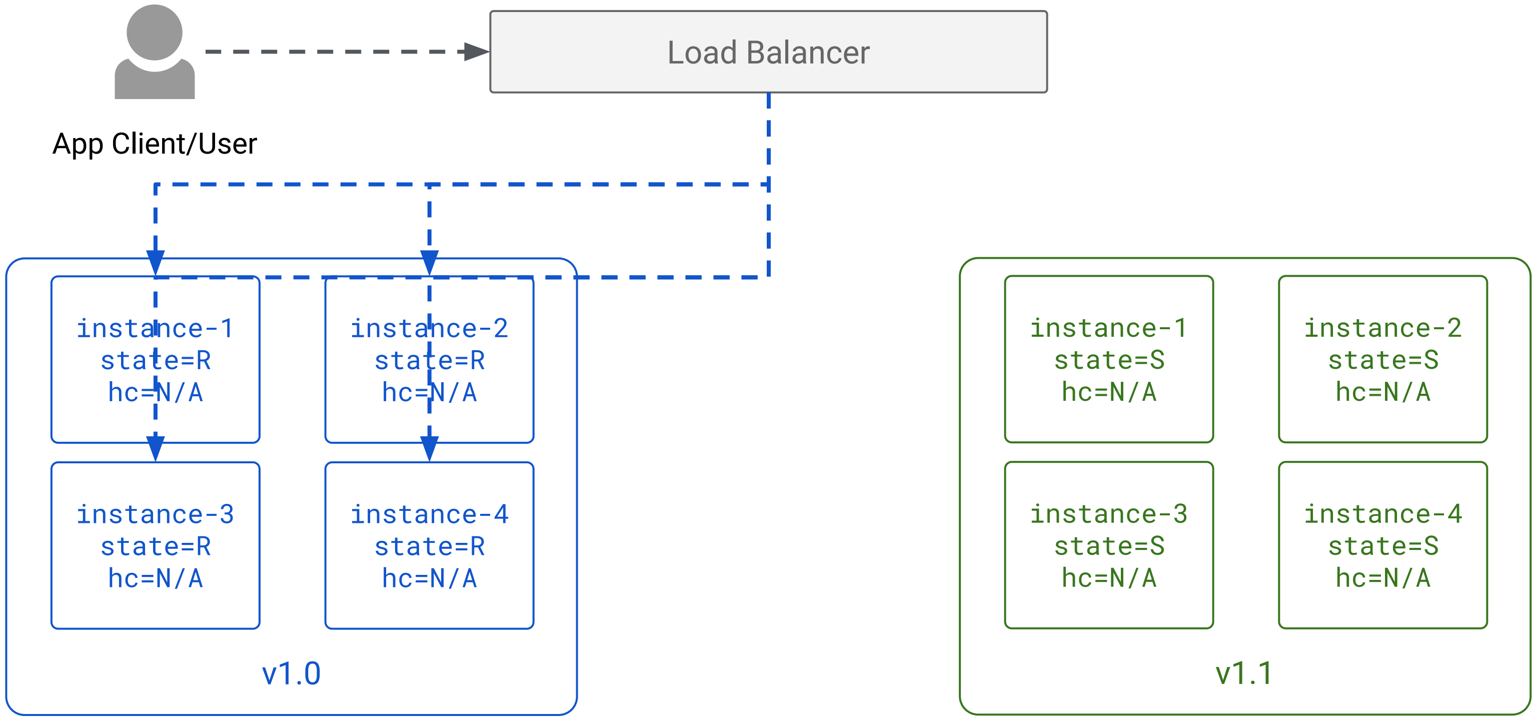
Blue/Green Upgrades

- Two separate services are created, each of which is running a different version of your application
  - One deployment is currently production
  - One deployment will be production
- Access to what is currently considered production is done through an external load balancer
  - In the context of DC/OS, the load balancer can be provided by either Marathon-LB or Edge-LB
- After the to-be production service has been launched and tested by an administrator, the load balancer is re-configured to direct traffic to the new version
- Old version can be kept around in case a flip-over is required due to failures in the new version
- Process repeats when the next version is ready to be rolled out

