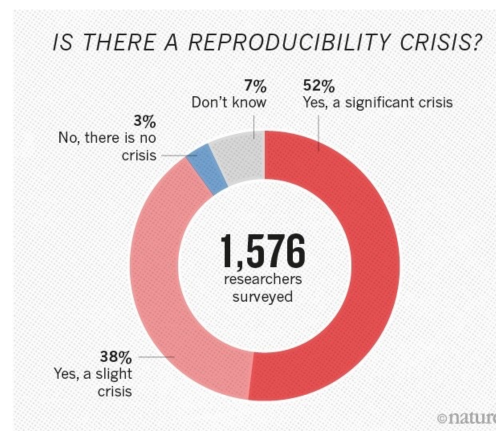
Container introduction

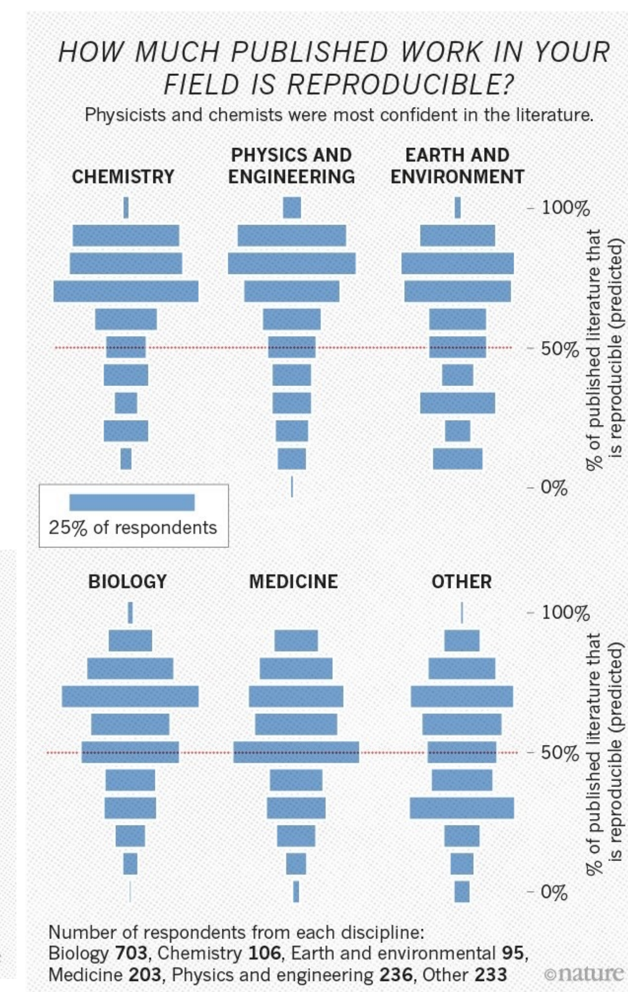
INFN

cc BY NC SA

What is Cloud Automation

- Simply put, Cloud Automation is **a set of processes and technologies** that allow to *automatize* several operations related to Cloud computing.
- Doing things *by hand* is rarely a good idea when complexity increases, and we have already seen several relatively complex technologies. This is closely linked to key topics such as **reproducibility**.
- For examples linked to biology, see e.g. "Cloud Computing May be Key to Data Reproducibility".
  - See also Nature, Vol. 533, 26 May 2016, pp. 452-454, "1,500 scientists lift the lid on reproducibility".

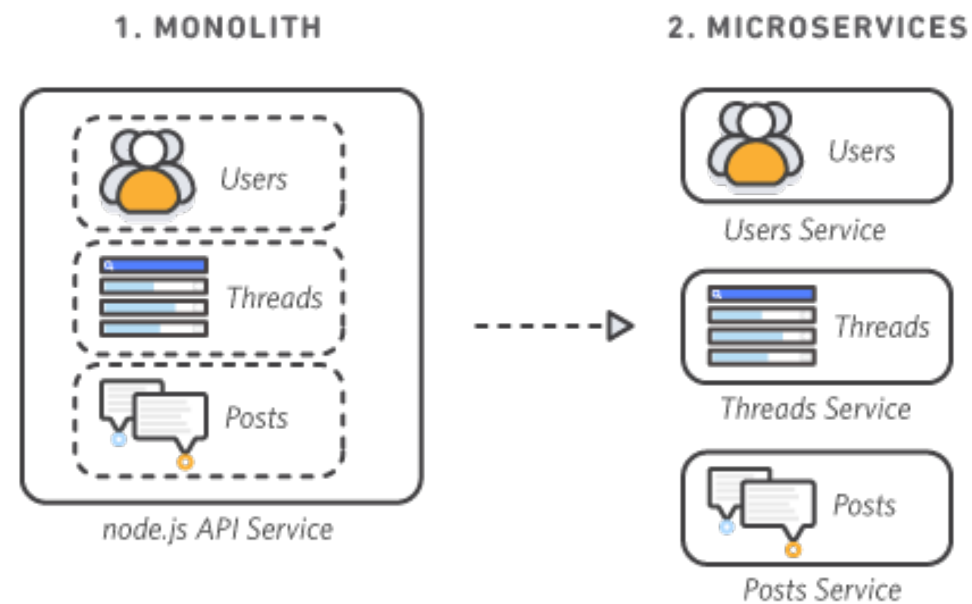




Microservices

- When discussing applications designed for the Cloud, you have already seen in the previous presentations the analogy of pets (each one is unique and irreplaceable) vs. cows (many identical instances of a functionally equivalent "item").
- **Microservices** are a way to build applications as **a collection of (potentially many) small autonomous services** vs. creating a big service (or anyway a few *fat ones*), called sometimes *monolith*.
- At high level, microservices reflect at the architectural level a **culture of autonomy and responsibility** in an organization: the single microservice can be developed and managed independently by different teams.
- In microservices architectures, the multiple, independent processes communicate with each other through the network.

