DevOps, Software Evolution and Software Maintenance

Scaling

Mircea Lungu, Associate Professor,
IT University of Copenhagen, Denmark
mlun@itu.dk

Before we Start: State of the Projects

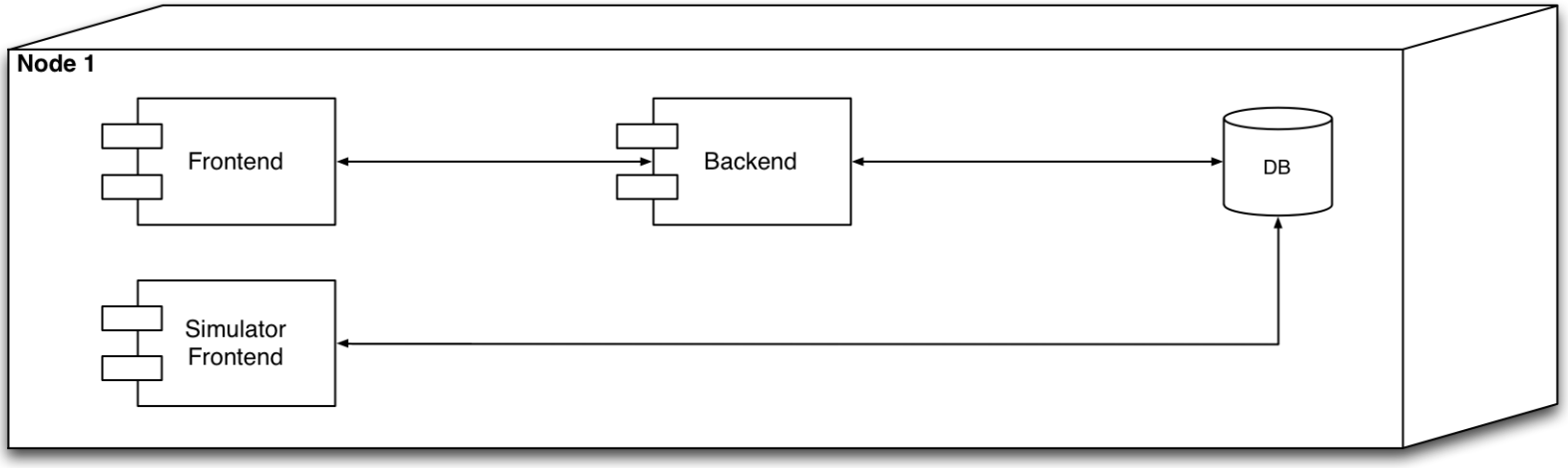
Logging: 6/16 … Keep up the good work!

Security … how are we doing?

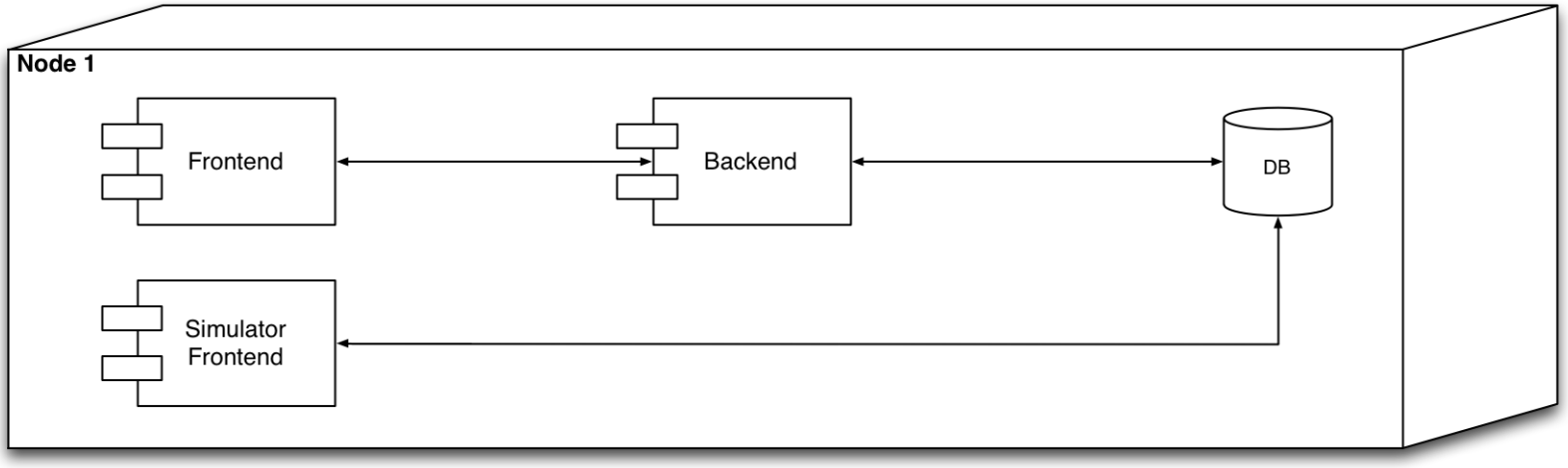
Latest processed events

Error plot

What happens if one of your components fails in this system?

