Container Orchestartion

Doina Cristina Duma (aiftim<at>infn.it)

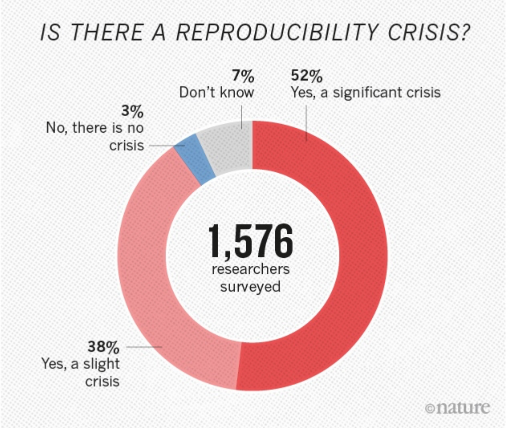
Big Data Analytics

9-12 Dic. 2019, Bologna

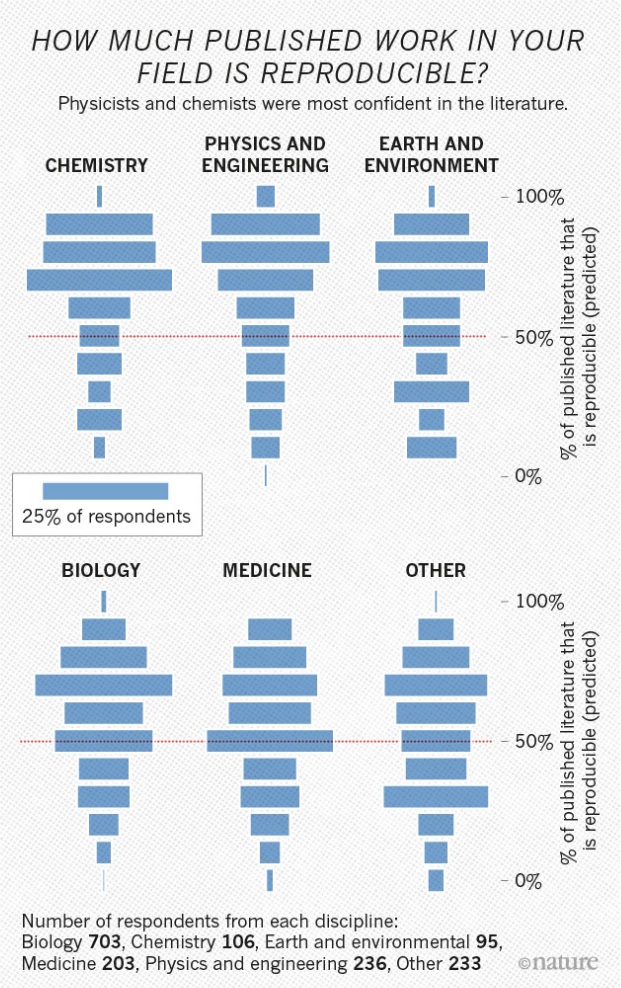
(many slides – courtesy of Davide Salomoni)

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".



IS THERE A REPRODUCIBILITY CRISIS?



HOW MUCH PUBLISHED WORK IN YOUR FIELD IS REPRODUCIBLE?

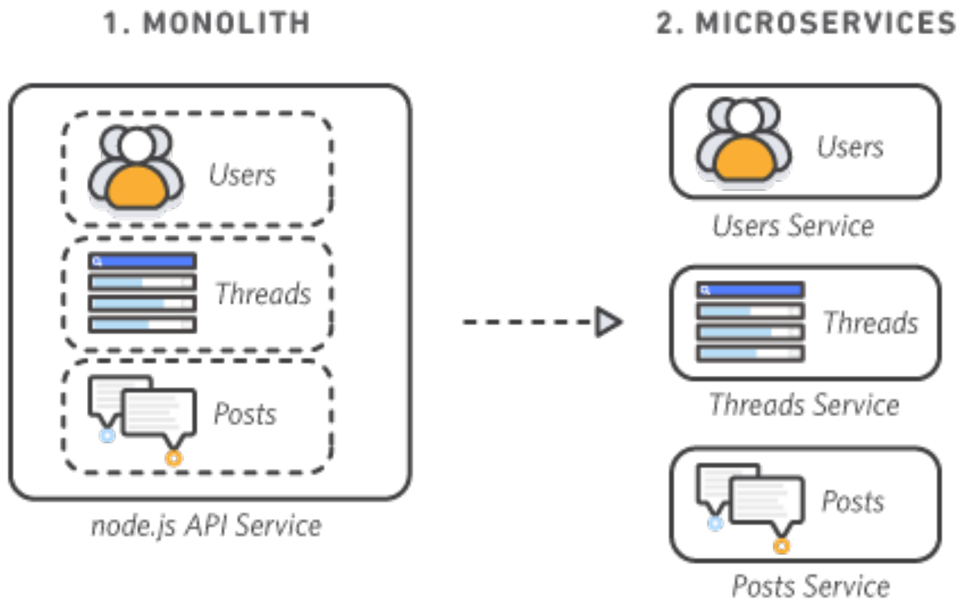
Physicists and chemists were most confident in the literature.

