Microservices Deployment and Scaling Tutorial

Table of Contents

[Objective 3](#_Toc450742571)

[Conventions Used in This document 3](#_Toc450742572)

[Microservices Deployment 4](#_Toc450742573)

[Rules of microservices deployments 5](#_Toc450742574)

[Docker 5](#_Toc450742575)

[Microservices Deployment Pattern 6](#_Toc450742576)

[Multiple Service Instances per Host 6](#_Toc450742577)

[Service Instance per Host 7](#_Toc450742578)

[Service Instance per Virtual Machine 7](#_Toc450742579)

[Service Instance per Container 8](#_Toc450742580)

[Serverless Deployment 9](#_Toc450742581)

[Zero Downtime Deployment 11](#_Toc450742582)

[Blue Green Deployment 11](#_Toc450742583)

[Canary Release 13](#_Toc450742584)

[Scaling Microservices 15](#_Toc450742585)

[The Art of Scalability 15](#_Toc450742586)

[X-axis scaling 15](#_Toc450742587)

[Y-axis scaling 16](#_Toc450742588)

[Z-axis scaling 17](#_Toc450742589)

[Deployment Guideline for Microservices Cluster 18](#_Toc450742590)

[The aim is to associate the load distribution mechanism (Load Balancer) for the deployment: 18](#_Toc450742591)

[Scaling & Deployment approach in Lift MicroService Template 19](#_Toc450742592)

[Prepare Cluster Environment (one time setup) 19](#_Toc450742593)

[Init Swarm 19](#_Toc450742594)

[First Deployment 19](#_Toc450742595)

[Build & Push Image 19](#_Toc450742596)

[Deploy 20](#_Toc450742597)

[Upgrade Application to New Version 21](#_Toc450742598)

[Check the group currently running 21](#_Toc450742599)

[Build & Push Image 21](#_Toc450742600)

[Deploy 22](#_Toc450742601)

[Publish the New Version 23](#_Toc450742602)

[Remove Instance(s) 23](#_Toc450742603)

[References 24](#_Toc450742604)

# Objective

This document is the guideline of microservices deployment, focus on Lift framework project. However, the concept can be applied to the project in any programming languages. The content will provide the basic instruction to deploy a project in various environments, as single node or cluster.

# Conventions Used in This document

The following typographical conventions are used in this tutorial:

Gray text

Indicate the key terms, page name, or specific topic in the screenshot.

Constant width

Indicate the programming code, script, or command line.

[Blue color with underline](http://localhost)

Indicate the hyperlink. The text is clickable to browse to the related content or website.

# Microservices Deployment

A microservices application may consists of tens or even hundreds of services. Services can be written in a variety of languages and frameworks. Each one is a mini-application with its own specific deployment, resource, scaling, and monitoring requirements. For example, you need to run a certain number of instances of each service based on the demand for that service. Also, each service instance must be provided with the appropriate CPU, memory, and I/O resources. What is even more challenging is that despite this complexity, deploying services must be fast, reliable and cost-effective.

The self-contained, independent, and reusable principles of microservice architecture help solve the problem of scaling and maintaining application services.

* **Self-contained**  
  Microservices are standalone operations that run without requiring other services. That means each runs on its own. You can scale each up or down and replace a service individually instead of updating the entire monolith given the business need and load.
* **Reusable**  
  Microservices are self-contained so you can reuse them for other applications and functions. We’re talking of reusing already deployed microservices.
* **Independent**  
  Microservices are platform agnostic, which means you can design them independent of infrastructure needs to run anywhere, in any cloud.

Database

Service A

Database

Service B

API Gateway

Instance 1

Instance 2

Database

Load Balancer

**Service C**

## Rules of microservices deployments

1. Each microservices deploy/undeploy independently  
   Each microservice is running on each own environment. Put infrastructure metadata for CPU, memory, storage, additional disks, virtual networks, firewalls, load balancing, and autoscaling in deployment policies attached to a cloud service like AWS, Google cloud, and more. Policies help you deploy the microservices defined in the script boxes to any platform and cloud. The orchestration engine interacts with the cloud APIs to auto-provision virtual machines based on metadata from deployment policies. So it’s easy to spin up and destroy microservices on demand without disrupting service availability.
2. Connect microservices via bindings  
   Each microservice provide the endpoints which handle REST or other protocol communications over the network. With bindings, network references are as easy to manipulate as programming variables.
3. Ability to scale at each microservices level  
   A service may get more traffic than other services. It may deploy to a cluster environment while some other services run as single node.
4. Failure in one microservice must not affect any of the other services

## Docker

Our Lift Microservice Template introduce docker for the microservices deployment.

Docker is a platform built on top of lightweight containers. It handles much of the work around handling containers. In Docker, you create and deploy apps which are synonymous with images in the VM world, albeit for a container-based platform. Docker manages the container provisioning, handles some of the networking problems, and even provides its own registry concept that allows you to store and version Docker applications.