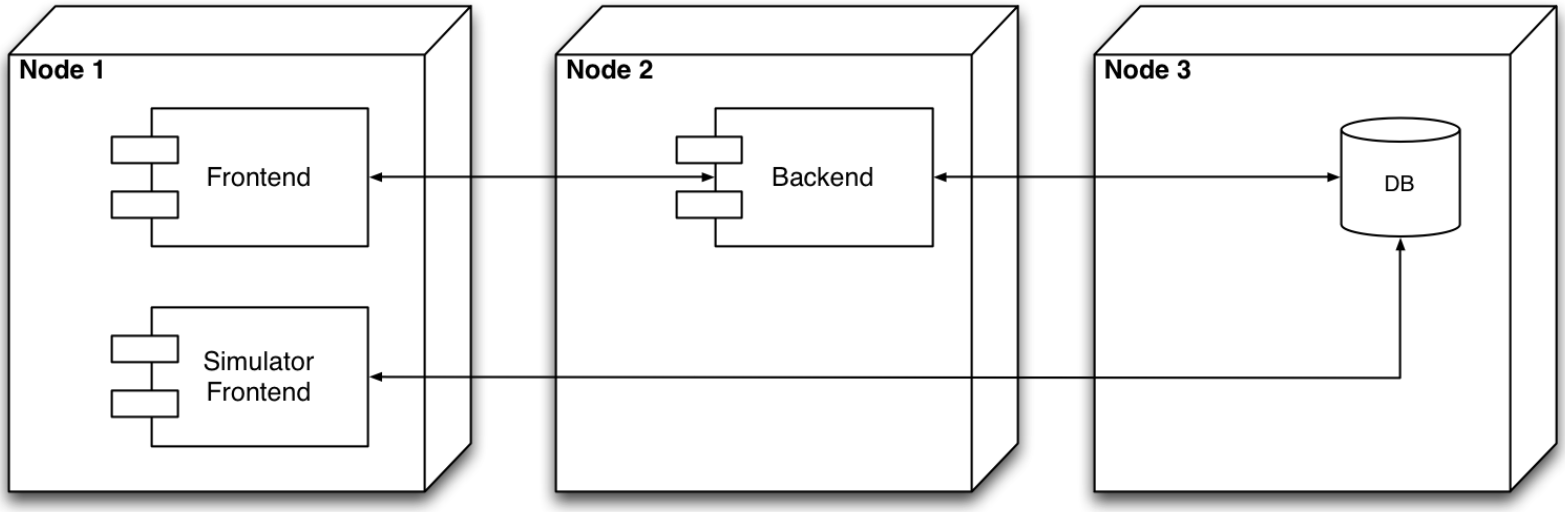


What happens if one of your components fails in this system?

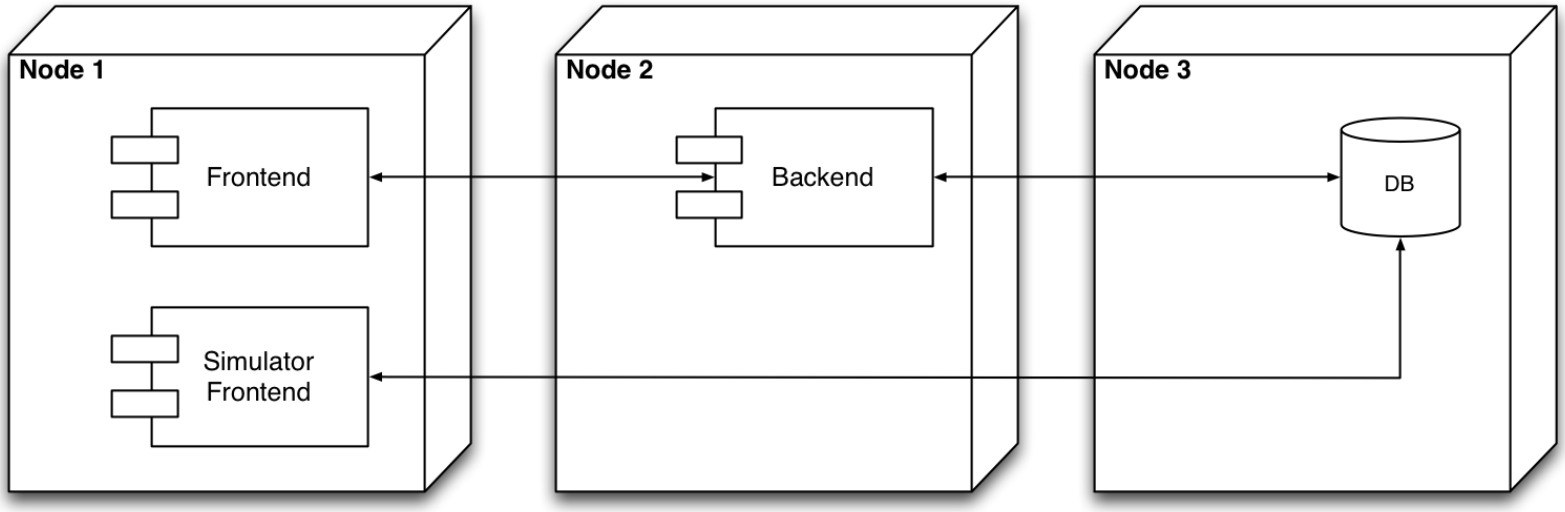


A system has a **single point of failure** if **a part of the system failing will stop the entire system from working**

Is this scenario better?



Is this scenario better?



Nope. Here we just have more **single points of failure** because each hardware node is one.

Solution?

Redundancy

= adding extra hardware and/or software components to the system and designing such that in case one fails, the other can take over

- solution to single-point of failure
- challenge = keeping redundant components in sync

What happens when there is a spike in number of users?

Possible Reasons:

- The Slashdot effect
- Seasonal spikes in demand
- Highly anticipated launch (e.g. the healthcare.gov story)

What happens when there is a spike in number of users?

Possible Reasons:

- The Slashdot effect
- Seasonal spikes in demand
- Highly anticipated launch (e.g. the healthcare.gov story)

Congestion

= **reduced quality of service that occurs when a network node or link is carrying more data than it can handle**

Solution to congestion?

Congestion

= **reduced quality of service that occurs when a network node or link is carrying more data than it can handle**

Solution to congestion?

Scaling

1. **Vertical**
2. **Horizontal**

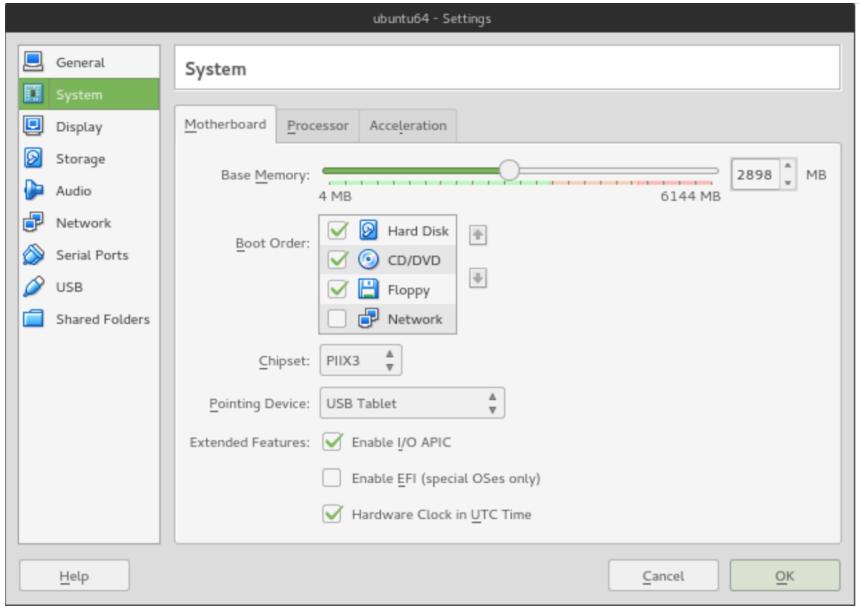
Vertical Scaling

Replacing resources with larger or more powerful ones

- In a physical server: open the hood, and add: more memory, harddisk, etc.
- In VMs: we can do that at runtime; still implies a temporary loss of service
- Usually a manual process (but can be automated)

Example 1. Vertical Scaling with the VirtualBox GUI

1. Power Off VM
2. Modify RAM and storage (either via GUI or CLI)
3. Power On VM
4. From within the VM update the OS wrt your new disk size (takes a while!)