Number of respondents from each discipline: Biology **703**, Chemistry **106**, Earth and environmental **95**, Medicine **203**, Physics and engineering **236**, Other **233**

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

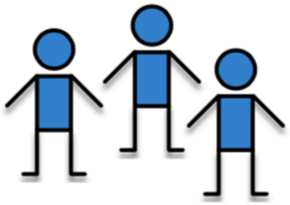
- Do everything
- Single application
- You have to distribute the entire application
- Single database
- Keep state in each application instance
- Single stack with a single technology

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

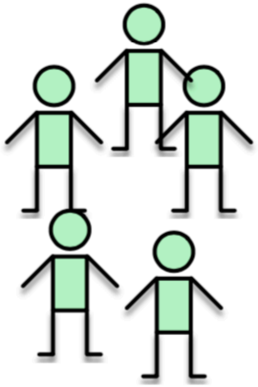
Organization in a monolith



Frontend

Orders, shipping, catalog

Classic teams:
1 team per “tier”



Backend

Orders, shipping, catalog

Database

Orders, shipping, catalog

From: LaToza, George Mason University, Fall 2017

Organization around business capabilities in microservices

Orders
Shipping
Catalog

Example: Amazon

Teams can focus on one business task
And be responsible directly to users

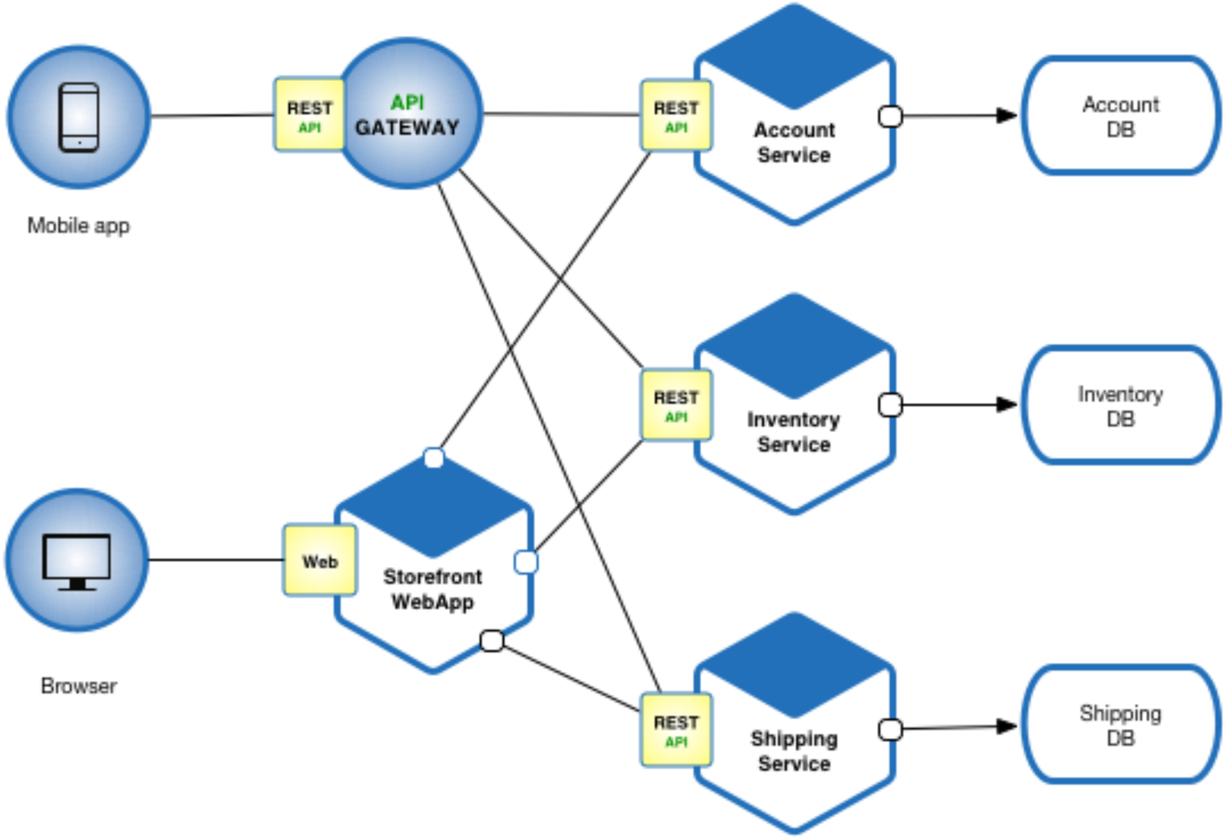
“Full Stack”

“2 pizza teams”