Docker engine lets developers and system administrators deploy self-sufficient application containers in Linux environments provides a great way to deploy. The key steps involved are as follows:

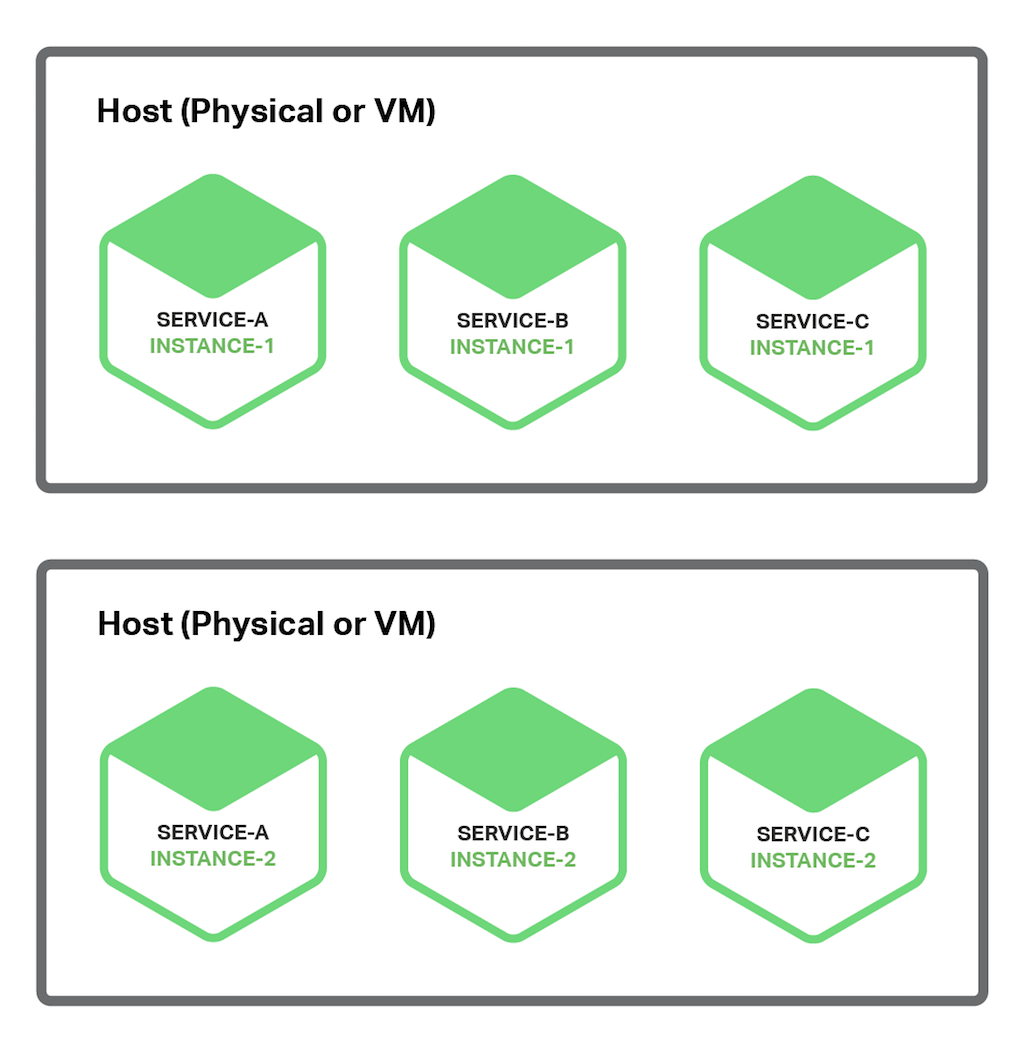
1. Package the microservice as a (Docker) container image.
2. Deploy each service instance as a container.
3. Scaling is done based on changing the number of container instances.
4. Building, deploying, and starting microservice will be much faster as we are using Docker containers (which is much faster than a regular VM)

|  |  |
| --- | --- |
| https://upload.wikimedia.org/wikipedia/commons/1/1e/Curation_bar_icon_info_35x35.png | More information about deployment by Docker, please read the following chapters in [Microservices Continuous Deployment Tutorial](Microservices%20Continuous%20Deployment%20Tutorial.docx) :   * Microservices Continuous Deployment * Application Deployment (Docker) |

# Microservices Deployment Pattern

## Multiple Service Instances per Host

This pattern is to use one or more physical or virtual hosts and run multiple service instances on each one. This is the traditional approach to application deployment. Each service instance runs at a well-known port on one or more hosts.



**Deployment Options:**

* Run each service instance as a process separately
* Run multiple service instances in the same process or process group e.g. deploy multiple java API applications on the same Apache Tomcat server