Example 2: VirtuaBox and CLI

```
$ VBoxManage list vms
"coursevm" {e072b310-1922-4113-93f5-2ca865e01722}
"lsd2018vm" {67fda2ea-7c3e-42f6-9a13-d0908020322d}
```

```
$ VBoxManage modifyvm "coursevm" --cpus 8
$ VBoxManage modifyvm "coursevm" --memory 8192
```

```
$ VBoxManage list hdds

UUID:           9953ea1b-7295-4547-94fa-209f49c258f5
Parent UUID:    base
State:          created
Type:           normal (base)
Location:       /path/to/node1ubuntu-16.04-amd64-disk001.vmdk
Storage format: VMDK
Capacity:       40960 MBytes
Encryption:     disabled
```

```
$ VBoxManage clonehd "9953ea1b-7295-4547-94fa-..." "cloned.vdi" --format vdi
$ VBoxManage modifymedium disk "cloned.vdi" --resize 65536
```

Example 3: Vertical Scaling with DigitalOcean

Similar to VirtualBox, only that on the Web

You can only resize VMs that are not running.

Example 4: Vertical Scaling With the REST API of DigitalOcean

```
$ curl -X POST -H 'Content-Type: application/json' \
       -H "Authorization: Bearer $DIGITAL_OCEAN_TOKEN" \
       -d '{"type":"resize","size":"s-2vcpu-4gb"}' \
       "https://api.digitalocean.com/v2/droplets/$DROPLET_ID/actions" | jq
```

Notes:

- $DIGITAL_OCEAN_TOKEN environment variable is defined
- $DROPLET_ID is defined
- Image types and sizes at: https://slugs.do-api.dev/

Example 4: Vertical Scaling With the REST API of DigitalOcean

```
$ curl -X POST -H 'Content-Type: application/json' \
       -H "Authorization: Bearer $DIGITAL_OCEAN_TOKEN" \
       -d '{"type":"resize","size":"s-2vcpu-4gb"}' \
       "https://api.digitalocean.com/v2/droplets/$DROPLET_ID/actions" | jq
```

Notes:

- $DIGITAL_OCEAN_TOKEN environment variable is defined
- $DROPLET_ID is defined
- Image types and sizes at: https://slugs.do-api.dev/
- Resize CPU and RAM **automatically shuts down the droplet**

```
$ curl -X POST -H 'Content-Type: application/json' \
-H "Authorization: Bearer $DIGITAL_OCEAN_TOKEN" \
-d '{"type":"power_on"}' \
"https://api.digitalocean.com/v2/droplets/$DROPLET_ID/actions"
```

Contexts for Vertical Scaling

- Legacy systems (e.g. bank mainframes)
- Situations where **HW cost is lower than distributing the architecture** (developer cost & expertise might be higher than upgrading the HW)
- Some types of software that does not scale well horizontally (e.g. DBs that don't cluster well)

Drawbacks of Vertical Scaling

1. Can't adapt fast and easy to varying workload
    1. Complicated to scale down
    2. Slow -- it implies switching machines off and on
2. Some services are too big for it
3. Still has a single point of failure

Horizontal Scaling

= **Increasing the computing power by**

- **adding more machines* to a setup**
- **making all the machines share the responsibilities**

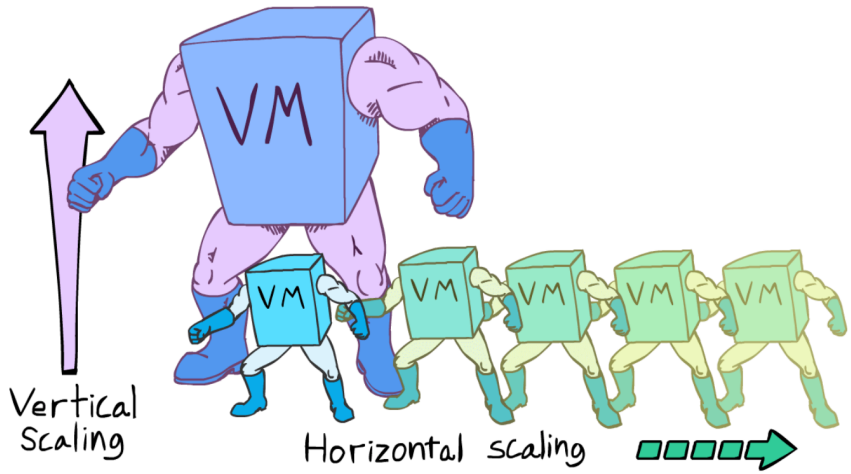


Image Source — turbonomic.com

*Initially was about physical machines. Nowadays VMs too.

Why?!

Why?!

Lack of Alternative

Some workloads exceed the capacity of the largest supercomputer and can only be handled by distributed systems, e.g.

- As of 2000 Google can not host all their DB on a single machine. In 2004 they introduce the MapReduce paper and the whole world gets excited.
- Brief History of Scaling at LinkedIn: *"An easy fix we did was classic vertical scaling ... While that bought some time, we needed to scale further"*

Why?!

Lack of Alternative

Some workloads exceed the capacity of the largest supercomputer and can only be handled by distributed systems, e.g.

- As of 2000 Google can not host all their DB on a single machine. In 2004 they introduce the MapReduce paper and the whole world gets excited.
- Brief History of Scaling at LinkedIn: *"An easy fix we did was classic vertical scaling ... While that bought some time, we needed to scale further"*