From: LaToza, George Mason University, Fall 2017

An example of a microservice architecture

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



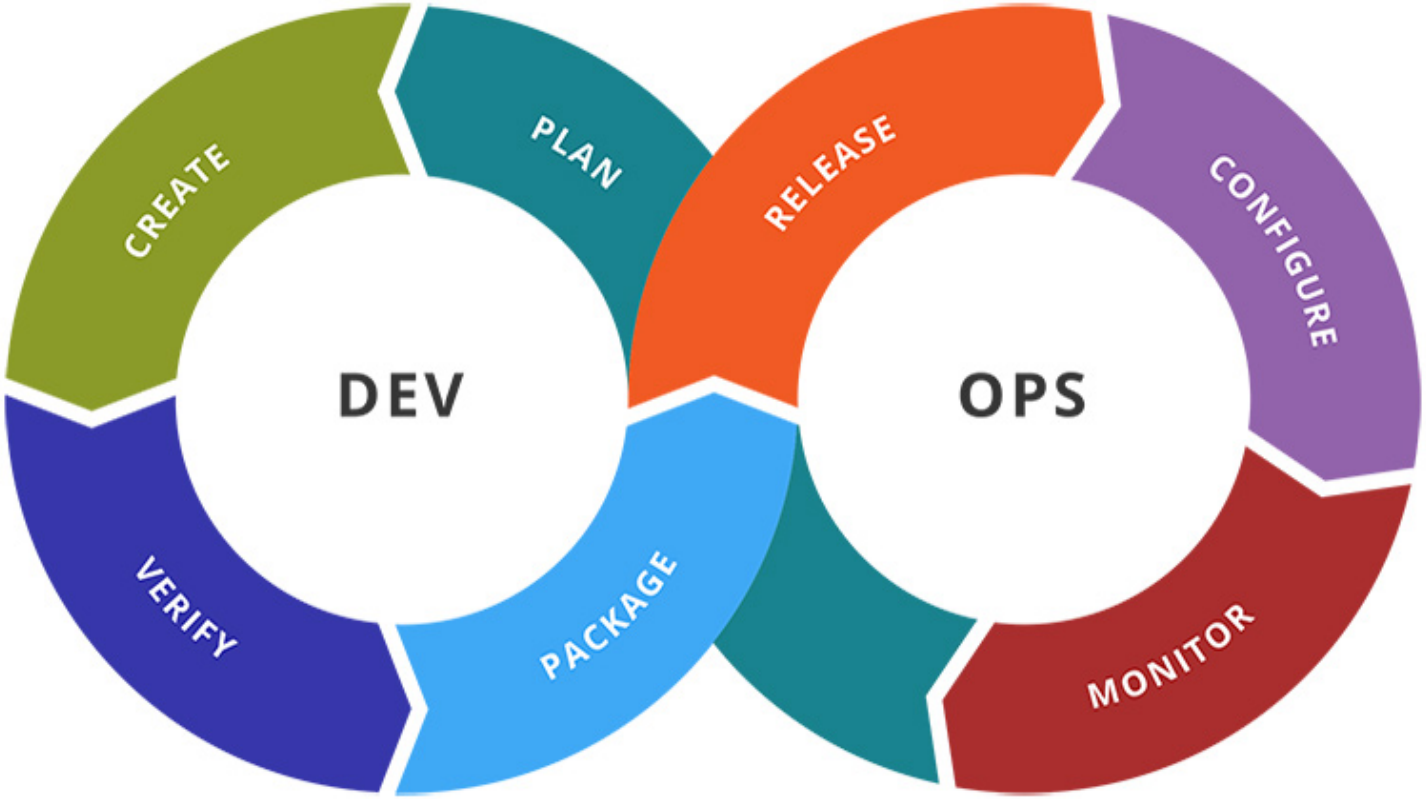
All good with microservices?

- 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.

Automation of the release pipelines

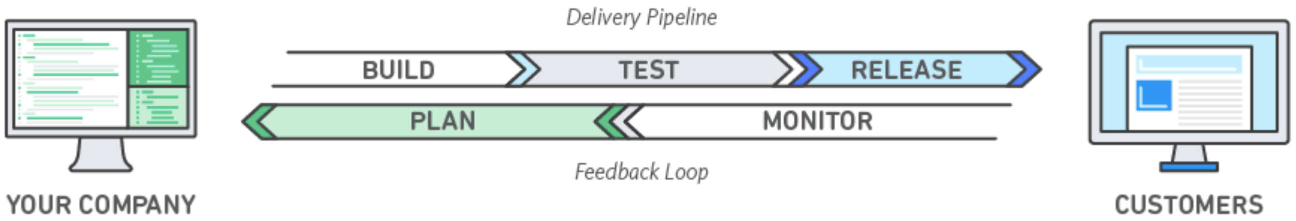
- Strictly related to the microservice architecture is the concept of **DevOps**.
- DevOps is a pattern for developing applications where **Development and Operation practices tightly integrate**.
  - In other words, rather than (1) writing a full "production level" application, (2) releasing it and then (3) waiting for operational feedback, the DevOps application release process is much more **agile**, and it follows **tight release and feedback schedules**.
- 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.