Application architectures



Monoliths vs. microservices

Monolithic Applications

- Do everything
- Single application
- You have to distribute the entire application
- Single database
- Keep state in each application instance
- Single stack with a single technology

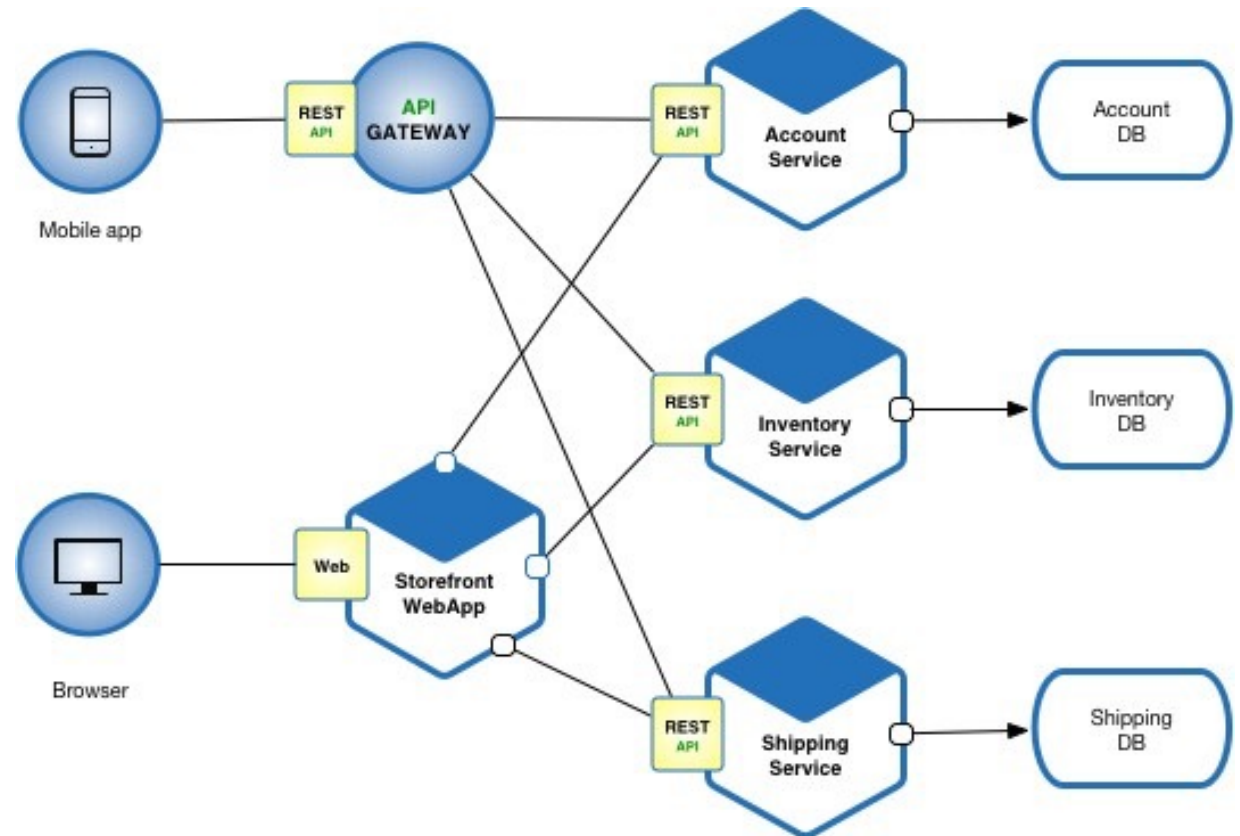
Microservices

- Each has a dedicated task
- Minimal services for each function
- Can be distributed individually
- Each has its own database
- State is external
- Each microservice can adopt its own preferred technology

Adapted from AWS

An example of a microservice architecture

- How to structure **an e-commerce application** (from https://microservices.io/patterns/microservices.html)



All good with microservices?

- Of course not. **There are cases when monolithic applications might make more sense**. With microservices, remember that you should:
  - Deploy each microservice independently.
  - Worry about microservice orchestration.
  - Unify the format of software integration and deployment pipelines.
  - Compared to monolithic systems, there are more services to monitor.
  - Since they form a distributed system, the model is more complex than with monoliths.
- **However, with microservices:**
  - Reliability is much easier, because (for example) if you happen to break one microservice, you will affect only one part, not the entire app.
  - Scalability is much better. With monoliths, horizontal scaling might be impossible and, when possible, it is connected to scaling the entire app, which is typically inefficient.

Container orchestration

- We just saw that **microservice architectures** are based on the composition of many independent (but communicating) services.
- We therefore need to explore how to effectively *orchestrate* many containers across distributed hosts. This is what we call **container orchestration**.
- Use «*container orchestration*» to automate and manage tasks such as:
  - Provisioning and deployment
  - Configuration and scheduling
  - Resource allocation
  - Container availability
  - Scaling or removing containers based on balancing workloads across your infrastructure
  - Load balancing and traffic routing
  - Monitoring container health
  - Configuring applications based on the container in which they will run
  - Keeping interactions between containers secure

Docker orchestration tools

Tools of Container Orchestration

Amazon ECS
FROM AMAZON

Azure Container Services
FROM MICROSOFT

Docker Swarm
DOCKER OPENSOURCE TOOLS

Google Container Engine
FROM GOOGLE CLOUD PLATFORM

Kubernetes
DOCKER OPENSOURCE TOOLS

CoreOS Fleet
FROM COREOS

Mesosphere Marathon
FROM MARATHON

Cloud Foundry's Diego
FROM CLOUD FOUNDRY

Docker Swarm (1)

- `Docker Swarm` is the traditional way of orchestrating containers with Docker. Compared to other methods we'll see later, it is relatively easy to use. Its <u>main features</u> are:
  - **Cluster management integrated with Docker Engine**: no other software than docker is needed.
  - **Decentralized design**: this means that any node in a Docker Swarm can assume any role at runtime.
  - **Scaling**: the Swarm manager can automatically scale up and down services, adding or removing tasks.
  - **Desired state reconciliation**: if something happens to a Swarm cluster (e.g. some containers crash), the Swarm manager will try to reconcile the state of the cluster to its desired state (e.g. bringing up some more containers).