Cost

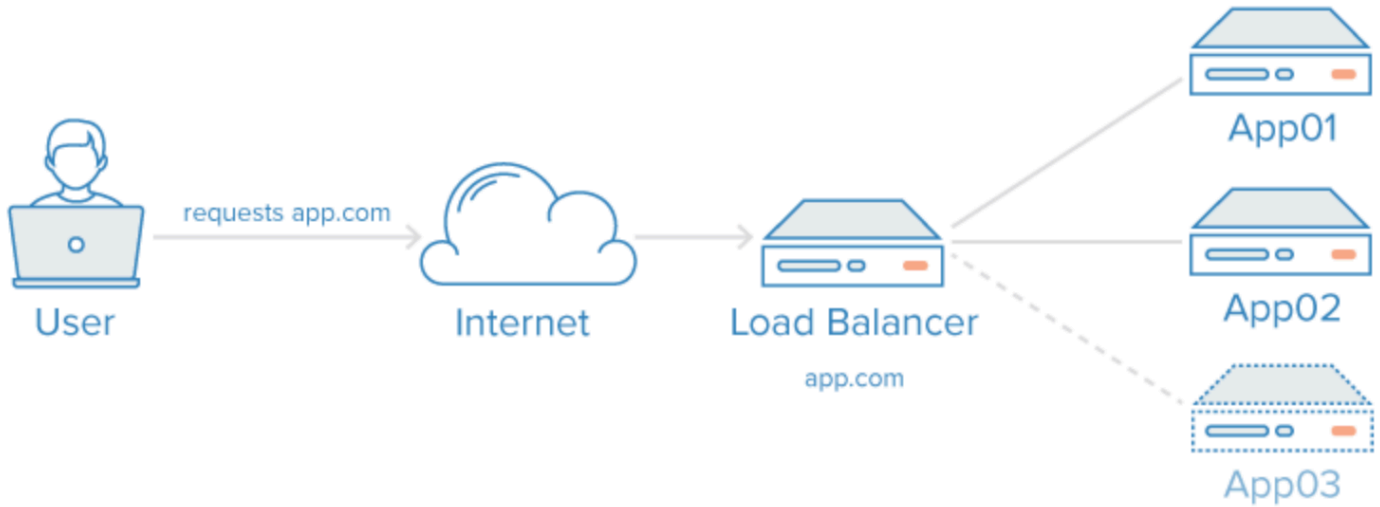
Tasks that once would have required expensive supercomputers can be done for less:

- seismic analysis
- biotechnology
- SETI@Home

Automated Load-Balancing

= **Distributing traffic to - and computation across multiple servers**

- Ensures no single server bears too much demand
- Improves responsiveness

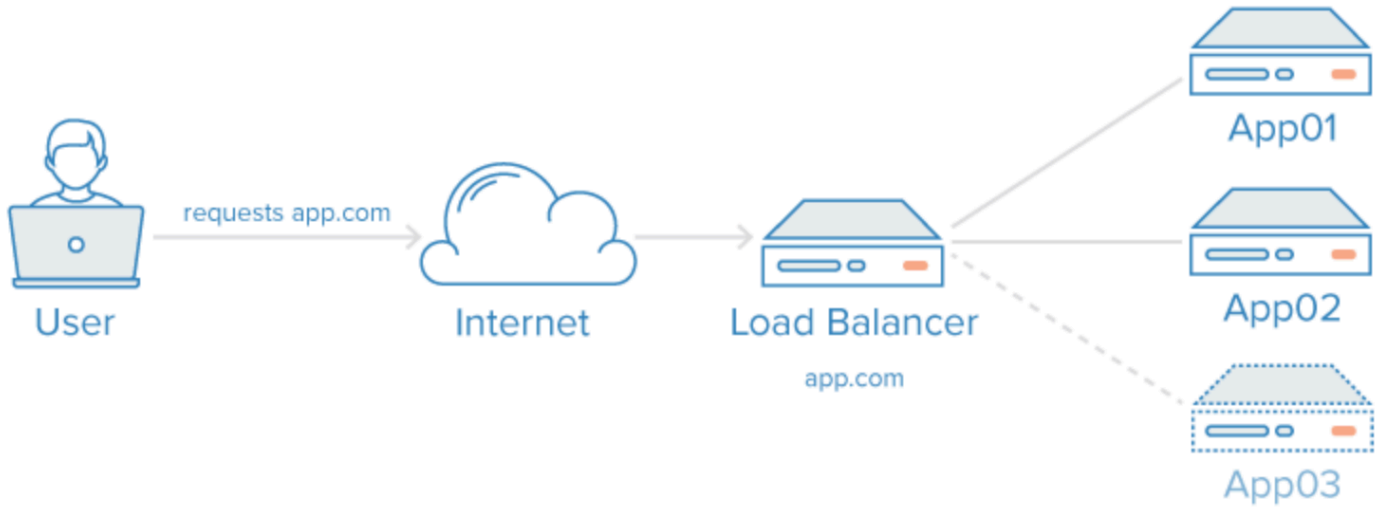


Solves **scaling** and SPF at the application server level but...

Automated Load-Balancing

= **Distributing traffic to - and computation across multiple servers**

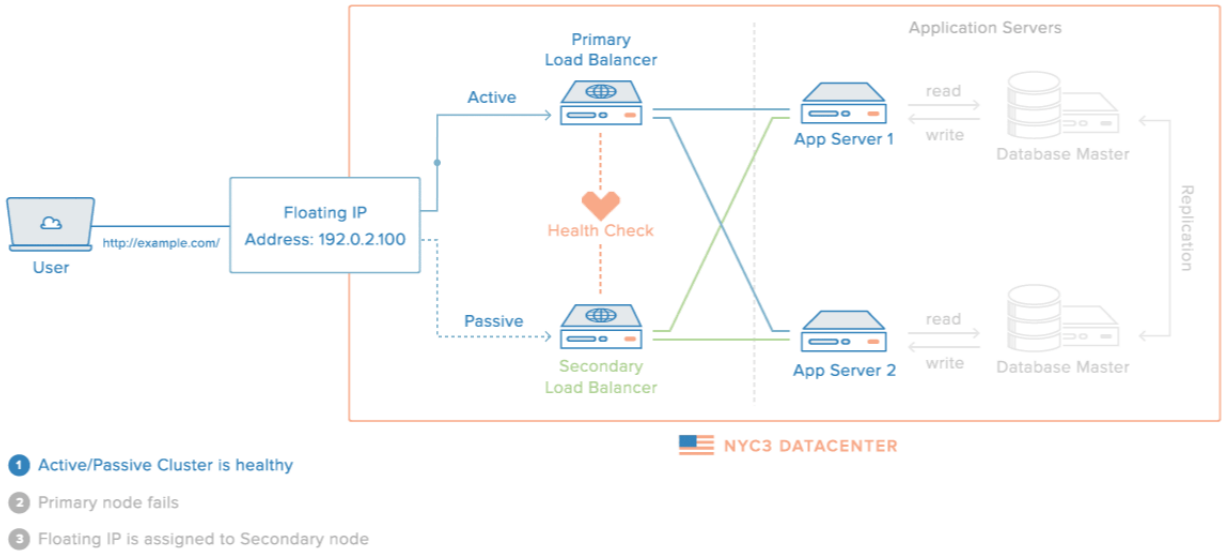
- Ensures no single server bears too much demand
- Improves responsiveness



Solves **scaling** and SPF at the application server level but...