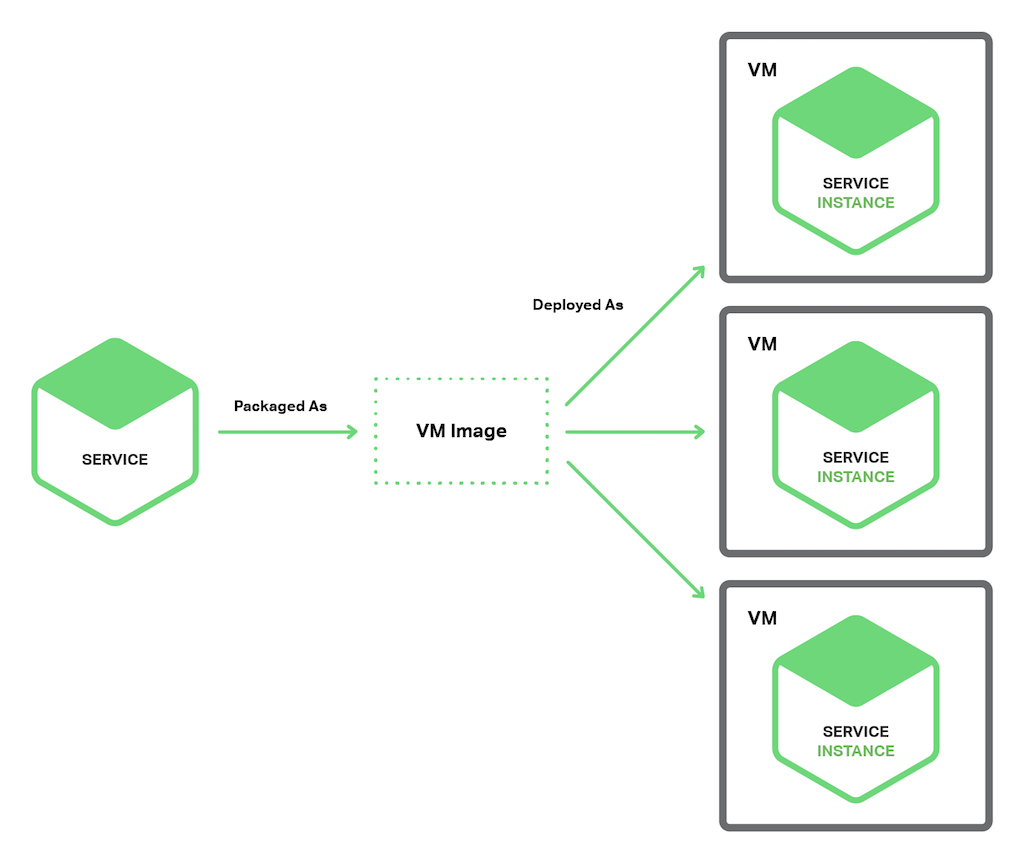
## Service Instance per Host

This pattern is to run each service instance in isolation on its own host. There are two different specializations of this pattern:

* Service Instance per Virtual Machine
* Service Instance per Container

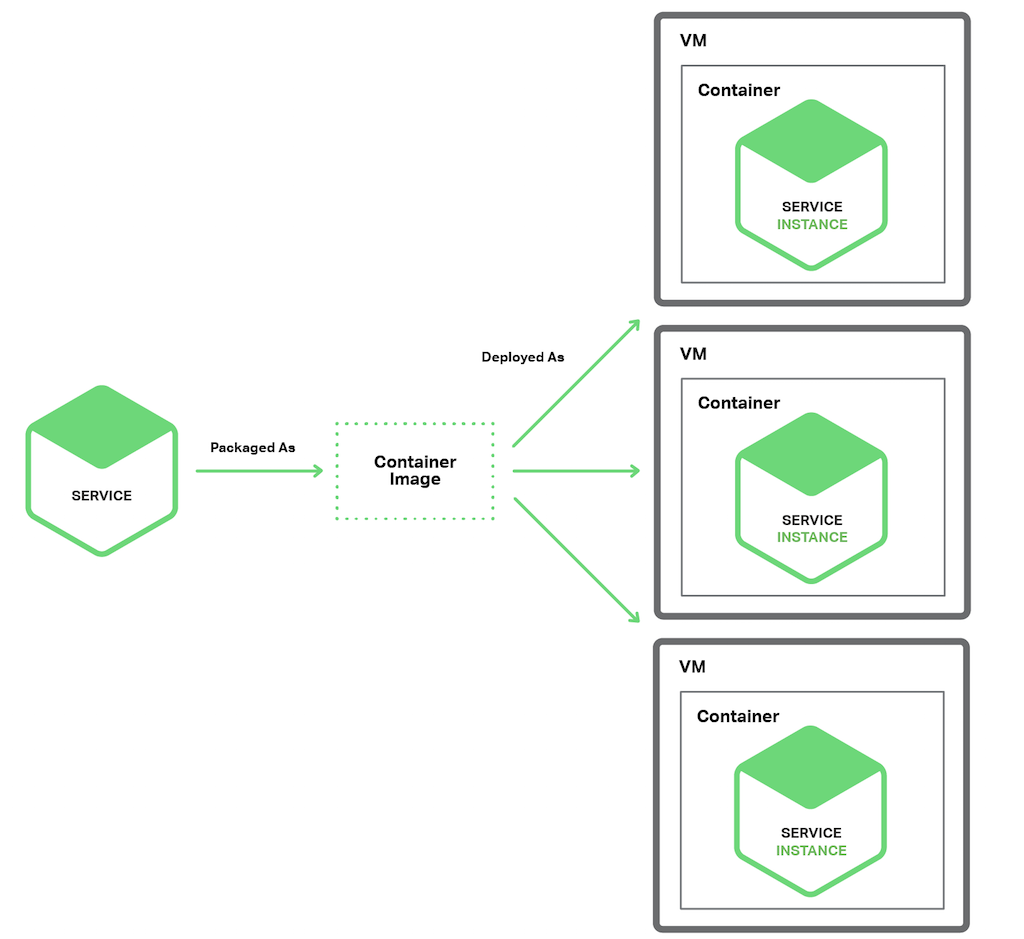
### Service Instance per Virtual Machine

This pattern is to package each service as a virtual machine (VM) image. Each service instance is a VM that is launched using that VM image.