Docker Swarm (2)

- `Docker Swarm` features, continued:
  - **Multi-host networking**: the Swarm manager can handle an overlay network spanning your services.
  - **Service discovery**: there is a DNS server embedded in each Swarm. The Swarm manager discovers services and assigns to each of them a unique DNS name.
  - **Load balancing**: you can specify how to distribute services among nodes.
  - **Secure by default**: the communication among all nodes in a Swarm cluster is protected by the cryptographic protocol called TLS (Transport Layer Security).
  - **Rolling updates**: if anything goes wrong, you can roll-back a task to a previous version of the service.

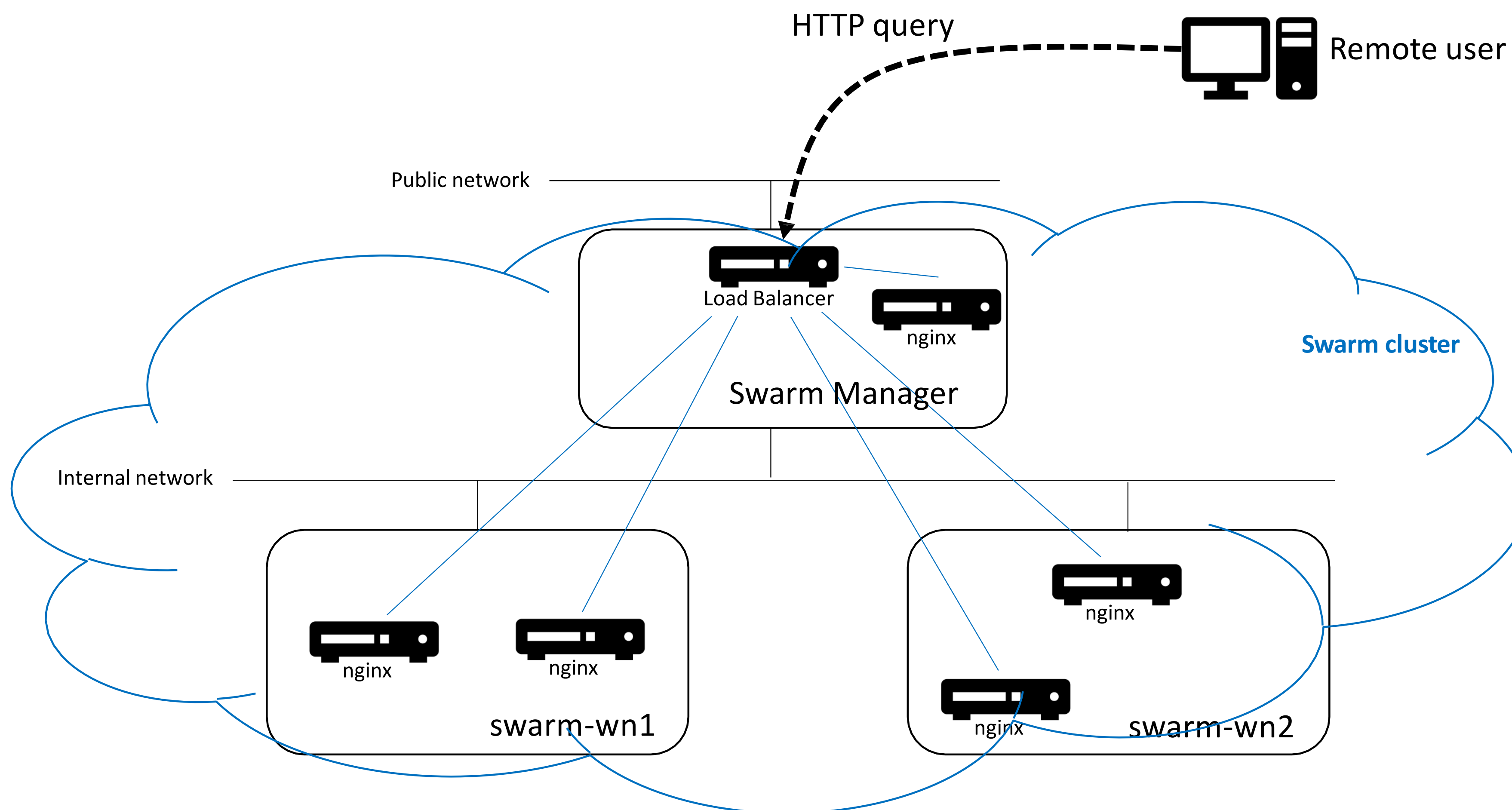
Hands-on with Docker Swarm

- We’ll loosely follow https://docs.docker.com/engine/swarm/swarm-tutorial/.
- For this hands-on, we need **three VMs with Docker installed**.
  - One of these machines will be the manager of the Swarm cluster, the other two will be called workers.
  - We’ll use our devopsXX; in order to have 3 VMs, you need ~~to **create 2**~~ **new VMs**; do it now and call devopsXX “manager”
  - **Important: make sure that Docker is installed on all three VMs.**
- We also need the IP addresses of the 3 machines involved, as well as the following open ports for all of them, to allow communication among the nodes (once you have your 3 VMs, **properly set up the security groups**):
  - **TCP port 2377** for cluster management communications.
  - **TCP and UDP port 7946** for communication among nodes.
  - **UDP port 4789** for overlay network traffic.

Docker Swarm hands-on: our use case

- To make things simple and quick, we'll use a Docker Hub container called "nginx"
  - Nginx is a commonly used web server (see https://nginx.org/en/), like Apache.
- We'll create a ***Swarm service*** based on the nginx container and deploy it in 5 instances, distributed across 2 VMs (swarm-wnX1 and swarm-wnX2).
  - All these containers will *not* be directly accessible from the Internet. So, in the end we'll have 5 web servers.
- We'll then **deploy a load balancer** on a 3rd VM (the manager). The load balancer will be reachable via a **public IP address**.
  - When people hit this IP address, the load balancer will route our requests to one of the nginx containers on *swarm-wnX1* or *swarm-wnX2*.