Redundant Load Balancer Setup

Load balancer is not anymore a single point of failure (setup description)



- Reserved IP (used to be Floating IP)
  - DigialOcean name for static IPs
  - Equivalents on other platforms, e.g. Elastic IPs @ Amazon
- Keepalived - daemon used for health check

Tools that Help With Horizontal Scaling

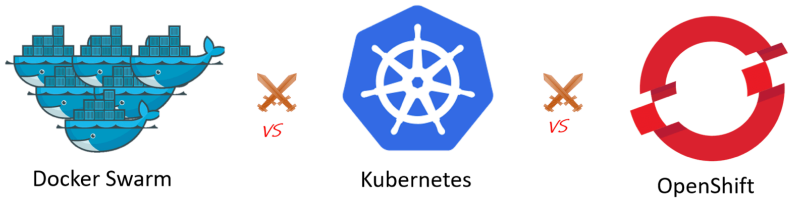
Specialized Tools for Distributing Computations

- Hadoop - Implementation of MapReduce
- Spark - Faster, more generic implementation

Container Orchestration Tools

- Managing **computing nodes** and services
- Resource aware task scheduling

Many Container Orchestration Alternatives



Docker Swarm Mode

- Comes with Docker by default
- The easiest to use from all the alternatives

Kubernetes

- Originally developed at Google
- We don't ask you to use it because we're nice :) (see hacker news discussion)

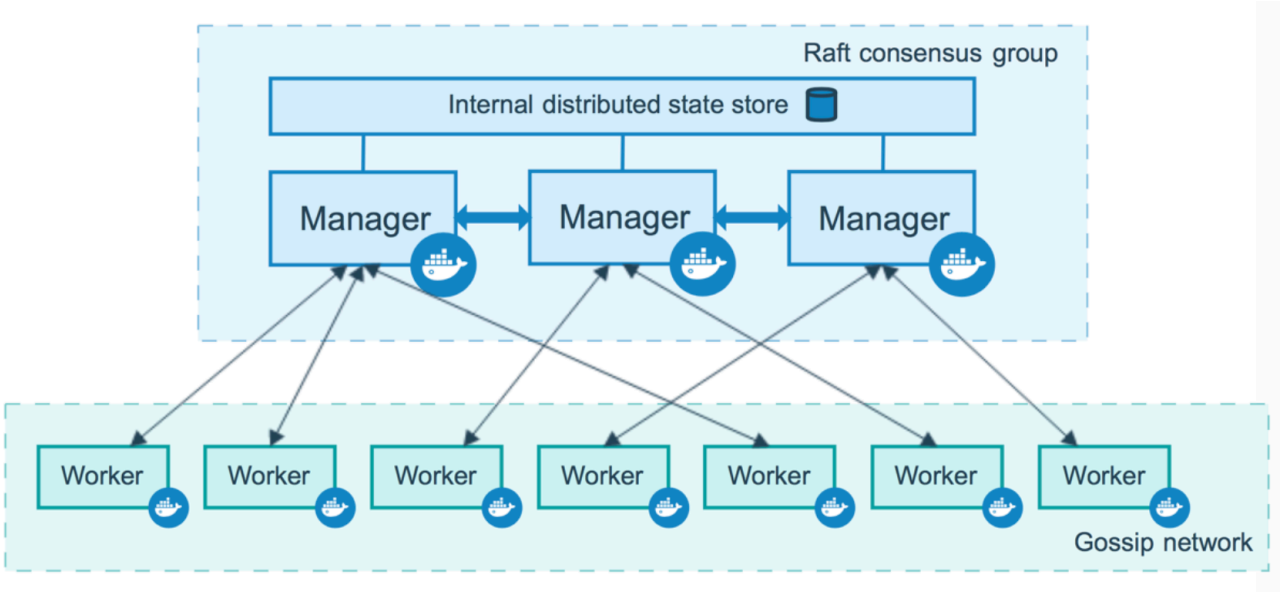
... and many more

Docker Swarm Concepts

- Node = A VM instance participating in a swarm
  1. Managers
  2. Workers
- Services
- Tasks
- The Routing Mesh

1. Manager Node

- Maintain cluster state
- Schedule services
- An *n manager swarm* tolerates maximum loss of *(n-1)/2* managers.



Notes:

- Docker recommends a *maximum of seven manager nodes for a swarm* (!?!)

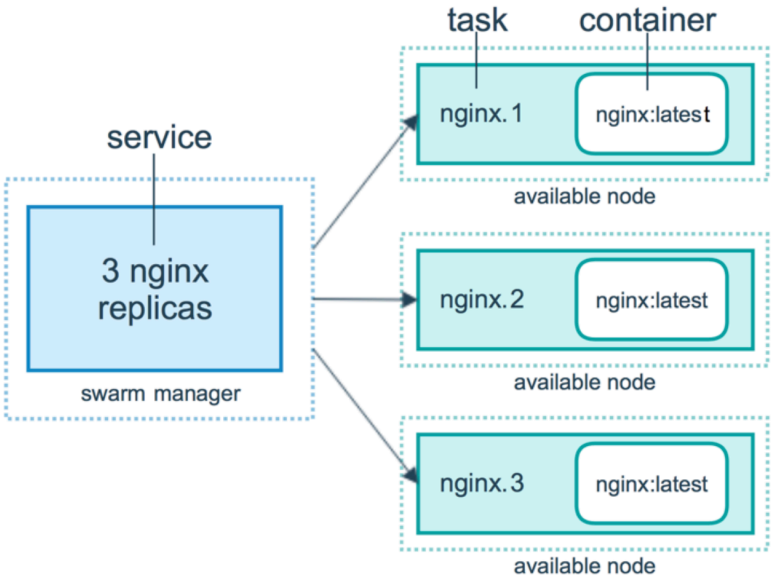
2. Worker Node

- Sole purpose to execute Docker containers
- Each one runs an instances of Docker Engine
- Do not participate in scheduling decisions
- Have at least one manager node
- By default, all managers are also workers

3. Service

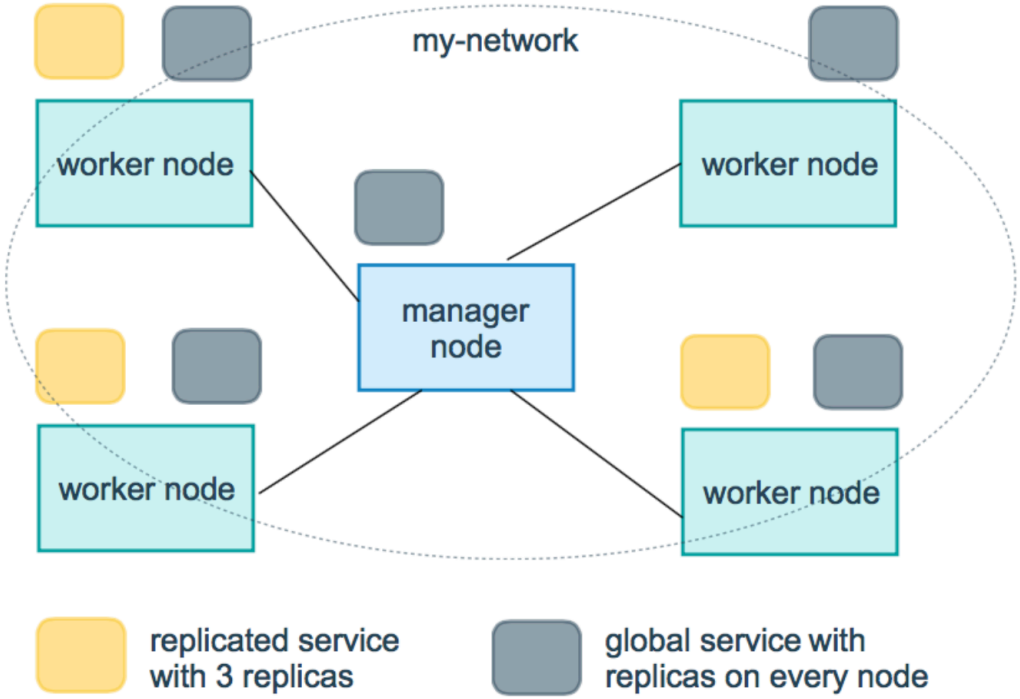
= the definition of the tasks to execute on the nodes