### Service Instance per Container

Each service instance runs in its own container. Containers are a virtualization mechanism at the operating system level. A container consists of one or more processes running in a sandbox. From the perspective of the processes, they have their own port namespace and root filesystem. You can limit a container’s memory and CPU resources. Some container implementations also have I/O rate limiting. Examples of container technologies include Docker and Solaris Zones.



To use this pattern, you package your service as a container image. A container image is a filesystem image consisting of the applications and libraries required to run the service. Some container images consist of a complete Linux root filesystem. Others are more lightweight. To deploy a Java service, for example, you build a container image containing the Java runtime, perhaps an Apache Tomcat server, and your compiled Java application.

Once you have packaged your service as a container image, you then launch one or more containers. You usually run multiple containers on each physical or virtual host. You might use a cluster manager such as Kubernetes or Marathon to manage your containers. A cluster manager treats the hosts as a pool of resources. It decides where to place each container based on the resources required by the container and resources available on each host.

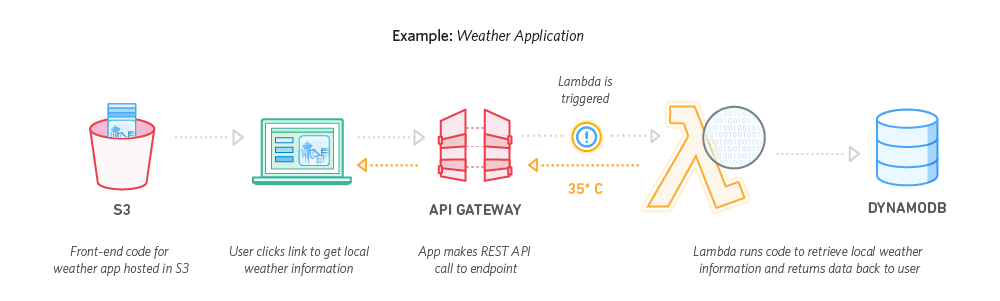
## Serverless Deployment

Example of serverless deployment technology is AWS Lambda. It supports Java, Node.js, and Python services.

AWS Lambda is a compute service where you can upload your code to AWS Lambda and the service can run the code on your behalf using AWS infrastructure. After you upload your code and create what we call a Lambda function, AWS Lambda takes care of provisioning and managing the servers that you use to run the code. You can use AWS Lambda as follows:

* As an event-driven compute service where AWS Lambda runs your code in response to events, such as changes to data in an Amazon S3 bucket or an Amazon DynamoDB table.
* As a compute service to run your code in response to HTTP requests using Amazon API Gateway or API calls made using AWS SDKs.

To deploy a microservice, you package it as a ZIP file and upload it to AWS Lambda. You also supply metadata, which among other things specifies the name of the function that is invoked to handle a request (a.k.a. an event). AWS Lambda automatically runs enough instances of your microservice to handle requests. You are simply billed for each request based on the time taken and the memory consumed. Of course, the devil is in the details and you will see shortly that AWS Lambda has limitations. But the notion that neither you as a developer nor anyone in your organization need worry about any aspect of servers, virtual machines, or containers is incredibly appealing.



A Lambda function is a stateless service. It typically handles requests by invoking AWS services. For example, a Lambda function that is invoked when an image is uploaded to an S3 bucket could insert an item into a DynamoDB images table and publish a message to a Kinesis stream to trigger image processing. A Lambda function can also invoke third-party web services.

There are four ways to invoke a Lambda function:

1. Directly, using a web service request
2. Automatically, in response to an event generated by an AWS service such as S3, DynamoDB, Kinesis, or Simple Email Service
3. Automatically, via an AWS API Gateway to handle HTTP requests from clients of the application
4. Periodically, according to a cron-like schedule

AWS Lambda is a convenient way to deploy microservices. The request-based pricing means that you only pay for the work that your services actually perform. Also, because you are not responsible for the IT infrastructure you can focus on developing your application.