Docker Swarm: our architecture



Create a Swarm cluster

INFN

- Login to the VM that should become the "Swarm manager" (the one you called "manager"=devopsX).
- On the manager, issue the command
  - `docker swarm init --advertise-addr <MANAGER-PRIVATE-IP>`
  - This initializes a Swarm cluster and tells the workers about the IP address of the Swarm manager. Note that this should be **the manager's private IP address**, not the public one.
  - Docker answers confirming that the current node is now manager and gives us the command to add a worker to the Swarm cluster. **Note it down**.
- Now log in to swarm-wnX1 and swarm-wnX2, and *on each of them* issue the command reported above by the manager
  - It should be something like `docker swarm join –token <token> <ip_addr>:2377`
- On the manager, issue the command `docker node ls` to view the **current state of the Swarm cluster**.
  - It should show the manager and the two workers, all in the "active" state. There are no running services in the cluster yet.

Create a Swarm service

INFN

- We will now **create a “service”**. We have to define:
  - How to name it – we’ll call it “**web_swarm**”.
  - The container image it is based on (nginx, found on DockerHub).
  - The port that can be used to contact the service.
  - How many replicas of the service we want to deploy.
- This is the command we have to issue on the manager:

```
docker service create --replicas 3 -p 8082:80 --name web_swarm nginx
```

  - With this command, we create 3 docker containers, each one based on the nginx image.
  - These containers will be automatically distributed across our Swarm cluster. Each container will expose port 80, which will be mapped to port 8082 on a VM host (swarm-wn1 or swarm-wn2).

Check the status of the Swarm service

INFN

- The **status of our service** can be checked on the manager with

  ```
  docker service ls
  ```

  - It will take some time before the service is shown as replicated 3 times, as requested – just repeat the command until it shows 3/3 replicas.
- In order to see **where (i.e. on which nodes) the service was distributed by Swarm**, issue this command on the manager:

  ```
  docker service ps web_swarm
  ```

- Once you have the 3 `web_swarm` replicas running, log in to either swarm-wn1 or swarm-wn2 and issue this command there:

  ```
  docker ps
  ```

  - You should see that one or more nginx containers are running on the node.

How to access the `web_swarm` service

- Remember that so far, the nodes of the Swarm cluster are only reachable via their private IP addresses. Therefore, we cannot directly use a browser to reach the web servers.
- But internally they can be reached (look back at the architectural diagram). So, log in e.g. to the manager and issue the command

  `curl http://<private_ip_address_of_VM1>:8082/` (or VM2)

  - You should get an answer. **Or not?**
  - Note that you will get an answer even if there is no `web_swarm` container running on VM1 (or VM2). **How can you prove that?**

Scaling up or down and draining

- When we created our service, we specified `--replicas 3`. If you want to **scale the service** to another number of replicas, just issue this command on the manager:

```
docker service scale web_swarm=7
```

- What is happening? On the manager, check with

```
docker service ls
docker service ps web_swarm
```

- Now suppose that you want to remove the service `web_swarm` from e.g. swarm-wn2 (because, for example, you want to shut it down for any reason). This is called draining a node. Try this:

```
docker node update --availability drain <VM2>
```

  - What is happening? Check with `docker service ps web_swarm`.

Load balancing the web servers

- We now want to create a **load balancer** on the manager node.
  - Its purpose is to expose a public IP address which will be reachable from the Internet and balance the queries to that IP address to the `web_swarm` services that are deployed in the Swarm cluster.
- The same nginx container that we previously used to create web servers can also be configured to act as load balancer. We just need to have a suitable nginx configuration file.
  - In this configuration file, we need to list the IP address (the private IP addresses, in our use case!) of all the hosts participating to the Swarm cluster.
  - That is, the **private IP addresses** of the manager, swarm-wn1 and swarm-wn2.

Create and run the load balancer

- **On the manager**, create the following Dockerfile in the same directory where you have put `nginx.conf`:

```
FROM nginx
COPY nginx.conf /etc/nginx/nginx.conf
```

- We can now build and then run our container with the load balancer configuration with commands we already know:

```
docker build -t load_balancer .
docker run -p 8080:80 -d load_balancer
```

- If we now open `http://<manager_public_ip>:8080/`, we should get a web page displayed. **Try it out now**.
  - From which `web_swarm` node is the answer coming? In the `nginx.conf` file we told the web server to log some information. Look at this information with the following command:

```
docker logs -f <load_balancer container>
```

The nginx configuration for load balancing

- On the manager, create this file and call it `nginx.conf`:

```
worker_processes 1;
events { worker_connections 1024; }
http {
  sendfile on;
  upstream swarm_cluster {
    server <manager_ip_addr>:8082;
    server <VM1_ip_addr>:8082;
    server <VM2_ip_addr>:8082;
  }

  server {
    listen 80;
    location / {
      proxy_pass http://swarm_cluster;
    }
  }
log_format upstreamlog '[$time_local] from $remote_addr to $upstream_addr';
access_log /var/log/nginx/access.log upstreamlog;
}
```