DevOps



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

DevOps benefits



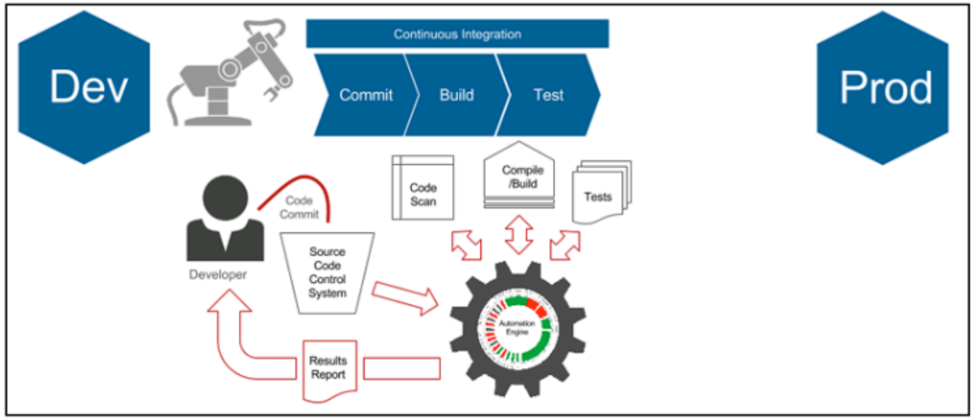
- **Speed** – microservices & continuous delivery
  - Innovate for customers faster
  - Adapt to changing markets better
  - Grow more efficient at driving business results
- **Rapid Delivery** – continuous integration and delivery
  - Increase the frequency and pace of releases
- **Reliability** – continuous integration and delivery, monitoring & logging
  - Ensure the quality of application updates and infrastructure changes
- **Scale** – automation, infrastructure as code
  - Operate and manage infrastructure and development processes at scale
- **Improved Collaboration**
  - Build more effective teams
- **Security** - automated compliance policies, fine-grained controls, and configuration management
  - Move quickly while retaining control and preserving compliance

Some DevOps principles

- DevOps is a comprehensive way **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 important to know its main principles and possibly **try to apply them whenever we write small or large applications**.

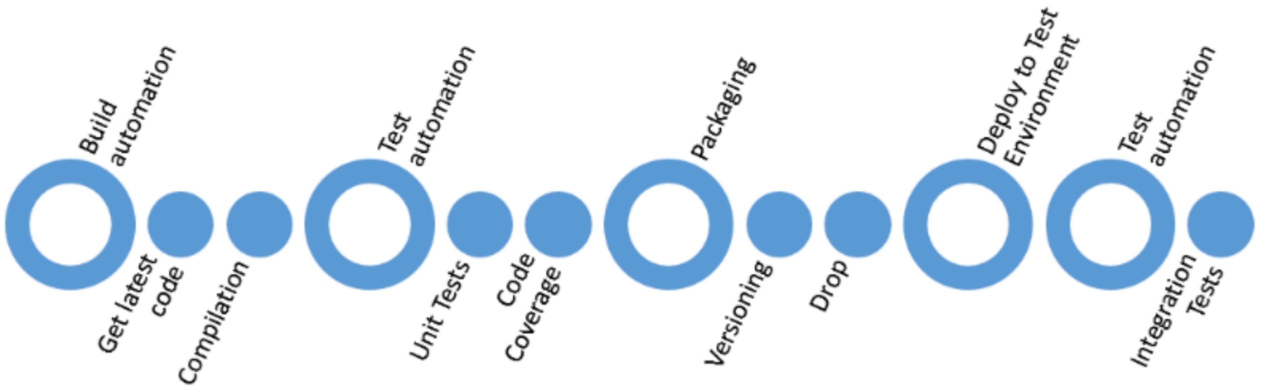
Continuous Integration

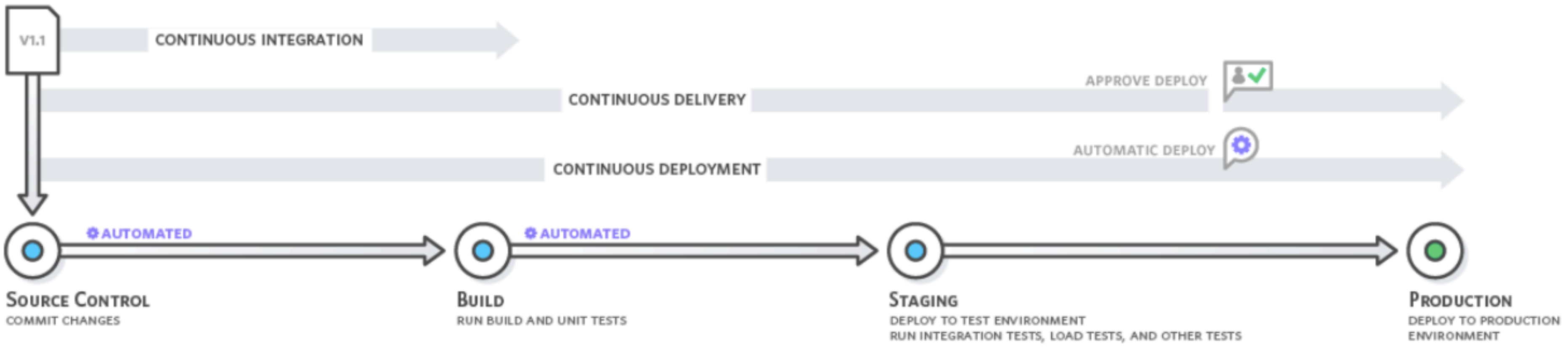
- **Continuous Integration** is a software development practice where developers **regularly merge** their code changes into a **central repository**, after which **automated builds** and **tests** are run
  - The result: **deployment packages** that can be used by Continuous Deployment (see later) for deployment to multiple environments.
  - A widely used tool for this: **Jenkins** (https://jenkins.io).