There are, however, some significant limitations. It is not intended to be used to deploy long-running services, such as a service that consumes messages from a third-party message broker. Requests must complete within 300 seconds. Services must be stateless, since in theory AWS Lambda might run a separate instance for each request. They must be written in one of the supported languages. Services must also start quickly; otherwise, they might be timed out and terminated.

|  |  |
| --- | --- |
| https://upload.wikimedia.org/wikipedia/commons/1/1e/Curation_bar_icon_info_35x35.png | **Case Study** - Building and Deploying Microservices with AWS Lambda and Semaphore:  <https://semaphoreci.com/community/tutorials/building-and-deploying-microservices-with-aws-lambda-and-semaphore> |

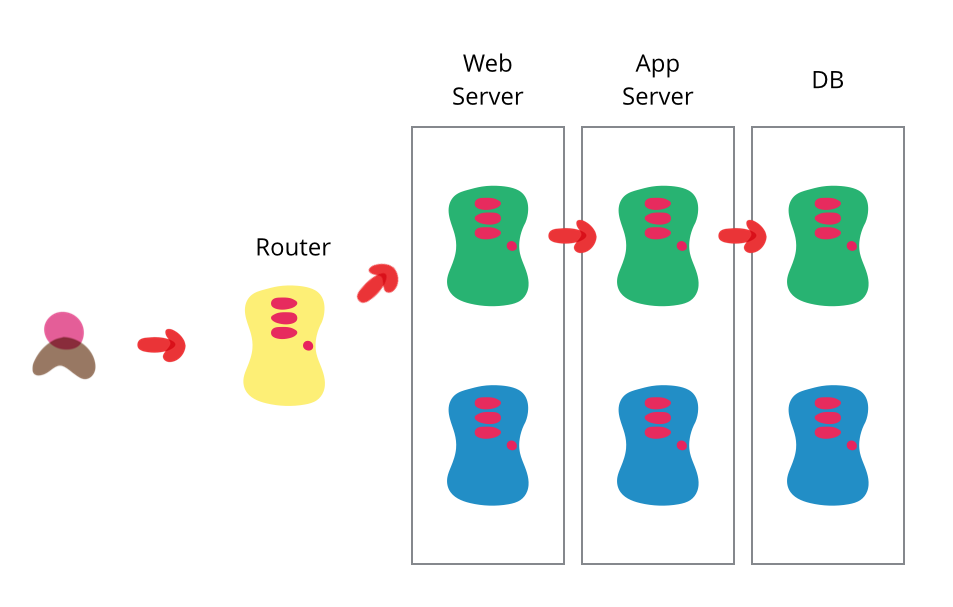
# Zero Downtime Deployment

## Blue Green Deployment

Modern development practices often distinguish between deploying and releasing software. Deployment is the step that involves getting the new code onto the servers. Releasing is the step where the new code begins to receive production traffic.

Blue-green deployment is a strategy for deploying and releasing software. It relies on maintaining two separate production-capable environments, nicknamed blue and green for ease of discussion.

According to post: <http://martinfowler.com/bliki/BlueGreenDeployment.html> by Martin Fowler, the basic concept behind blue-green deployment technique is that two sets of environments, each capable of serving your application in production, are maintained. These two environments should be nearly identical. By convention, these are referred to as the blue and the green environments.



Only one of these environments is active and receiving production traffic at any one time. In front of the web endpoints for these environments (either web servers or load balancers), a router or other traffic directing mechanism pushes all production traffic to the currently active environment.

Web Server

App Server

Database

Load Balancer



###### Running 2 environments – only one is active

When a new release is planned, it is deployed to the non-active environment. For blue-green deployments, the non-active environment functions as a final staging environment. It mirrors the production environment very closely and can be used for final testing before deciding to push changes live.

Web Server

App Server

Database

Load Balancer



###### New release deploy to non-active (blue) > change route to blue > blue is set active. (Green is running but not active, standby for rollback if new release has serious issue)

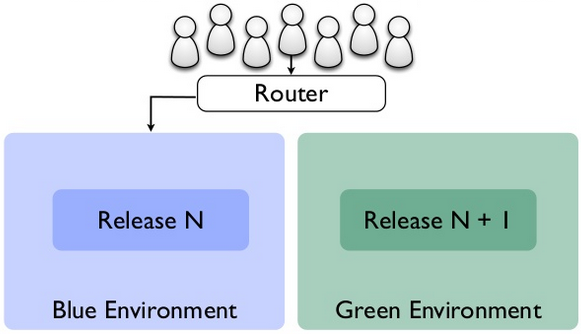
Once you have tested your deployment internally and have gained confidence in its robustness, you can release the new version quickly and easily by adjusting the routing mechanism. Basically, you flip the switch at the traffic directing layer so that all production traffic begins to move to your new software version. The previously active environment becomes non-active and your previous staging environment becomes your new production environment.

At this point, your previous software version is non-active, but still accessible. If your newly active deployment suffers from any serious issues, reverting to your previous version is as simple as modifying the routing mechanism again.

## Canary Release

Danilo Sato : “Canary release is a technique to reduce the risk of introducing a new software version in production by slowly rolling out the change to a small subset of users before rolling it out to the entire infrastructure and making it available to everybody.”

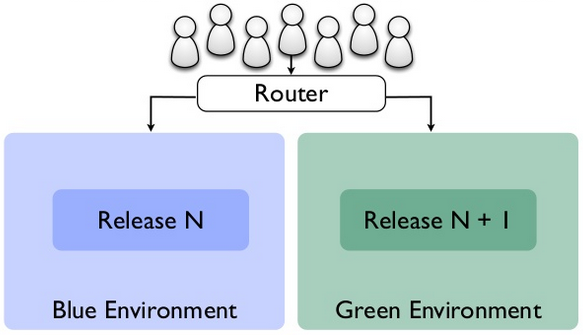
Similar to a Blue Green Deployment, you start by deploying the new version of your software to a subset of your infrastructure which no users are routed.