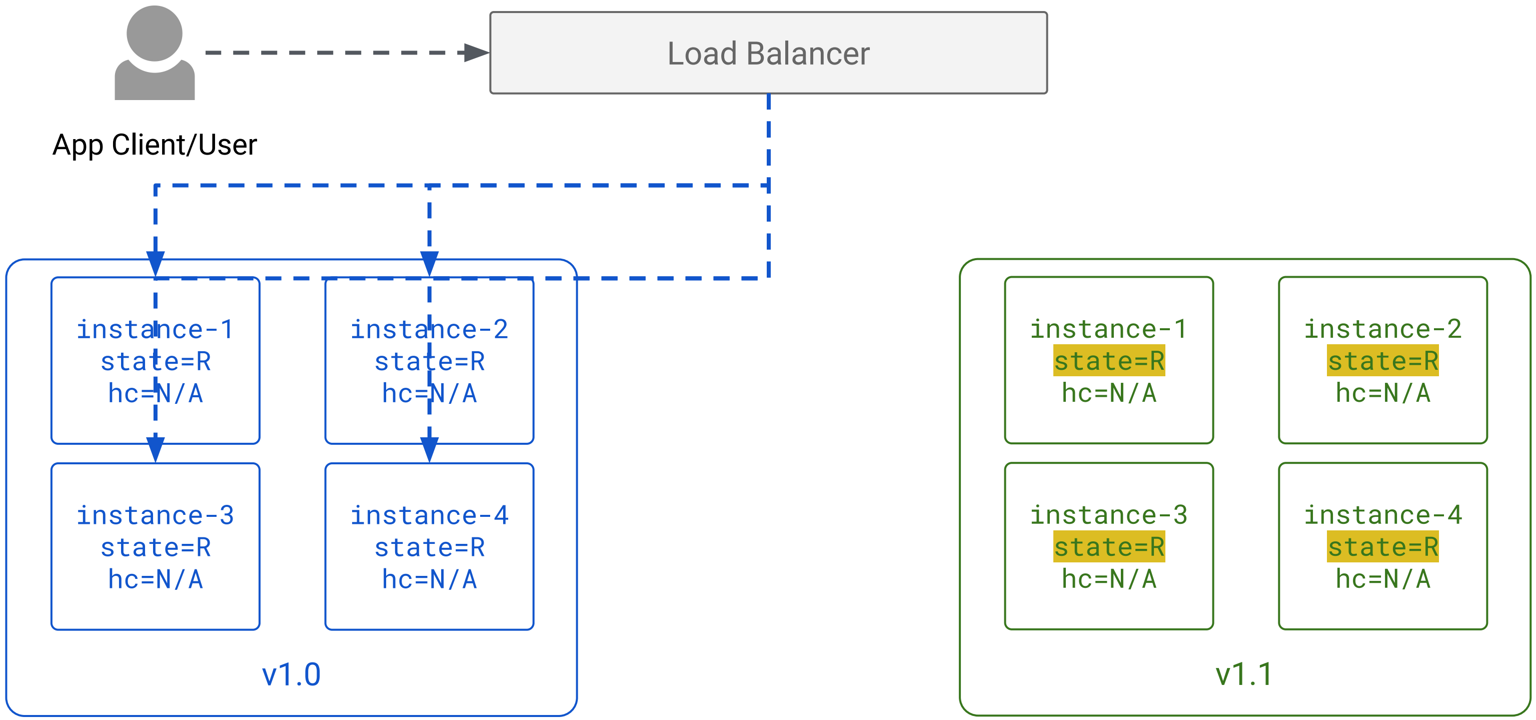
Blue/Green Upgrades



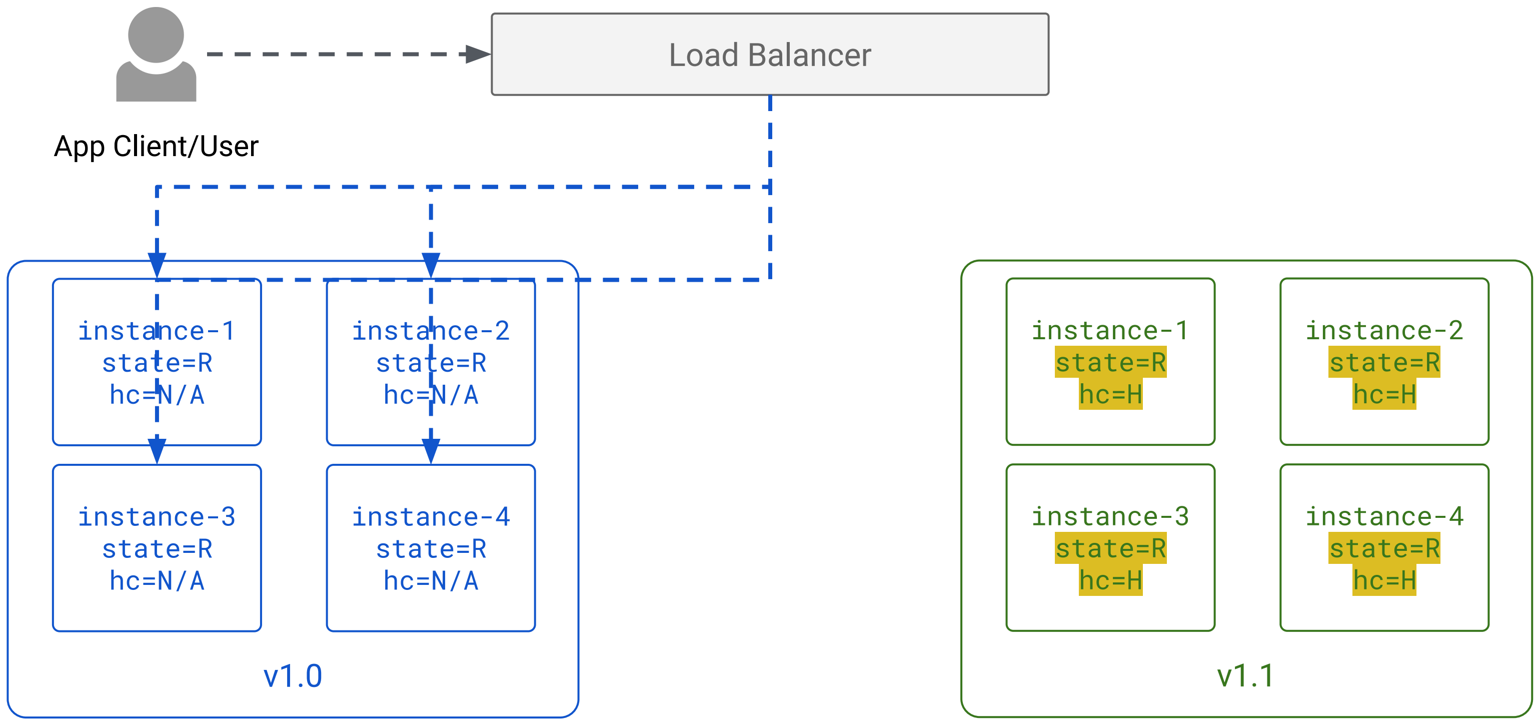
Blue/Green Upgrades


Load Balancer
App Client/User
instance-1 state=R hc=N/A
instance-2 state=R hc=N/A
instance-3 state=R hc=N/A
instance-4 state=R hc=N/A
v1.0
instance-1 state=S hc=N/A
instance-2 state=S hc=N/A
instance-3 state=S hc=N/A
instance-4 state=S hc=N/A
v1.1