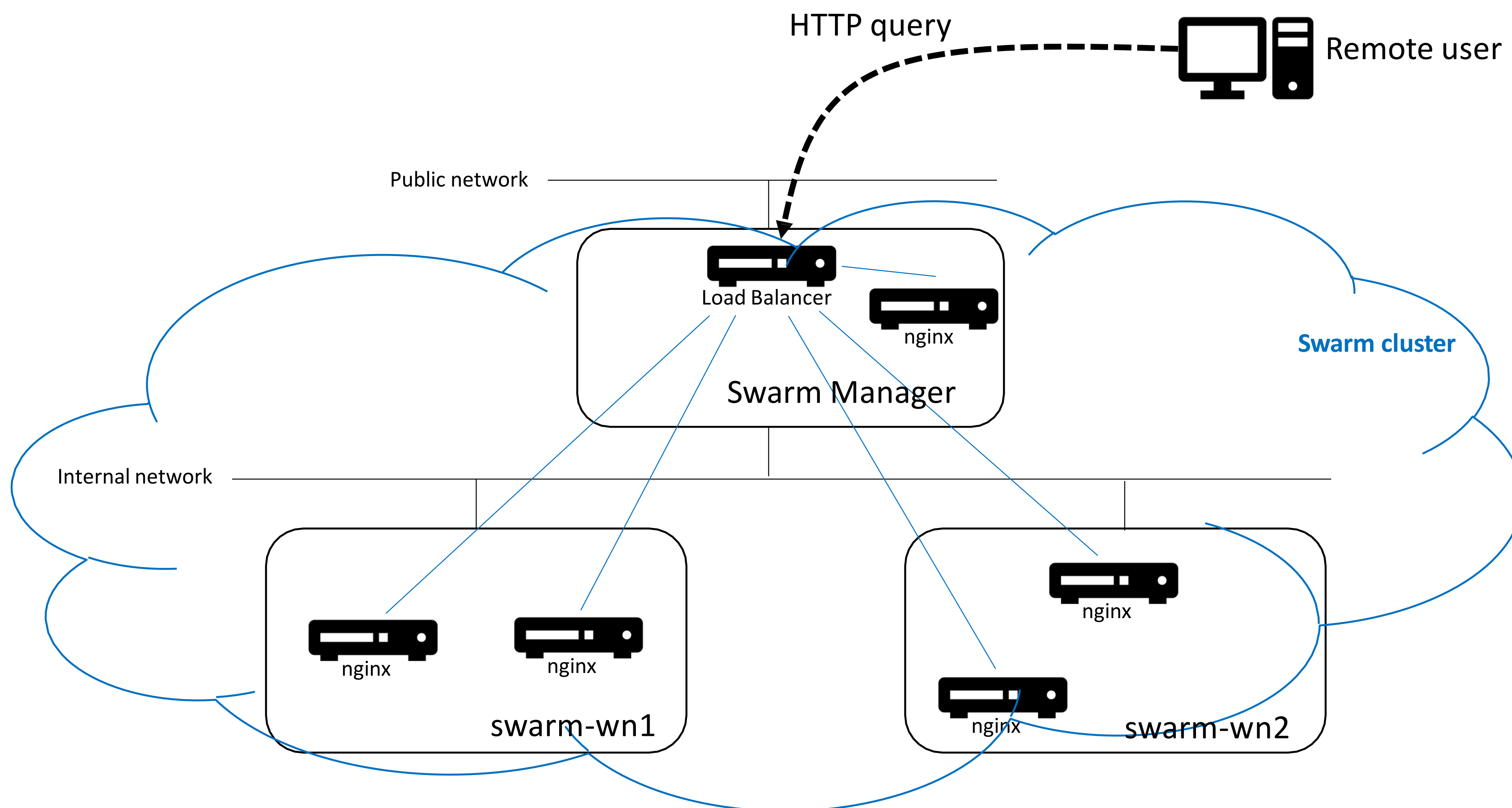
A few notes

- Docker Swarm services are persistent. Try to shut down all 3 nodes and then start only the manager. You will see that the manager brings up all replicas automatically on itself.
  - The load balancer configuration, on the other hand, is a stand-alone container and does not automatically restart.
- Remove a Swarm service with:

```
docker service rm <service_name>
```

- An interesting point is to combine Docker Swarm with custom Docker images or with Docker Compose. This is left as an exercise.

Docker Swarm: our architecture



Infrastructure as Code (1)

- With the idea of **Infrastructure as Code (IaC)**, instead of manually creating the infrastructure we need for our applications (e.g. virtual machines, disk volumes, installations, configurations), we **define what we want** through machine-readable definition files.
  - IaC is based on the realization that "**Complexity kills Productivity**": it therefore aims to simplify how you can realize complex infrastructures and set-ups.
- There are many tools that allow us to combine automation with virtualization. With IaC, all the specifications for the infrastructure we are generating should be explicitly **written into configuration files**.

Infrastructure as Code (2)

INFN

- Some of the most popular tools for IaC are **Puppet** (https://puppet.com), **Ansible** (https://www.ansible.com), **Terraform** (https://www.terraform.io) and **Chef** (https://www.chef.io/chef/). Docker itself provides some form of IaC.
- While we won't explore any of these in detail in this course, it is important to highlight that it is **fundamental that whatever you do with your code and data should be reproducible and manageable**.
- You are therefore **encouraged to use automated installation and configuration tools** in your work, also because they enable you to fully profit from the DevOps paradigm we have already seen (Continuous Integration, Continuous Delivery, Deployment Orchestration).

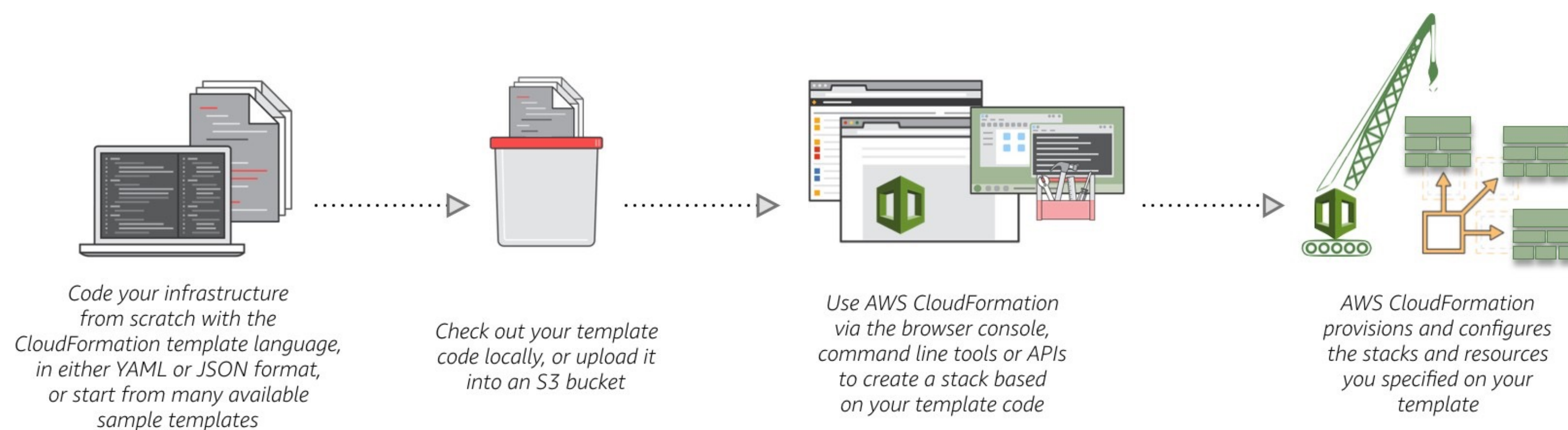
Template-based orchestration

INFN

- There are several **templating mechanisms** that can be used to describe and provision resources needed by an application in a Cloud infrastructure.
- In some sense, this extends what we have seen e.g. with Docker Swarm to cover *any requirements* your applications might have and automatize your deployments in the Cloud.