When you are happy with the new version, you can start routing a few selected users to it. There are different strategies to choose which users will see the new version: a simple strategy is to use a random sample; some companies choose to release the new version to their internal users and employees before releasing to the world; another more sophisticated approach is to choose users based on their profile and other demographics.

As you gain more confidence in the new version, you can start releasing it to more servers in your infrastructure and routing more users to it.

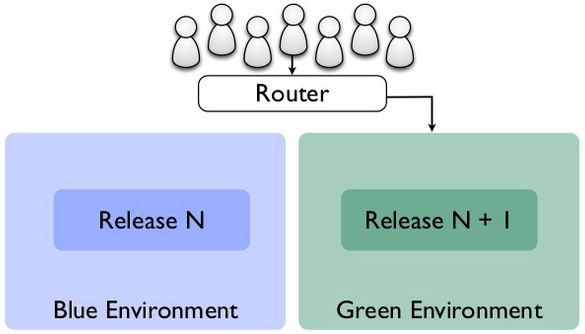
If you find any problems with the new version, the rollback strategy is simply to reroute users back to the old version until you have fixed the problem.



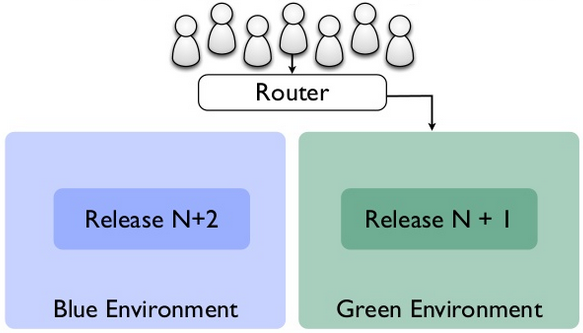
95%

5%

When it’s confident that the new release is working fine and ready for full launch, you can remove the route to the old release so now everyone can access the same latest version.



And if you have another new release, perform the same process cycle. New release is applied to the non-active environment, change route to the new release for a small group of users, full launch by remove route to old release when ready.



This pattern is said used by Facebook, spreading its updates at first to all its employees and all users if all goes well for a day.

# Scaling Microservices

Atul Saini: “With the velocity of data growing at the rate of 50% per year, the issue of scaling a Microservices architectures is critical in todays’ demanding enterprise environments. Just creating the Microervices is not sufficient. Scaling a microservices architecture requires careful choices with respect to the underling infrastructure and as well as the strategy on how to orchestrate the Microservices after deployment.”

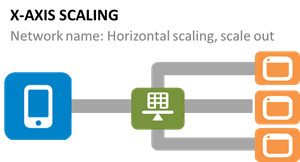
As microservices application is broken down into smaller components that may or may not be spread across separate server, so we may apply scalability to each microservice with different approach.

## The Art of Scalability

The Scale Cube is an interesting scaling strategies. Following is an overview of the three axes perspective on scale: x, y and z.

### X-axis scaling

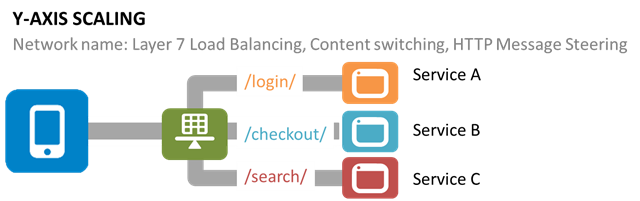
X-axis scaling is essentially a typical horizontal (scale out) scaling pattern implemented using a load balancer.



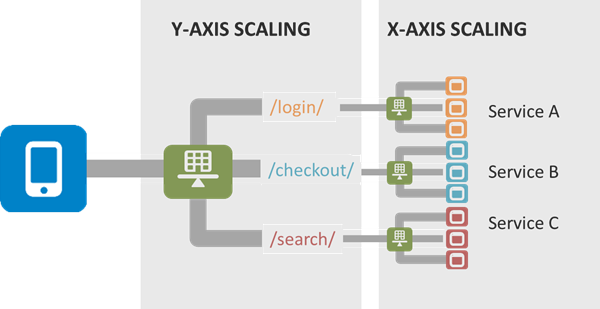
Each microservice is scaled out (along the x-axis) using an app proxy or application delivery controller (ADC). This allows operations to tune each app proxy or ADC based on the specific purpose of the microservice, improving performance by applying image optimization, compression or even caching where appropriate to the specific service. In a monolithic application, an ADC will be a better choice for this scalability model because of its ability to interpret requests and optimize responses with the benefit of context. In cloud-scale microservice architectures, an app proxy may be the better option when considering cost per service and the relatively simple delivery needs of a given service.

### Y-axis scaling

Y-axis scaling is essentially a layer 7-based sharding pattern when applied to infrastructure. Y-axis scaling relies on the decomposition of applications into services. It is highly appropriate for SOA or RESTful APIs that group like functionality into a service. For example, verb-based decomposition focused on "login" or "checkout" or noun-based decomposition with an emphasis on "customer" or "partner." The key is that there is some mechanism within each request - either in the URI or in the HTTP headers - that enable the app proxy or ADC to determine to which service the request needs to be forwarded.