Source: https://jaxenter.com/how-to-move-from-ci-to-cd-with-jenkins-workflow-128135.html

Continuous Integration





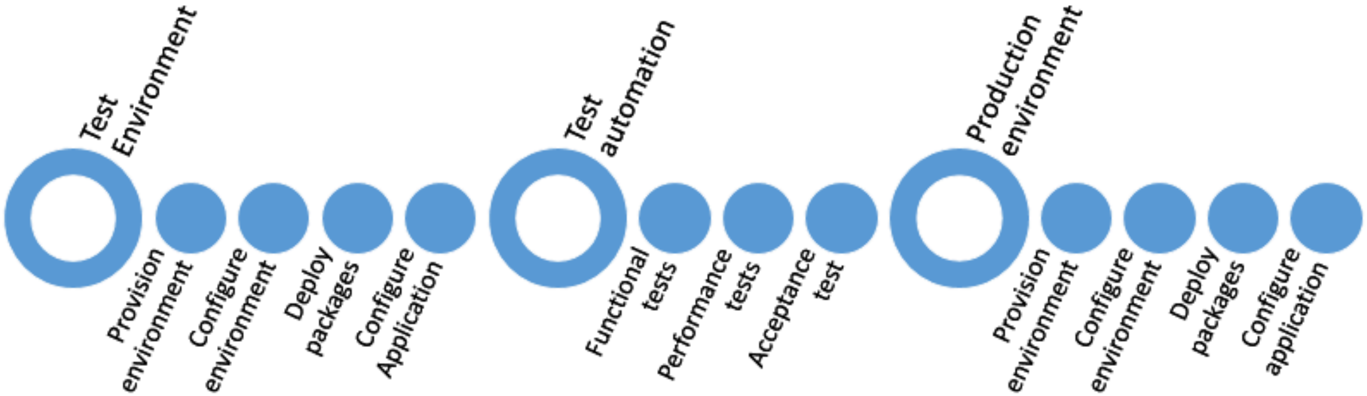
Why CI if I use Python?

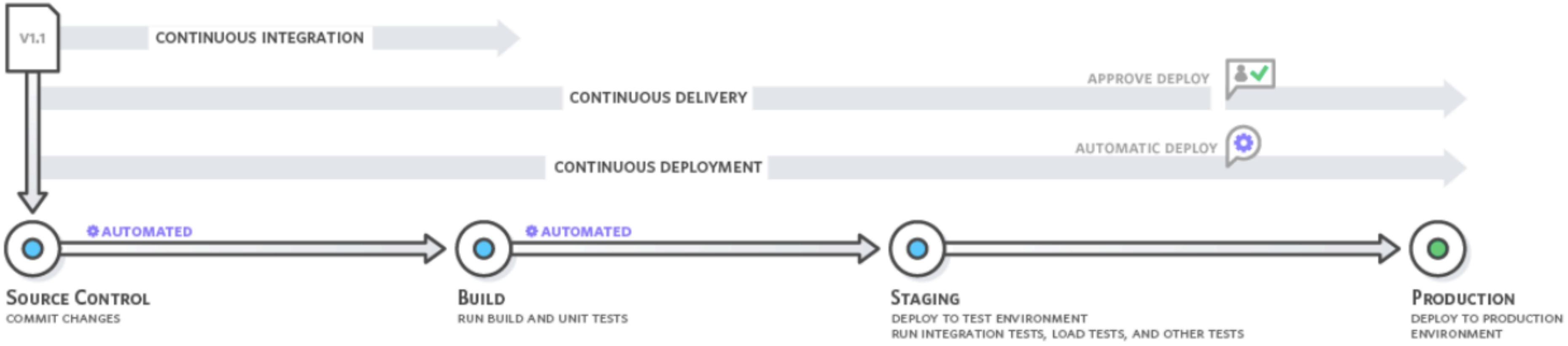
- Python does not need a “compilation step”. However, you can and should still use some Continuous Integration best practices in your projects, even if you only use Python. For example, you most likely want some Quality Assurance tests to be run *automatically*, such as:
  - **sloccount** to count the lines of code (i.e. non-blank, non-comment) in a program (not only in Python) – this seems simple, but it can give you an estimate about the complexity of a project. See https://dwheeler.com/sloccount/.
  - **Pylint** is “a Python static code analysis tool which looks for programming errors, helps enforcing a coding standard, sniffs for code smells and offers simple refactoring suggestions”. It is sometimes annoying but I would say it is a must use. See https://pypi.org/project/pylint/.
  - **Pytest** (https://docs.pytest.org/en/latest/index.html) and **Nose2** (https://github.com/nose-devs/nose2) make it easy to write tests for code coverage. Never underestimate the importance of writing tests in your programs!