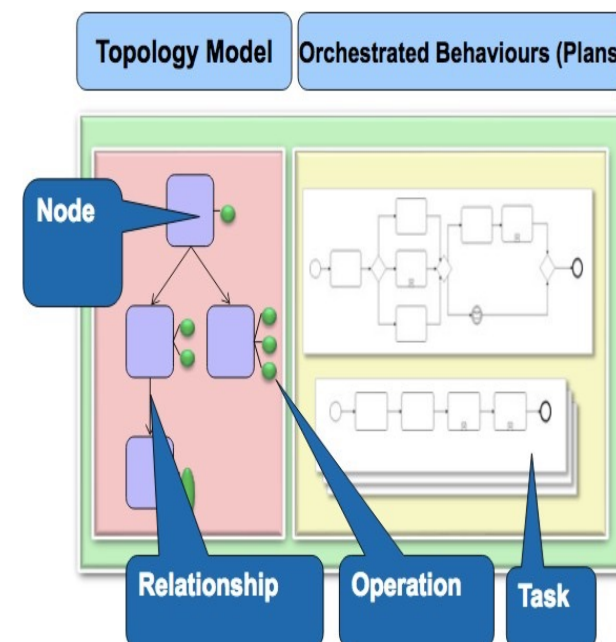
AWS CloudFormation

- The Amazon way of defining a complete topology for an application is through the CloudFormation language.