Sharding can be implemented in the app, itself, using a routing object to dissect the URI or that functionality can be offloaded to the network and implementing using the data path programmability associated with an app proxy or ADC. This programmability allows operators or developers to implement targeted logic that dissects the URI and determines to which service the request should be directed. This pattern can be (and often is) implemented along with an X-axis scaling strategy for the specific service.

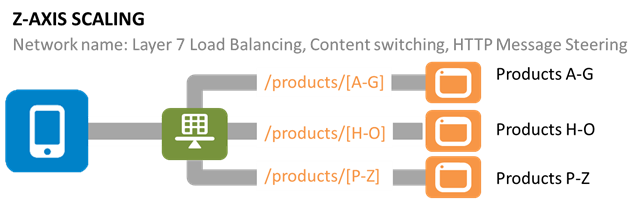


The combination of both Y and X axis scaling is increasingly a good choice for bifurcated networks which split "core" networking from "app" networking. The core network usually provides a significantly capable load balancing service managed by the network team while the app network includes app proxies or virtual ADCs that are managed by operations or developers.

### Z-axis scaling

Z-axis scaling is a cross between X and Y scaling strategies, using a data sharding-based approach. This strategy is commonly used to shard databases, but can also be used to scale monolithic applications based on some user characteristic.

Z-axis scaling is like X-axis scaling in that it relies on cloning of application instances. The difference is that some other component - like an app proxy or ADC - is responsible for distributing requests based on some other information, like the data being requested or the user identity. As long as the data is accessible to the app proxy or the ADC (increasingly iintermediaries have the ability to reach out and query databases or directories to obtain additional information)



When using Z-axis scaling each server runs an identical copy of the code. In this respect, it’s similar to X-axis scaling. The big difference is that each server is responsible for only a subset of the data. Some component of the system is responsible for routing each request to the appropriate server. One commonly used routing criteria is an attribute of the request such as the primary key of the entity being accessed. Another common routing criteria is the customer type. For example, an application might provide paying customers with a higher SLA than free customers by routing their requests to a different set of servers with more capacity.

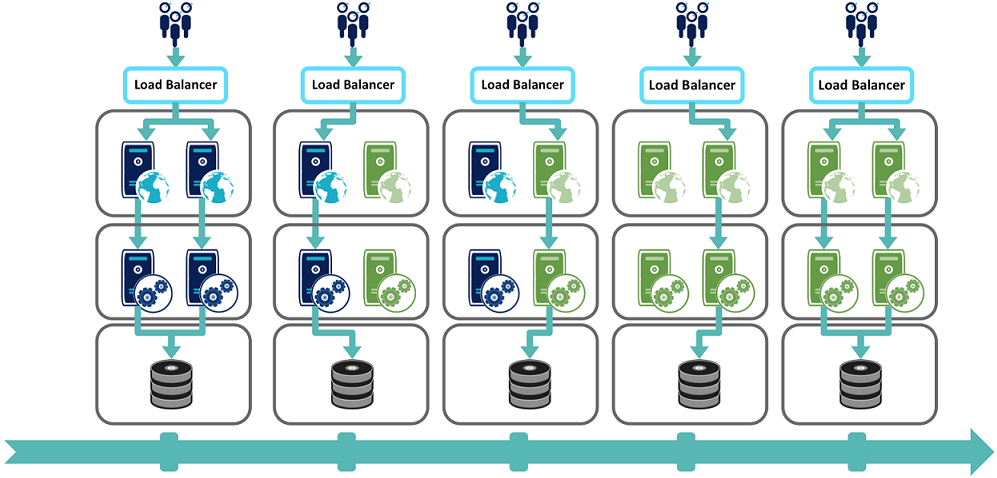
The point is that the scaling strategies associated with application architecture can be duplicated and/or augmented by the use of a app proxy or ADC. It is almost always the case that such an intermediary will be necessary to scale an application. That's because reality is that it's just as bad to let network logic (routing) seep into business logic as it is business logic to seep into the presentation (GUI) layer.

Keep your logics separate, and use the tools available to act on the scaling strategy best suited for your application or service.

|  |  |
| --- | --- |
| https://upload.wikimedia.org/wikipedia/commons/1/1e/Curation_bar_icon_info_35x35.png | **References**:   * [The Art of Scalability](http://theartofscalability.com) * [The Art of Scale: Microservices, The Scale Cube and Load Balancing](https://devcentral.f5.com/articles/the-art-of-scale-microservices-the-scale-cube-and-load-balancing) |

## Deployment Guideline for Microservices Cluster

## The aim is to associate the load distribution mechanism (Load Balancer) for the deployment:



Web  
Server

Application  
Server

Database  
Server

**In production**

**Step 1**

**Step 2**

**Step 3**

**Step 4**

Assume that there are 2 production lines running, both are in production with release N

**Step 1** – The load balancer disconnects one of the production lines and deploy the version N + 1 to the disconnected line.