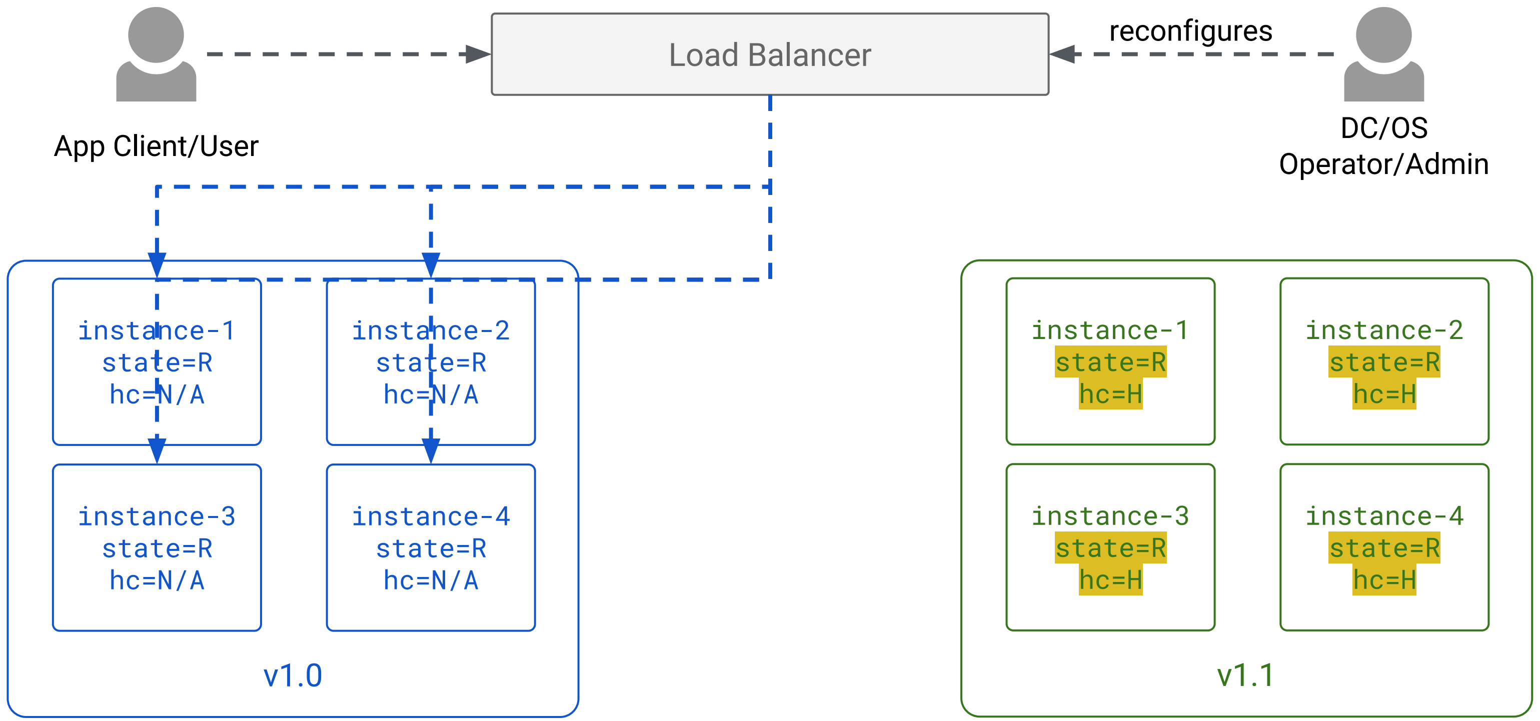
Blue/Green Upgrades



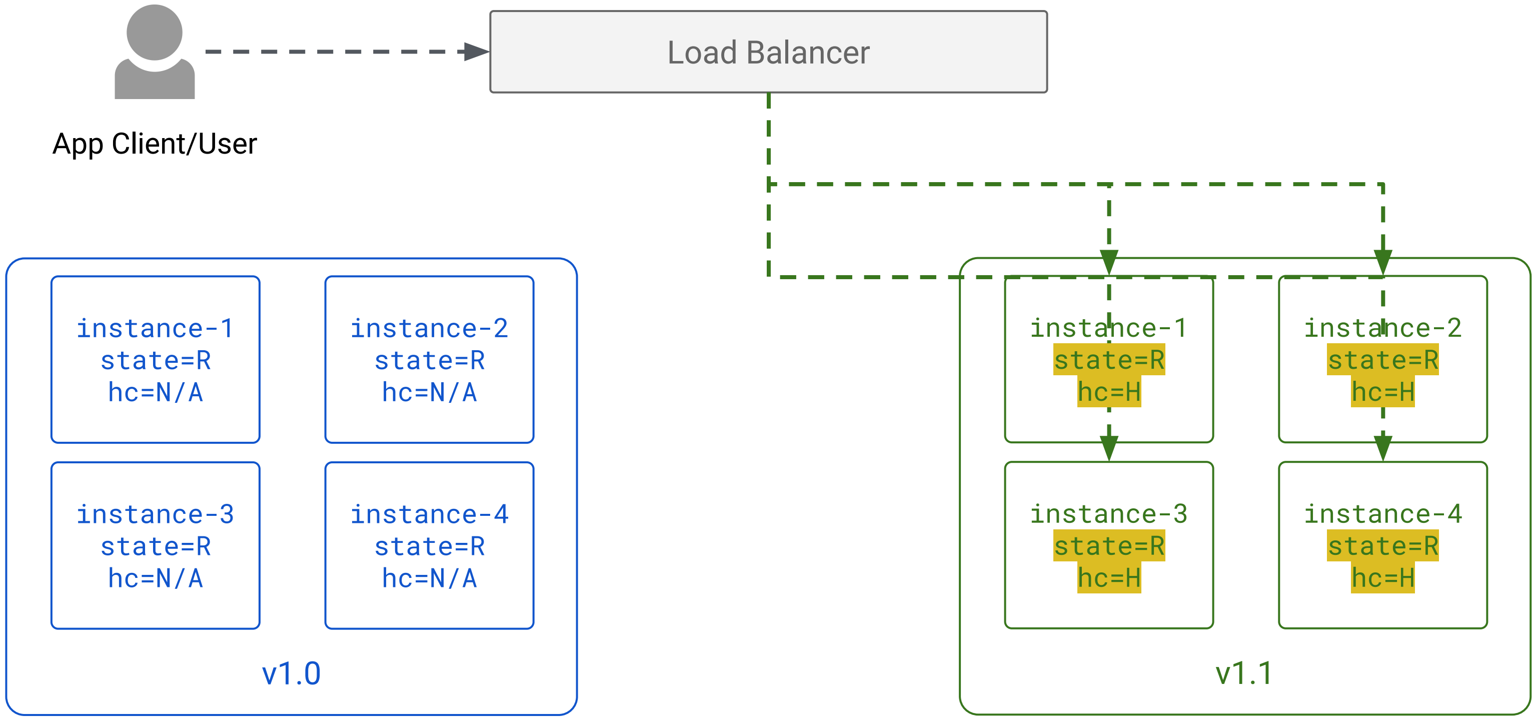
Blue/Green Upgrades



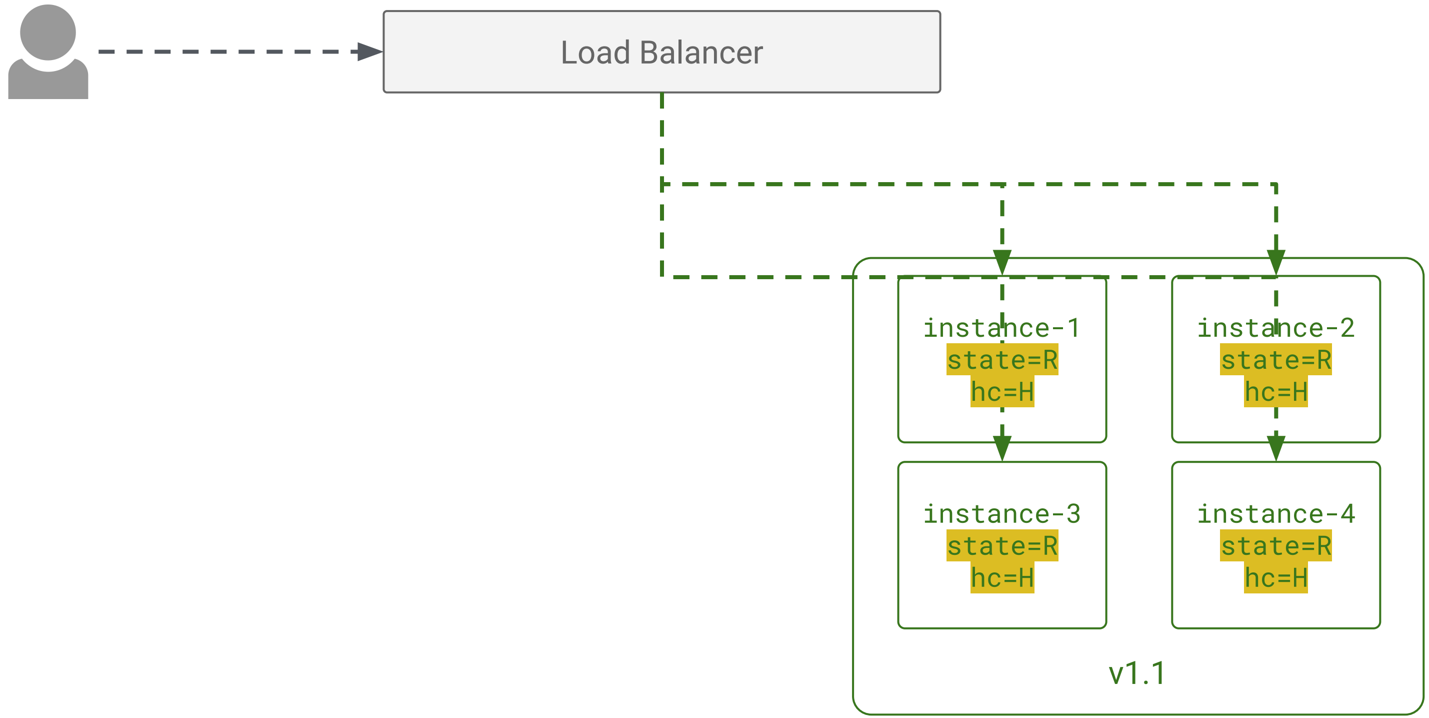
Blue/Green Upgrades



Blue/Green Upgrades

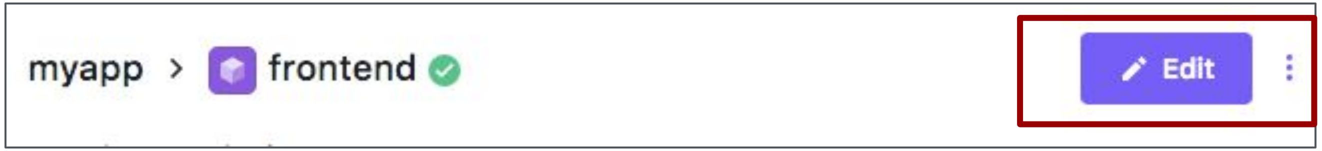


Blue/Green Upgrades