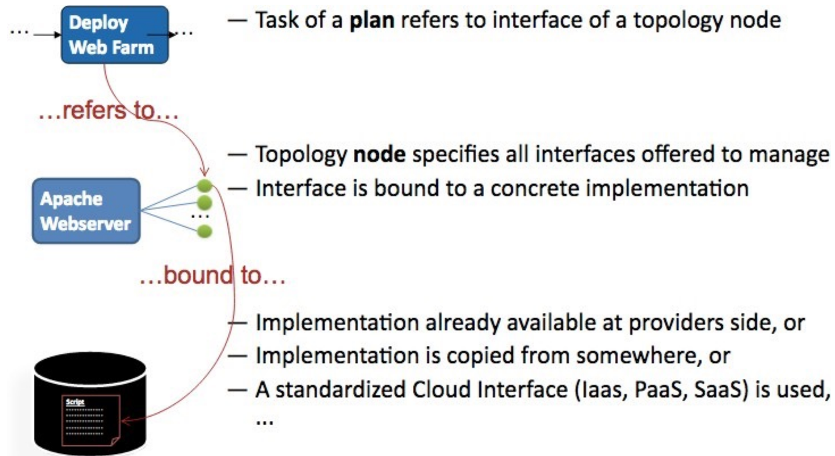


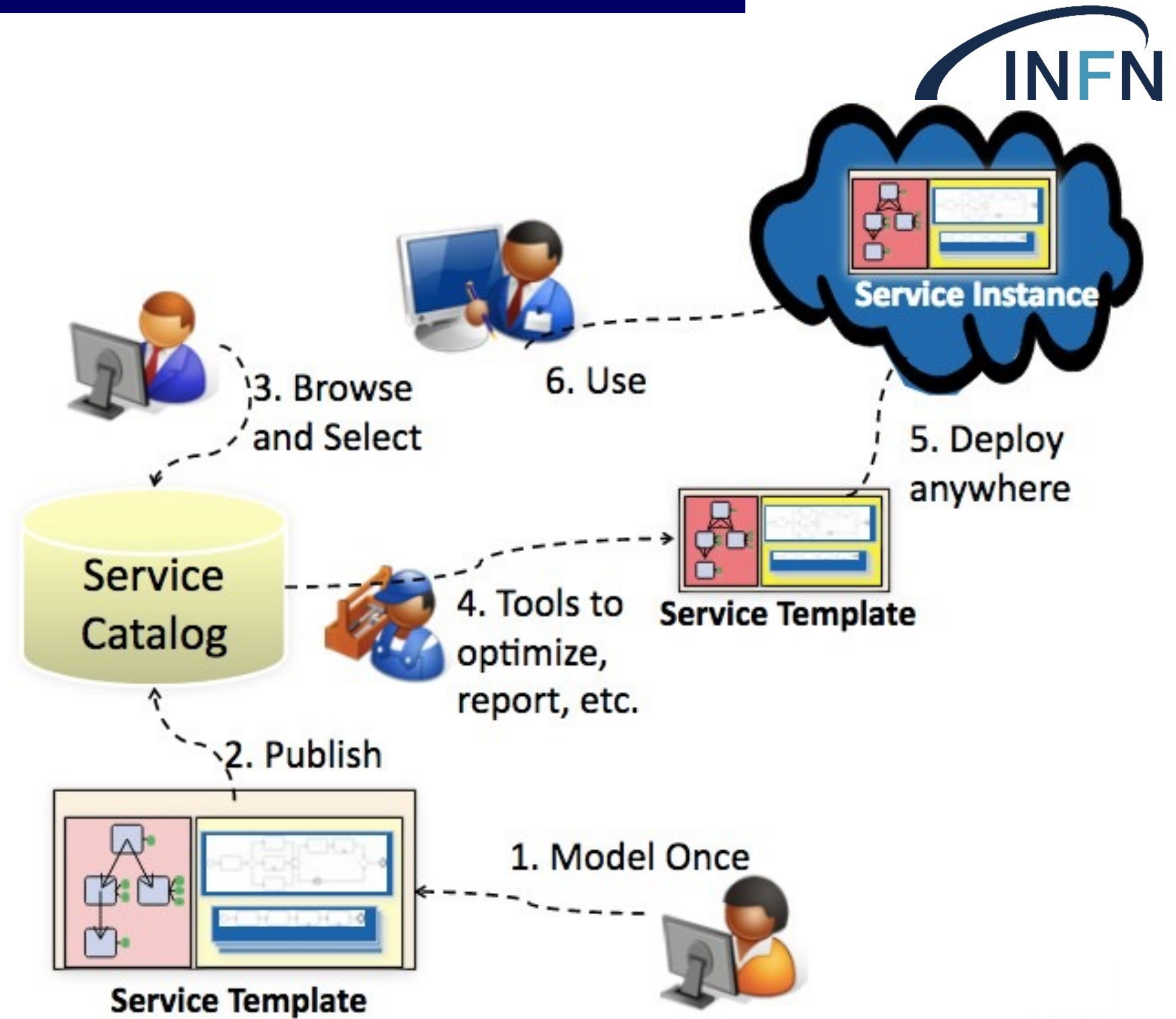
TOSCA

- AWS CloudFormation is Amazon-specific. As such, it can only be used with AWS.
- **TOSCA** (Topology and Orchestration Specification for Cloud Applications) is on the other hand a public standard:
  - ***It is an OASIS (https://www.oasis-open.org/) standard language to describe a topology of cloud-based web services, their components, relationships, and the processes that manage them***
- It standardizes the language to describe:
  - The structure of an IT Service (its topology model) .
  - How to ***orchestrate operational behavior*** (plans such as build, deploy, patch, shutdown, etc.) .
  - ***A declarative model*** that spans applications, virtual and physical infrastructures.



Vision





Automation of the release pipelines

- Strictly related to the microservice architecture is the concept of **DevOps**.
- DevOps is a pattern for developing applications where **Development and Operation practices tightly integrate**.
  - In other words, rather than (1) writing a full "production level" application, (2) releasing it and then (3) waiting for operational feedback, the DevOps application release process is much more **agile**, and it follows **tight release and feedback schedules**.
  - This is a concept that extends beyond the people who practically do development and operations. It includes end users as soon as this is possible.

Release early, release often

- The DevOps *mantra* is "**release early, release often**": this implies utilizing a set of *tools and processes* to facilitate **automation, monitoring and continuous integration** of all the involved components (microservices, for example) to quickly complete the development and delivery cycles.
- This is an example of **risk reduction** in software development.