Continuous Deployment

- **Continuous Deployment** refers to the capability to deploy applications and services to pre-production and production environments through automation.
  - Provision and configure an environment.
  - Deploy and configure an application on top of it.
  - After conducting multiple validations (functional performance) tests on a **pre-production** environment.
    - Provision and configure the **production** environment.
    - An application is deployed to production environments through **automation.**

Continuous Deployment

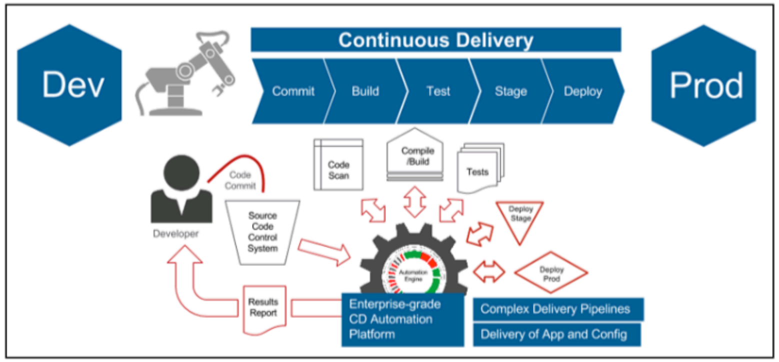




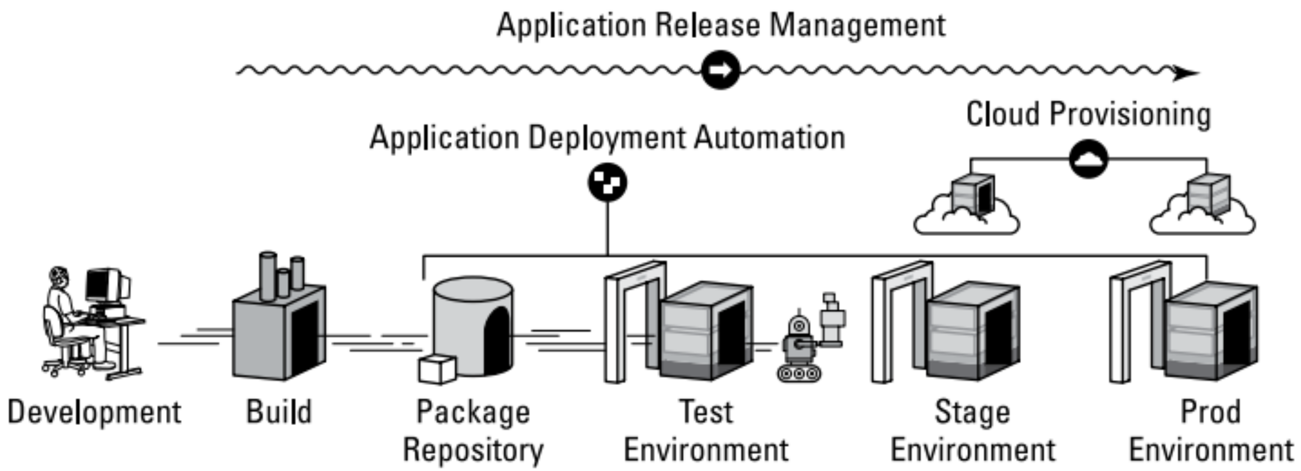
Continuous Delivery

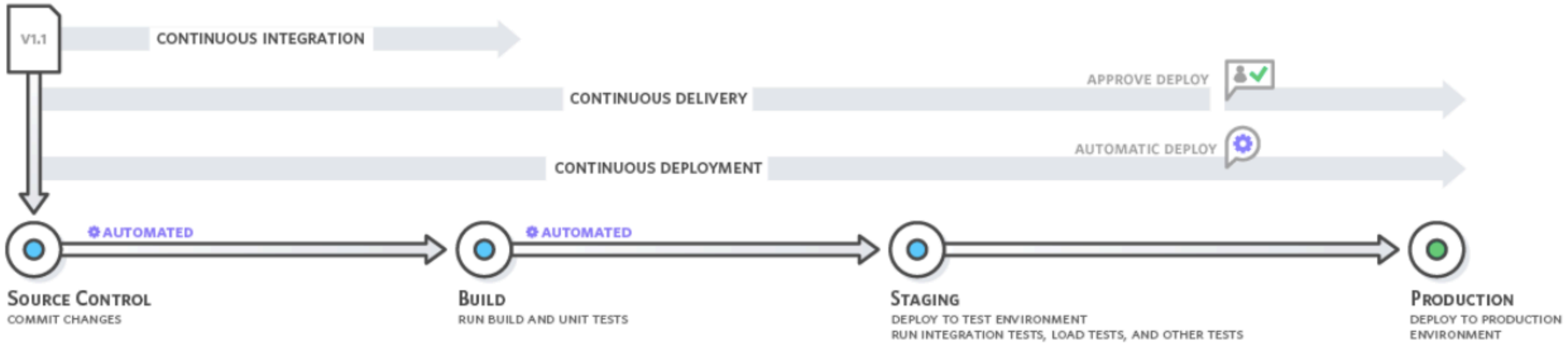
- **Continuous Delivery** is a software development practice where code changes are *automatically*:
  - Built,
  - Tested,
  - **Prepared** for a release to production.
- It expands upon continuous integration by deploying all code changes to a **testing** environment **and/or** a **production** environment after the build stage.
- When continuous delivery is **implemented properly**, developers will **always** have a **deployment-ready** build artifact that has passed through a standardized test process.

Source: https://jaxenter.com/how-to-move-from-ci-to-cd-with-jenkins-workflow-128135.html

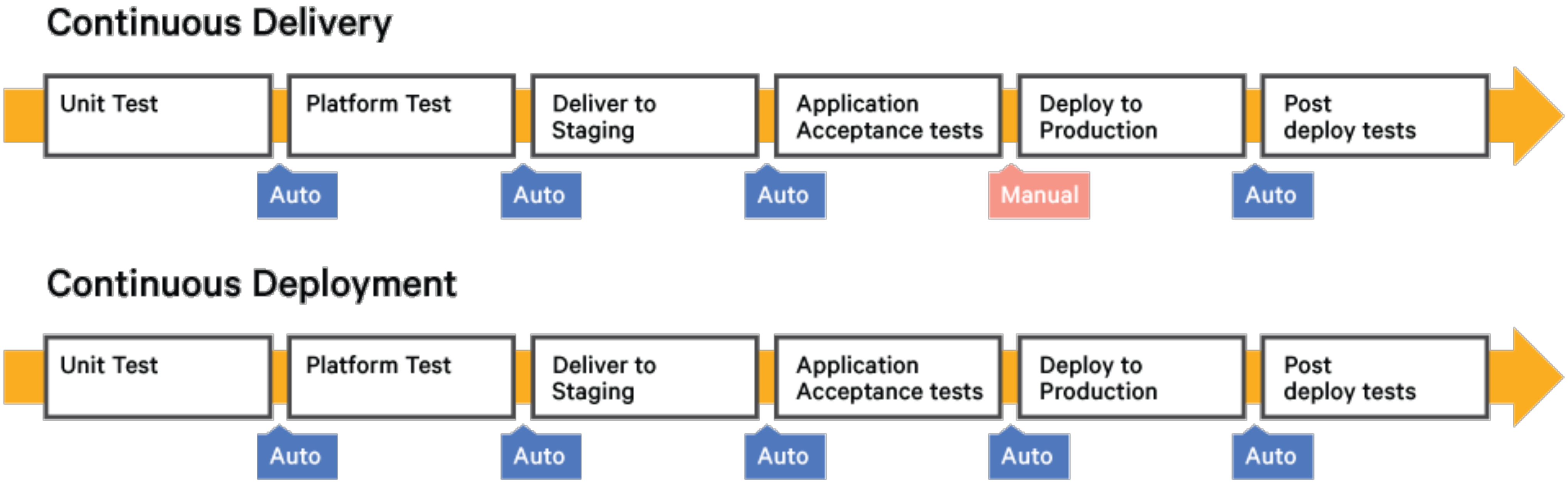


Continuous Delivery





Continuous Deployment vs. Continuous Delivery



Continuous Delivery

Unit Test → Platform Test → Deliver to Staging → Application Acceptance tests → Deploy to Production → Post deploy tests