- The primary abstraction of user interaction with the swarm
- Is bound to a port



Types of Services

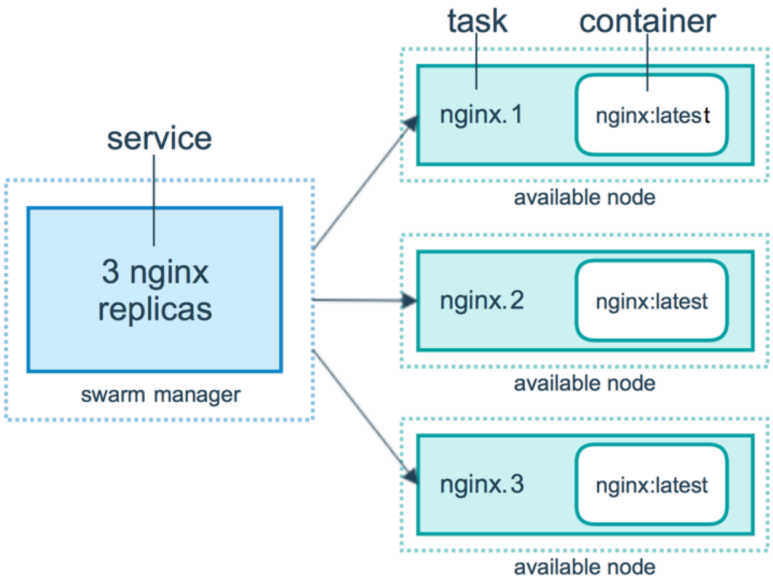
Can be **replicated** or **global** (exactly one replica running on each node)



What service does it make sense to have "global" replication for?

4. Task

- The atomic scheduling unit of swarm
- Carries **a container and the commands to run inside it**
- Manager nodes assign tasks to worker nodes according to the number of replicas set in the service scale



"A service is a description of a desired state, and a task does the work"

5. The Routing Mesh

- Routes all incoming requests to published ports on available nodes to an active container
- Enables each node in the swarm to accept connections
  - on published ports
  - for any service running in the swarm
  - even if there's no task running on the node

- Can support load balancing in Docker Swarm

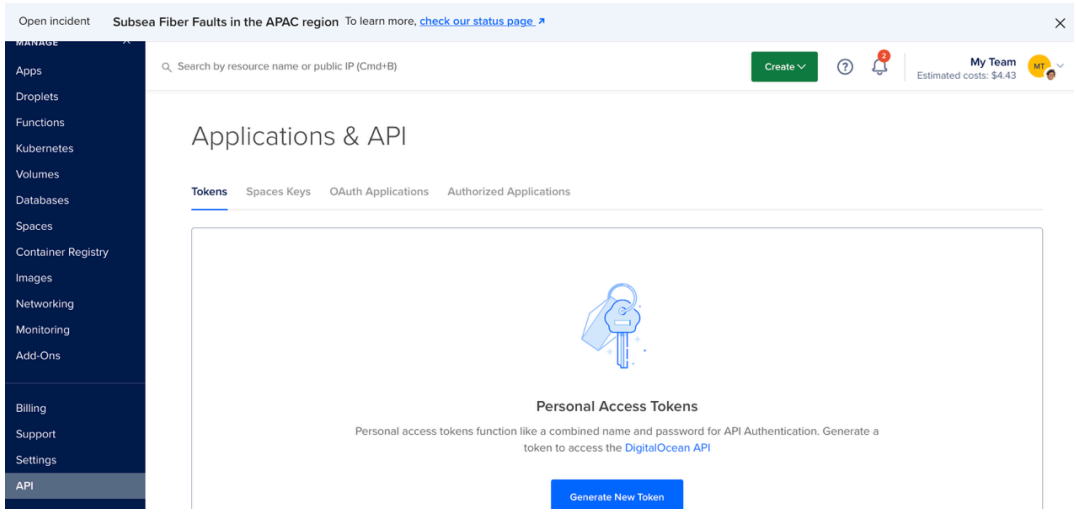
Read more: https://docs.docker.com/engine/swarm/ingress

New Docker Commands

- `docker swarm` ... to manage a cluster (swarm)
- `docker service` ... to manage replicated containers (services) in the swarm

Interactive

CLI Deployment of A Docker Swarm cluster on DigitalOcean



Assumes:

- Definition envvar: $DIGITAL_OCEAN_TOKEN
- Availability the `jq` command line tool (e.g., `brew install jq` on a mac)
- SSH public key uploaded in DigitalOcean>Settings>Security

Creating a Docker Swarm Cluster Node

```
export DIGITALOCEAN_PRIVATE_NETWORKING=true
export DROPLETS_API="https://api.digitalocean.com/v2/droplets"
export BEARER_AUTH_TOKEN="Authorization: Bearer $DIGITAL_OCEAN_TOKEN"
export JSON_CONTENT="Content-Type: application/json"
```

```
CONFIG='{"name":"swarm-manager","tags":["demo"],
    "size":"s-1vcpu-1gb", "image":"docker-20-04",
    "ssh_keys":["01:97:fe:0a:01:e3:a9:68:99:60:b5:e9:74:30:8f:71"]}'

SWARM_MANAGER_ID=$(curl -X POST $DROPLETS_API -d$CONFIG\
    -H $BEARER_AUTH_TOKEN -H $JSON_CONTENT\
    | jq -r .droplet.id ) && sleep 5 && echo $SWARM_MANAGER_ID
```

```
export JQFILTER='.droplets | .[] | select (.name == "swarm-manager")
    | .networks.v4 | .[]| select (.type == "public") | .ip_address'


SWARM_MANAGER_IP=$(curl -s GET $DROPLETS_API\
    -H $BEARER_AUTH_TOKEN -H $JSON_CONTENT\
    | jq -r $JQFILTER) && echo "SWARM_MANAGER_IP=$SWARM_MANAGER_IP"
```