**Step 2** – Once the server that run version N + 1 is ready to launch, the load balancer change route to the version N + 1 line

**Step 3** – Then deploy update version N + 1 to the disconnected line which is version N

**Step 4** – When ready, reconnect to the load balancing

# Scaling & Deployment approach in Lift MicroService Template

Lift MicroService Template provide lm script that can deploy package in cluster environment with docker swarm.

Repository: <https://github.com/dotography-code/LiftMicroservices>

Please refer the above link for the instruction about how to use the script. This document will provide you the information about what the script does.

## Prepare Cluster Environment (one time setup)

### Init Swarm

The command is to create docker cluster.

|  |
| --- |
| ./lm swarm init |

After it’s run successfully, it would create 4 nodes as Docker Virtual Machine as cluster environment:

Registry

Swarm Manager

Node 1

Node 2

* Docker Private Registry
* Docker Swarm Manager
* 2 Application Nodes
  + Node 1
  + Node 2

## First Deployment

### Build & Push Image

This step is to build the application, create docker image and push the image to Registry

|  |
| --- |
| ./lm swarm publish -n app -v 1.0.0 |

Following are the steps of the process:

Registry

Swarm Manager

Node 1

Node 2

Docker Image

* Build the package
* Build the docker image
* Tag the docker image
* Push to Registry

### Deploy

This is to deploy the new built container to each node of Swarm cluster.

|  |
| --- |
| ./lm swarm deploy -n app -v 1.0.0 -g blue -c 2 |

The process is to run docker container from the newly docker image from Registry into every swarm app node. From the sample, it would run at:

Registry

Swarm Manager

Node 1

Node 2

Docker Container

Docker Container

Docker Image

* Node 1 – app v1.0.0
* Node 2 – app v1.0.0

So now we have the cluster running and ready to test.

The test URL is http://<Swarm Manager IP Address >/test /id

Swarm Manager is working as load balancer and distribute traffic to the cluster node (default is round robin).

Node 1

Swarm Manager

Node 2



Docker Container

Docker Container

From the example, now we have 1 product line as blue, runs 2 instances on 2 nodes – each node run 1 instances of application version 1.0.0.

## Upgrade Application to New Version

Blue / Green deployment is a strategy to bring up a new version in parallel to the existing production line.

### Check the group currently running

|  |
| --- |
| ./lm swarm active |

This would return blue or green according to the current status. We will use the inactive group to deploy the new version. From the example above, current is blue so we will build the new version into green.

### Build & Push Image

This step is to build the application, create docker image and push the image to Registry

|  |
| --- |
| ./lm swarm publish -n app -v 1.0.1 |

Following are the steps of the process:

Swarm Manager

Node 1

Node 2

Registry

V 1.0.0

V 1.0.1

* Build the package
* Build the docker image
* Tag the docker image
* Push to Registry

### Deploy

This is to deploy the new built container to each node of Swarm cluster in the green group.

|  |
| --- |
| ./lm swarm deploy -n app -v 1.0.1 -g green -c 2 |

The process is to run docker container from the newly docker image from Registry into every swarm app node. From the sample, it would run at:

* Node 1 – add new instance app v1.0.1
* Node 2 – add new instance app v1.0.1

Swarm Manager



Registry

V 1.0.0

V 1.0.1

Node 1

V1.0.0

V1.0.1

Node 2

V1.0.0

V1.0.1

So now we have the new cluster group running and ready to test at port 8080. Note that it’s still not activated. The production line is still blue.

The test URL is http://<Swarm Manager IP Address >:8080/test /id

### Publish the New Version

Once you are happy with the new version and ready to launch, you can run this command.

|  |
| --- |
| ./lm swarm activate green |

This would switch the status of the two groups. The new production line would now be green which run at port 80 and the blue line is now change to port 8080.

Swarm Manager



Node 1

V1.0.0

V1.0.1

Node 2

V1.0.0

V1.0.1

## Remove Instance(s)

After new version is properly deployed, you may keep the old version running or remove it to reduce resources consumption.

|  |
| --- |
| ./lm swarm undeploy blue |

This would stop and remove the container in blue line.

# References

* Book: Building Microservices (Designing Fine-grained Systems) by Sam Newman
* <https://www.nginx.com/blog/deploying-microservices>
* <http://devops.com/2015/05/07/3-golden-rules-microservices-deployments>
* <https://dzone.com/articles/automate-codecommit-and-codepipeline-in-aws-cloudf>
* <http://docs.aws.amazon.com/lambda/latest/dg/welcome.html>
* <http://techbeacon.com/challenges-scaling-microservices>
* <http://martinfowler.com/bliki/BlueGreenDeployment.html>
* <http://martinfowler.com/bliki/CanaryRelease.html>
* <https://www.digitalocean.com/community/tutorials/how-to-use-blue-green-deployments-to-release-software-safely>
* <http://www.slideshare.net/thekua/patterns-for-continuous-delivery>
* <https://www.fiorano.com/blog/microservices/scaling-a-microservices-architecture-in-the-cloud>
* <http://blog.octo.com/zero-downtime-deployment>
* <http://theartofscalability.com>
* <https://devcentral.f5.com/articles/the-art-of-scale-microservices-the-scale-cube-and-load-balancing>