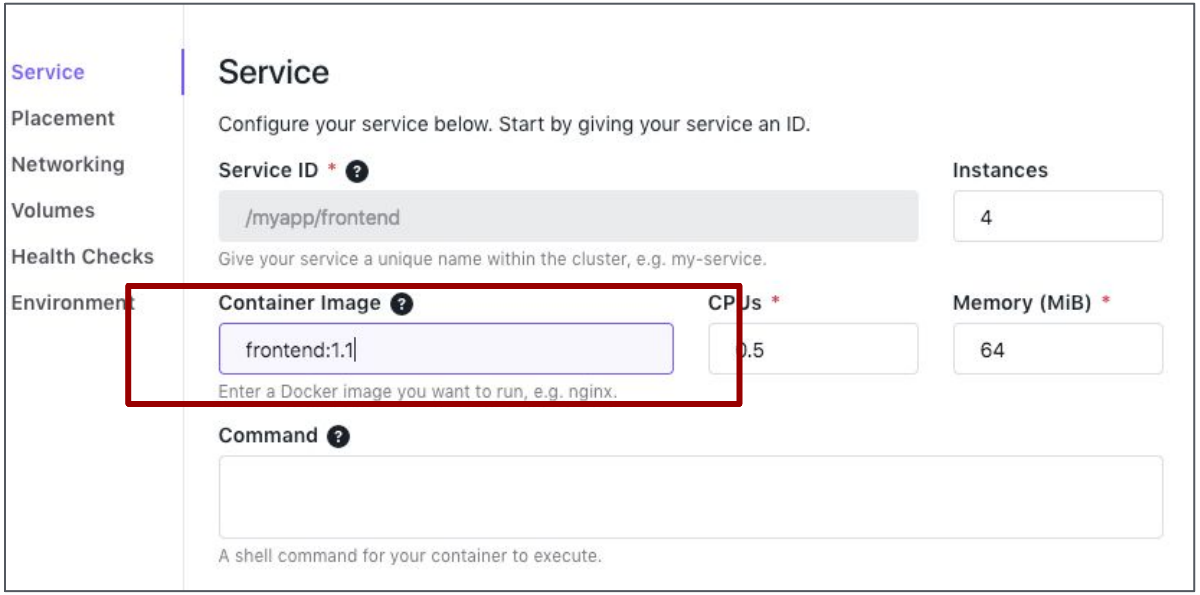
Performing a Rolling Upgrade Through DC/OS GUI

- Navigate to **Services** and select the service you are upgrading
- Click on the **Edit** button



Performing a Rolling Upgrade Through DC/OS GUI

- Change container image to new version and deploy your changes



Service

Configure your service below. Start by giving your service an ID.

Service ID * : /myapp/frontend
Give your service a unique name within the cluster, e.g. my-service.

Instances: 4

Container Image: frontend:1.1
Enter a Docker image you want to run, e.g. nginx.