Creating Worker Nodes

Worker1

```
WORKER1_ID=$(curl -X POST $DROPLETS_API\
       -d'{"name":"worker1","tags":["demo"],"region":"fra1",
       "size":"s-1vcpu-1gb","image":"docker-20-04",
       "ssh_keys":["01:97:fe:0a:01:e3:a9:68:99:60:b5:e9:74:30:8f:71"]}'\
       -H $BEARER_AUTH_TOKEN -H $JSON_CONTENT\
       | jq -r .droplet.id )\
       && sleep 3 && echo $WORKER1_ID
```

```
export JQFILTER='.droplets | .[] | select (.name == "worker1") | .networks.v4 | .[


WORKER1_IP=$(curl -s GET $DROPLETS_API\
    -H $BEARER_AUTH_TOKEN -H $JSON_CONTENT\
    | jq -r $JQFILTER)\
    && echo "WORKER1_IP=$WORKER1_IP"
```

Worker2

```
WORKER2_ID=$(curl -X POST $DROPLETS_API\
       -d'{"name":"worker2","tags":["demo"],"region":"fra1",
       "size":"s-1vcpu-1gb","image":"docker-20-04",
       "ssh_keys":["01:97:fe:0a:01:e3:a9:68:99:60:b5:e9:74:30:8f:71"]}'\
       -H $BEARER_AUTH_TOKEN -H $JSON_CONTENT\
       | jq -r .droplet.id )\
       && sleep 3 && echo $WORKER2_ID
```

```
export JQFILTER='.droplets | .[] | select (.name == "worker2") | .networks.v4 | .[


WORKER2_IP=$(curl -s GET $DROPLETS_API\
    -H $BEARER_AUTH_TOKEN -H $JSON_CONTENT\
    | jq -r $JQFILTER)\
    && echo "WORKER2_IP=$WORKER2_IP"
```

Making swarm-manager a Cluster Manager

Open the ports that Docker needs

```
ssh root@$SWARM_MANAGER_IP "ufw allow 22/tcp && ufw allow 2376/tcp &&\
ufw allow 2377/tcp && ufw allow 7946/tcp && ufw allow 7946/udp &&\
ufw allow 4789/udp && ufw reload && ufw --force  enable &&\
systemctl restart docker"
```

Initialize the swarm

```
ssh root@$SWARM_MANAGER_IP "docker swarm init --advertise-addr $SWARM_MANAGER_IP"
```

```
Swarm initialized: current node (sozjy3nmfrieacm2pbgj41ek3) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-4rndqz4hwe38wtbl9fwgj33rk48ok3hri7a0xy42o7s

To add a manager to this swarm, run 'docker swarm join-token manager' and follow t
```

Converting node-1 and node-2 to Workers

Let's get that token from the swarm-manager

```
$ ssh root@$SWARM_MANAGER_IP "docker swarm join-token worker -q"
SWMTKN-1-4rndqz4hwe38wtbl9fwgj33rk48ok3hri7a0xy42o7sf5ll38z-afkri2vu57m5z31v34bny1

$ WORKER_TOKEN=`ssh root@$SWARM_MANAGER_IP "docker swarm join-token worker -q"`
```

and build a command that we can run on node-1 and node-2 to join the swarm.

```
REMOTE_JOIN_CMD="docker swarm join --token $WORKER_TOKEN $SWARM_MANAGER_IP:2377"

ssh root@$WORKER1_IP "$REMOTE_JOIN_CMD"
```

```
This node joined a swarm as a worker.
```

```
ssh root@$WORKER2_IP "$REMOTE_JOIN_CMD"
```

```
This node joined a swarm as a worker.
```

Seeing the state of the cluster on the manager

```
$ ssh root@$SWARM_MANAGER_IP "docker node ls"

ID                            HOSTNAME            STATUS              AVAILABILITY
sozjy3nmfrieacm2pbgj41ek3 *   node-0              Ready               Active
hy6ie5xq561f9w1zpiyaqkrk5     node-1              Ready               Active
```

Starting a Service

Now that everything is setup, let's run a service on our cluster:

```
$ ssh root@$SWARM_MANAGER_IP "docker service create -p 8080:8080 --name appserver
overall progress: 0 out of 1 tasks
...
overall progress: 1 out of 1 tasks
verify: Waiting 5 seconds to verify that tasks are stable...
...
verify: Waiting 1 seconds to verify that tasks are stable...
verify: Service converged
```

... about 1-2 min ..

Checking the state of the service

```
$ ssh root@$SWARM_MANAGER_IP "docker service ls"
ID                  NAME                MODE                REPLICAS            IM
ttkqm9wzthgu        appserver           replicated          1/1                 st
```

You may directly ask for the state of a service with

```
$ ssh root@$SWARM_MANAGER_IP "docker service ps appserver"
```

Now, on a Mac you can:

```
$ open http://$SWARM_MANAGER_IP:8080
```

Alternatively, navigate manually to the swarm manager's IP port 8080 and see the webpage served.

Checking that the Swarm also restarts services

To demonstrate this we used the ***crashserver service***: a webserver which kills itself three seconds after serving an http request

Take some time and observe the behavior of the container before continuing with the guide.

- note how the infrastructure is self-healing, by checking the state of the service multiple times after an invocation as shown above.
- the service becomes unavailable while Swarm is recreating the container after it has been killed

How to increase availability?

Scaling the service to increase availability?

```
$ ssh root@$SWARM_MANAGER_IP "docker service scale appserver=5"
$ ssh root@$SWARM_MANAGER_IP "docker service ls"

ID                  NAME                MODE                REPLICAS            IM
ttkqm9wzthgu        appserver           replicated          5/5                 st



$ ssh root@$SWARM_MANAGER_IP "docker service ps appserver"


ID                  NAME                IMAGE                           NODE
vbg02o9bsaog        appserver.1         stifstof/crashserver:latest   node-1
mudpe1lokpj7        appserver.2         stifstof/crashserver:latest   node-0
t7enei6pz4jw        appserver.3         stifstof/crashserver:latest   node-0
sfpn4f2kg5nq        appserver.4         stifstof/crashserver:latest   node-1
wa8f99b6t199        appserver.5         stifstof/crashserver:latest   node-0
```

Does the replication work?

You should now be able to invoke the webpage without seeing the error-page each time the container is killed, but instead see the request being served by another container. Nice!

Althrough it is possible to kill all container by manically invoking the /status endpoint, if you want to test the self-healing feature of swarm, you can invoke the /kill endpoint, which will kill the container immediately, so you don't have to wait.