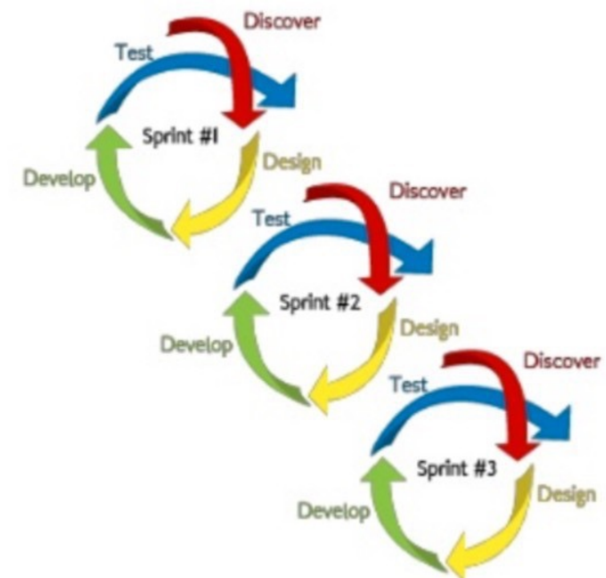
run in

short cycles

and

feedback is

frequent,



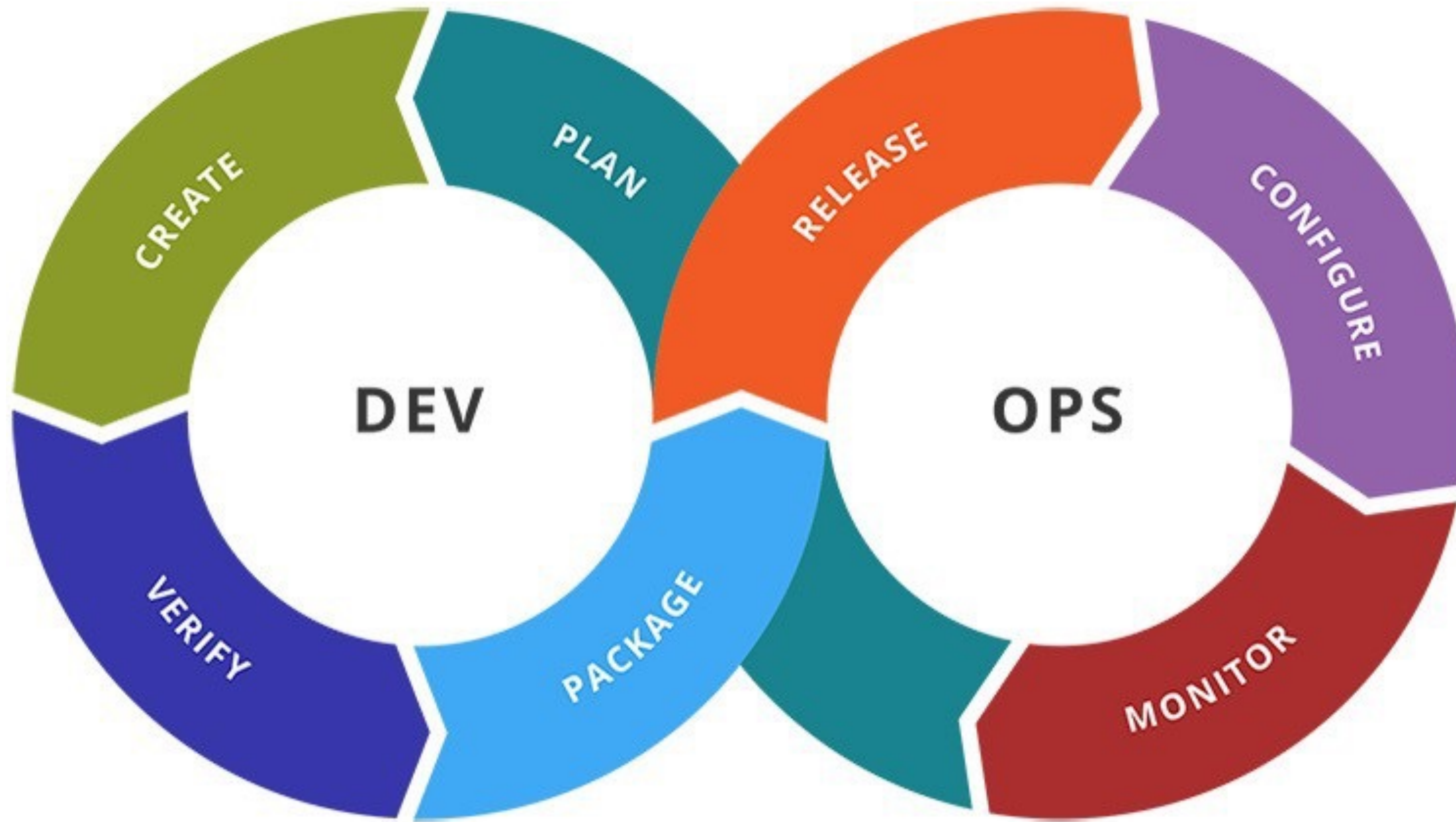
increasing **value** to the customer,

encouraging **quality**,

and ensuring **cost effectiveness.**

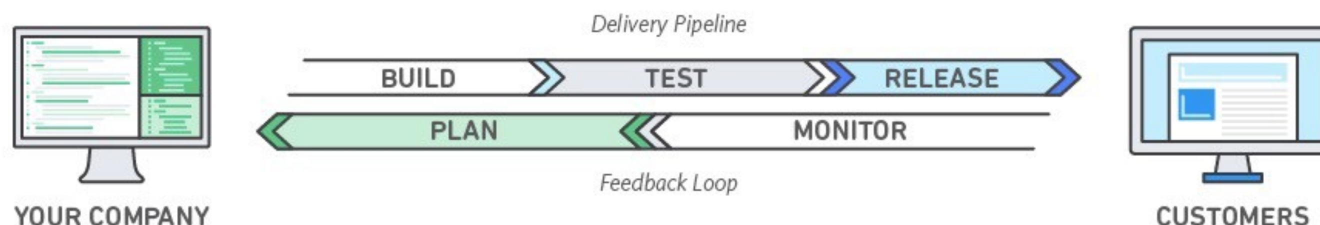
Source: https://medium.com/@warren2lynch/scrum-philosophy-release-early-release-often-a5b864fd62a8

DevOps



Source: https://nickjanetakis.com/blog/what-is-devops

DevOps benefits



- **Speed** – microservices & continuous delivery
  - Innovate faster
  - Better adapt to changing requirements
- **Rapid Delivery** – continuous integration and delivery
  - Higher release frequency
- **Reliability** – continuous integration and delivery, monitoring & logging
  - Ensure the quality of application updates
- **Scale** – automation, treat infrastructures *as code*
  - Operate and manage infrastructures and development processes at scale
- **Improved Collaboration**
  - Less friction, more effective teams
- **Security** - automated compliance policies, fine-grained controls, configuration management
  - Move quickly but preserve control and compliance

The DevOps principles

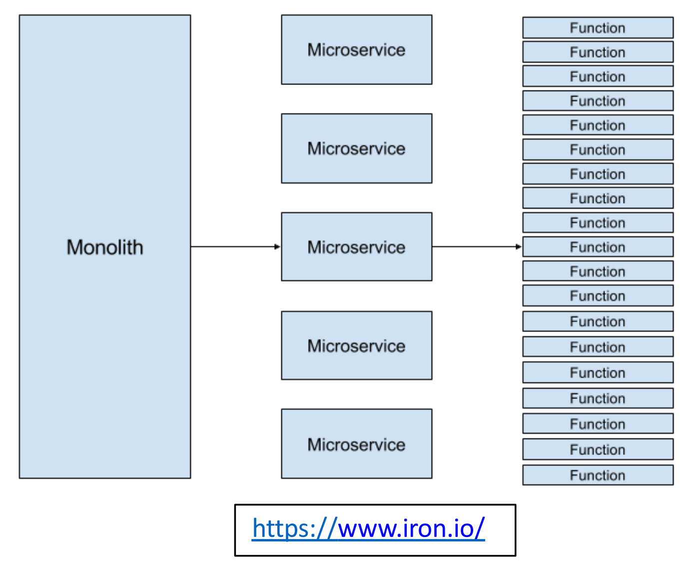
INFN

- DevOps is a comprehensive way of thinking **covering all the stages of an application lifetime**.
- It is particularly applicable to **distributed, microservices-based applications**, which we typically find in Cloud environments.
- It is therefore important to know its main principles and **try to apply them whenever we write applications, be they small or big**. Let's now see them in some detail.

Not covered in detail in this course

Serverless technologies

- With **serverless technologies**, we perform another step toward automating and facilitating writing and using applications and Cloud resources.
  - Remember that **what eventually matters are the applications, not the infrastructure.**
- With **serverless**, a Cloud provider is responsible for executing a piece of code, written by you, by **dynamically finding and allocating the resources needed by the code**.
- In serverless, your code is typically structured around a set of stateless functions. Thus, serverless computing is also called **Functions as a Service, or FaaS**. We won't cover FaaS in detail in this course, but it is an important concept.
  - The running of the serverless functions can be **triggered** by some conditions, such as for example database events, file uploads, scheduled events, various alerts, etc.
  - Structuring an app around stateless functions is consistent with the idea of **microservices** we have already seen. This time, however, we focus just on the app code, and deal as little as possible with resource provisioning and deployment.



https://www.iron.io/

Conclusions

Container orchestration automates the deployment, management, scaling, and networking of containers. Anyone that need to deploy and manage hundreds or thousands of Linux containers and hosts can benefit from container orchestration.

Container orchestration can be used in any environment where you use containers. It can help you to deploy the same application across different environments without needing to redesign it. And microservices in containers make it easier to orchestrate services, including storage, networking, and security.

Containers give your microservice-based apps an ideal application deployment unit and self-contained execution environment. They make it possible to run multiple parts of an app independently in microservices, on the same hardware, with much greater control over individual pieces and life cycles.

Managing the lifecycle of containers with orchestration also supports DevOps teams who integrate it into CI/CD workflows. Along with application programming interfaces (APIs) and DevOps teams, containerized microservices are the foundation for cloud-native applications.