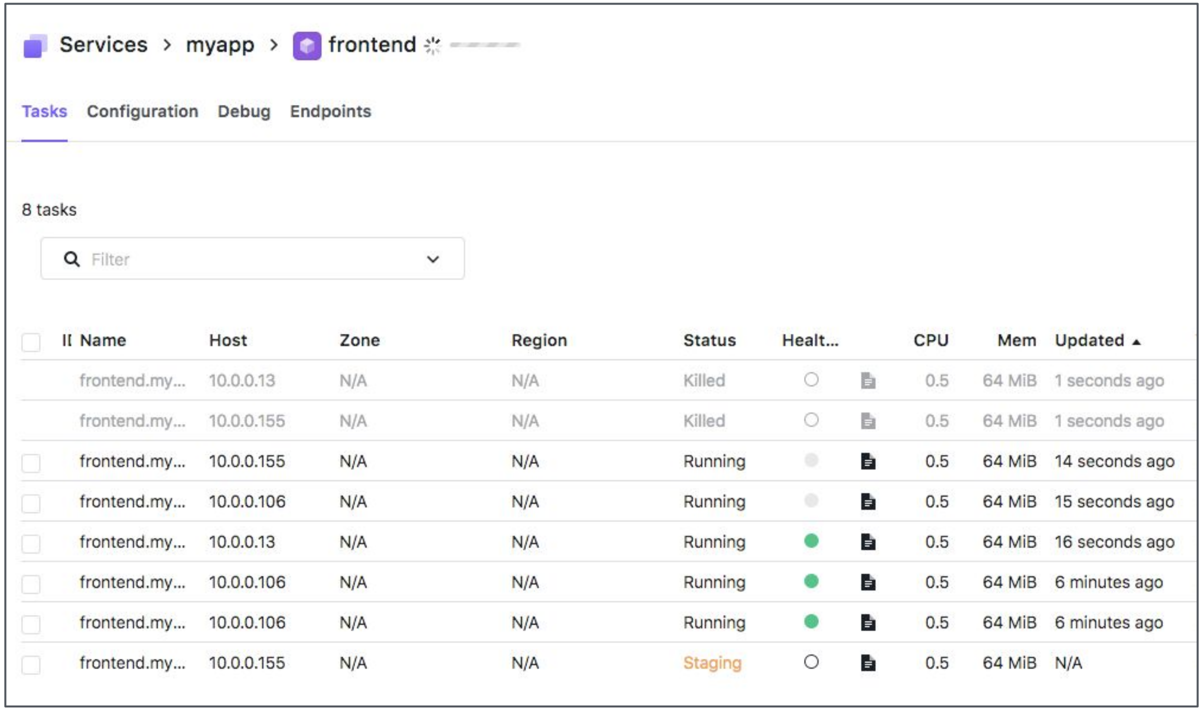
CPUs * : 0.5

Memory (MiB) * : 64

Command:
A shell command for your container to execute.

Performing a Rolling Upgrade Through DC/OS GUI

- Watch the magic happen!



8 tasks

Name	Host	Zone	Region	Status	Healt...		CPU	Mem	Updated ▲
frontend.my...	10.0.0.13	N/A	N/A	Killed	○		0.5	64 MiB	1 seconds ago
frontend.my...	10.0.0.155	N/A	N/A	Killed	○		0.5	64 MiB	1 seconds ago
frontend.my...	10.0.0.155	N/A	N/A	Running	●		0.5	64 MiB	14 seconds ago
frontend.my...	10.0.0.106	N/A	N/A	Running	●		0.5	64 MiB	15 seconds ago
frontend.my...	10.0.0.13	N/A	N/A	Running	●		0.5	64 MiB	16 seconds ago
frontend.my...	10.0.0.106	N/A	N/A	Running	●		0.5	64 MiB	6 minutes ago
frontend.my...	10.0.0.106	N/A	N/A	Running	●		0.5	64 MiB	6 minutes ago
frontend.my...	10.0.0.155	N/A	N/A	Staging	○		0.5	64 MiB	N/A

Lab 10

Rolling Upgrades

Persistence

Overview

Two options for stateful services:

1. Local volumes

   Documentation

2. External volumes

   Documentation

Local Volumes

Overview:

- Provisions a directory on the agent node and maps it to an instance of a Marathon service
- State is persisted even if containers are killed
- Configures dynamic reservations to ensure that a container gets relaunched on the same agent node

Limitations:

- Resource requirements (CPU, RAM, disk) cannot change post-launch
- Replication and backups are not the responsibility of DC/OS
- Destroying a persistent volume does not reclaim space from the filesystem - this must be done manually

Declaring a Local Volume

```
“container”: {
     “volumes”: [{
          “containerPath”: “data”,
          “mode”: “RW”,
          “persistent”: {
               “size”: 1024
          }
     }]
}
“unreachableStrategy”: “disabled”
“upgradeStrategy”: {
     “maximumOverCapacity”: 0,
     “minimumHealthCapacity”: 0
}
```

Master and Agent Disk and Filesystem Best Practices

When possible, it is a good practice to have separate partitions for the following mount points:

- `/var/lib/mesos`: This is the disk space that is advertised in the DC/OS UI
- `/var/lib/mesos/slave/volumes`: This is used by frameworks that consume `ROOT` persistent volumes
- `/var/lib/mesos/docker/store`: This is used to store the Docker image layers that are used to provision UCR containers
- `/var/lib/docker`: This is used to store the Docker image layers and by containers provisioned by the Docker containerizer
- `/var/lib/dcos`: On masters, this is where Exhibitor stores Zookeeper data
- `/var/lib/dcos`: On agents, this is where persistent configuration files are stored
- `/dcos/volume<n>`: This is used by frameworks that consume `MOUNT` persistent volumes

External Volumes

- Allows the associate of a device from an external storage system to an instance of a Marathon service
- Supported storage backends:
  - Amazon EBS
  - GCE disk
  - OpenStack Cinder
  - EMC ScaleIO
  - EMC XtremIO
  - EMC VMAX
  - EMC Isilon
- State is persisted even if application tasks are killed/fail
- Underlying implementation is provided through Dell/EMC Code's REXRAY project

Declaring an External Volume

```
“container”: {
     “volumes”: [{
          “containerPath”: “data”,
          “mode”: “RW”,
          “external”: {
               “size”: 1024,
               “name”: “my-persistent-volume”,
               “provider”: “dvdi”,
               “options”: { “dvdi/driver”: “rexray” }
          }
     }]
},
“upgradeStrategy”: {
     “minimumHealthCapacity”: 0,
     “maximumOverCapacity”: 0
}
```

Monitoring, Metrics, and Logs

DC/OS Metrics API

- The `dcos-metrics` component is responsible for providing metrics pertaining to various entities in a DC/OS cluster
- Runs on every node in the cluster
- Exposes a REST API that allows for metrics consumption
- Three layers of metrics available on DC/OS:
    - **Host**: metrics relating to a node (master/agent) in the cluster
    - **Container**: metrics about cgroup allocations from containers running in the cluster
    - **Application**: metrics about a specific application running inside a container in the cluster

DC/OS Metrics: Host

Metric	Description
`cpu.{cores,idle,total}`	Percentage of cores used/idle/total.
`load.{1,5,15}min`	Load average for the past 1, 5, and 15 minute(s).
`memory.{free,total}`	Amount of free/total memory in bytes.
`processes`	Number of processes that are running.
`swap.{free,used,total}`	Amount of swap free/used/total.
`uptime`	The system reliability and load average.
`filesystem.{name}.capacity.{free,total,used}`	Amount of free/total/used storage capacity for filesystem
`network.{name}.in.{bytes,dropped,errors,packets}`	Incoming network metrics for a network interface
`network.{name}.out.{bytes,dropped,errors,packets}`	Outgoing network metrics for a network interface

DC/OS Metrics: Container

Metric	Description
`cpus.limit`	Number of allocated CPUs
`cpus.system_time_secs`	Total CPU time spent in kernel mode in seconds
`cpus.throttled_time_secs`	Total time in seconds the CPU was throttled
`cpus.user_time_secs`	Total CPU time spent in user mode
`disk.limit_bytes`	Hard memory limit for disk in bytes
`disk.used_bytes`	Hard disk used in bytes
`mem.limit_bytes`	Hard memory limit for a container
`mem.total_bytes`	Total memory of a process in RAM
`net.rx.{bytes,dropped,errors,packets}`	Incoming network metrics
`net.tx.{bytes,dropped,errors,packets}`	Outgoing network metrics

DC/OS Metrics: Application

- Application metrics can be published to the dcos-metrics component through environment variables
  - `STATSD_UDP_HOST`: initialized to the IP address of the agent where the task is running
  - `STATSD_UDP_PORT`: initialized to the bind port of the metrics-api on the agent
- Metrics are appended with structured data such as:
  - `agent_id`
  - `container_id`
  - `framework_id`
  - `hostname`
  - `labels`
- Only works with UCR containers

Logs

- DC/OS provides various methods for extracting the logs which can then be sent off to various log repositories and analytics engines
  - `dcos node log`
  - `dcos service log`
  - `dcos task log`
- You can also check the Mesos GUI
  - `http(s)://<master_ip>/mesos`

DC/OS Logging

- DC/OS logs can be retrieved through a provided REST API
- Logs can be returned in plaintext, JSON, or as event-streams
- Requires superuser credentials
- Documentation

Lab 11

Monitoring and Metrics

Summary: Day Two

In this module we covered:

- Launching containers
- Specifying constraints
- Configuring health and readiness checks
- Pods
- Application upgrades
- Persistence options for stateful services
- Monitoring, metrics, and logs

MESOSPHERE