Checking that the routing mesh works as advertised

```
open http://$WORKER1_IP:8080
```

```
open http://$WORKER2_IP:8080
```

Note: Remmeber to open the 8080 port from the firewall.

Cleaning up to not pay anymore...

```
curl -X DELETE\
  -H $BEARER_AUTH_TOKEN -H $JSON_CONTENT\
  "https://api.digitalocean.com/v2/droplets?tag_name=demo"
```

See: documentation for the delete API endpoint

How do you **deploy a new version of the service when it's replicated?**

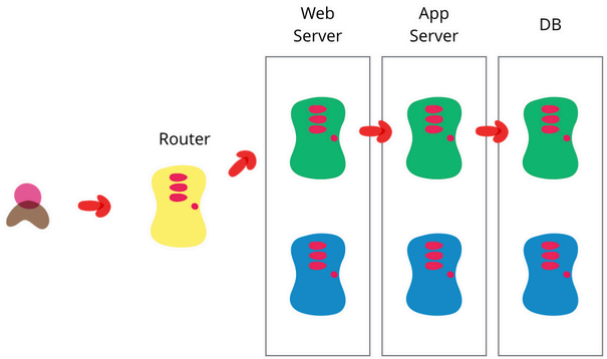
How do you **deploy a new version of the service when it's replicated?**

Upgrade Strategies

1. Blue-Green
2. Rolling Updates
3. Canary

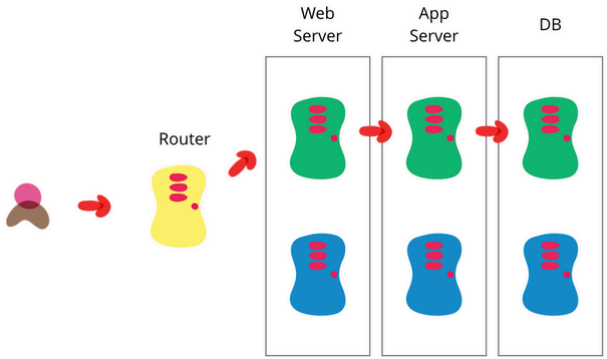
Blue-green

Two identical environments, where only one is hot at any time



Blue-green

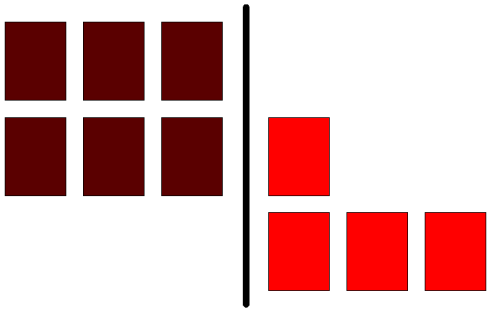
Two identical environments, where only one is hot at any time



1. Currently deployed application (Green) is serving incoming traffic
2. New version (Blue) is deployed and tested, but not yet receiving traffic
3. When Blue is ready, LB starts sending incoming traffic to it too
4. For a while: two versions of the application run in parallel
5. LB stops sending incoming traffic to the "Green"; "Blue" is handling all the incoming traffic
6. Green can now be safely removed
7. Blue is marked as Green...

Rolling Updates

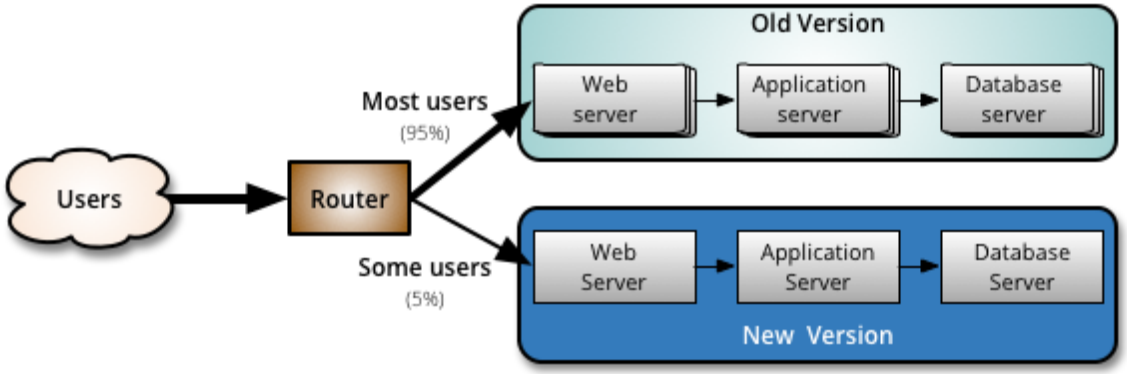
Deploy in rolling iterations



See https://opensource.com/article/17/5/colorful-deployme

Canary

Deploy to a small group first, then deploy to the rest



See: Article on martinfowler.com

Docker Swarm

Two update-order options: (stop-first | start-first)

- stop-first (default) -- corresponds to rolling updates
- start-first -- corresponds to blue-green service deployment

Rolling Updates (stop-first):

1. Stop the first *task*
2. Schedule update for the stopped task
3. Start the container for the updated task
4. If the update to a task returns RUNNING, wait for the specified delay period (`--update-delay` flag) then start the next task
5. If, at any me during the update, a task returns FAILED, pause the update

DB Migrations

With Blue Green Deployment

From article on martinfowler.com:

> *"The trick is to separate the deployment of schema changes from application upgrades"*

Same DB

> *"(One) variation would be to use the same database, making the blue-green switches for web and domain layers."*

1. deploy a database refactoring to change the schema to support both the new and old version of the application,
2. check everything is working fine so you have a rollback point,
3. then deploy the new version of the application.
4. And when the upgrade has bedded down remove the database support for

Why not Horizontal Scaling?

It can be more complicated that vertical (see hacker news thread on k8s)

Why not Horizontal Scaling?

It can be more complicated that vertical (see hacker news thread on k8s)

To *cargocult* what others (e.g. Google, Facebook) do

- There are many large scale infras that use veritcal scaling
  - Thibault Duplessis on the architecture of Lichess
  - StackOverflow does not use horizontal scaling (podcast, tweet)
- If Google needs it, than probably you don't

Why not Horizontal Scaling?

It can be more complicated that vertical (see hacker news thread on k8s)

To *cargocult* what others (e.g. Google, Facebook) do

- There are many large scale infras that use veritcal scaling
  - Thibault Duplessis on the architecture of Lichess
  - StackOverflow does not use horizontal scaling (podcast, tweet)
- If Google needs it, than probably you don't

To make up for bad architecture: algo, comm patterns, language, etc.

What Next?

Exercise: *Try out the swarm creation example from this lecture*

Practical: Scale your API

Workshop: propose topics on the Teams channel