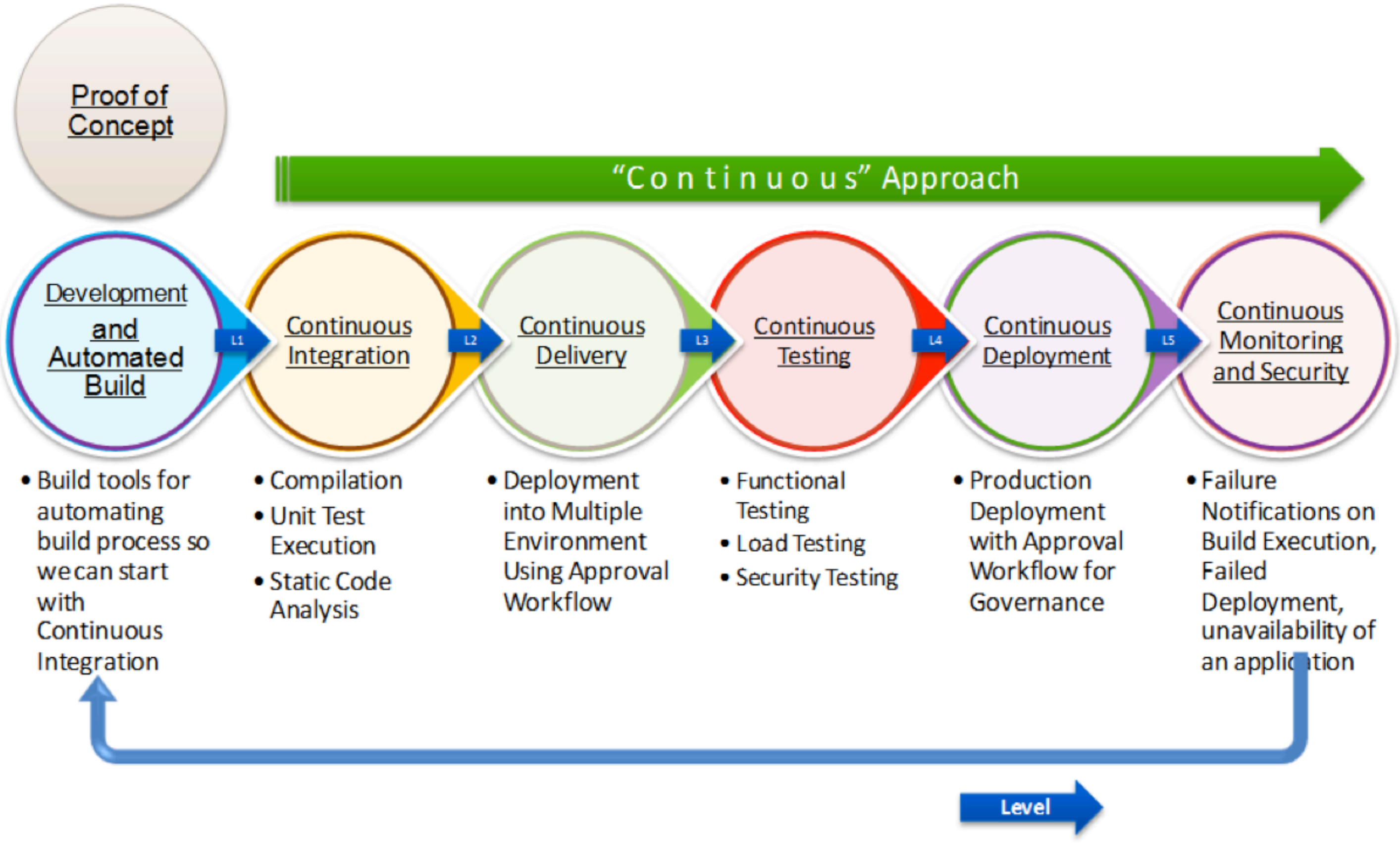
Auto | Auto | Auto | Manual | Auto

Continuous Deployment

Unit Test → Platform Test → Deliver to Staging → Application Acceptance tests → Deploy to Production → Post deploy tests

Auto | Auto | Auto | Auto | Auto

The “Continuous” mantra



Proof of Concept

“Continuous” Approach

Development and Automated Build

- Build tools for automating build process so we can start with Continuous Integration

L1

Continuous Integration

- Compilation
- Unit Test Execution
- Static Code Analysis

L2

Continuous Delivery

- Deployment into Multiple Environment Using Approval Workflow

L3

Continuous Testing

- Functional Testing
- Load Testing
- Security Testing

L4

Continuous Deployment

- Production Deployment with Approval Workflow for Governance

L5

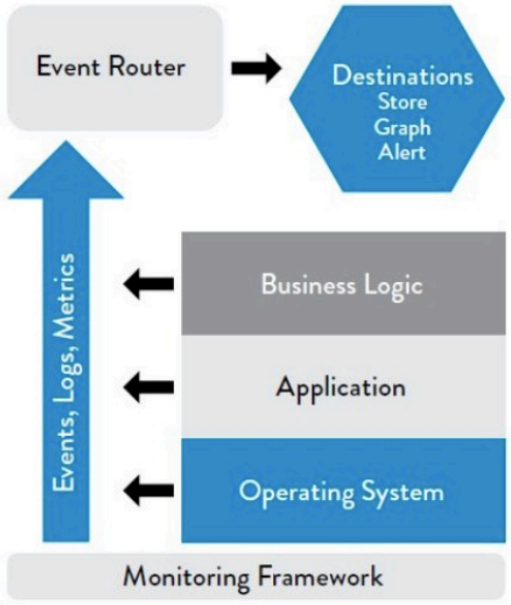
Continuous Monitoring and Security

- Failure Notifications on Build Execution, Failed Deployment, unavailability of an application

Level

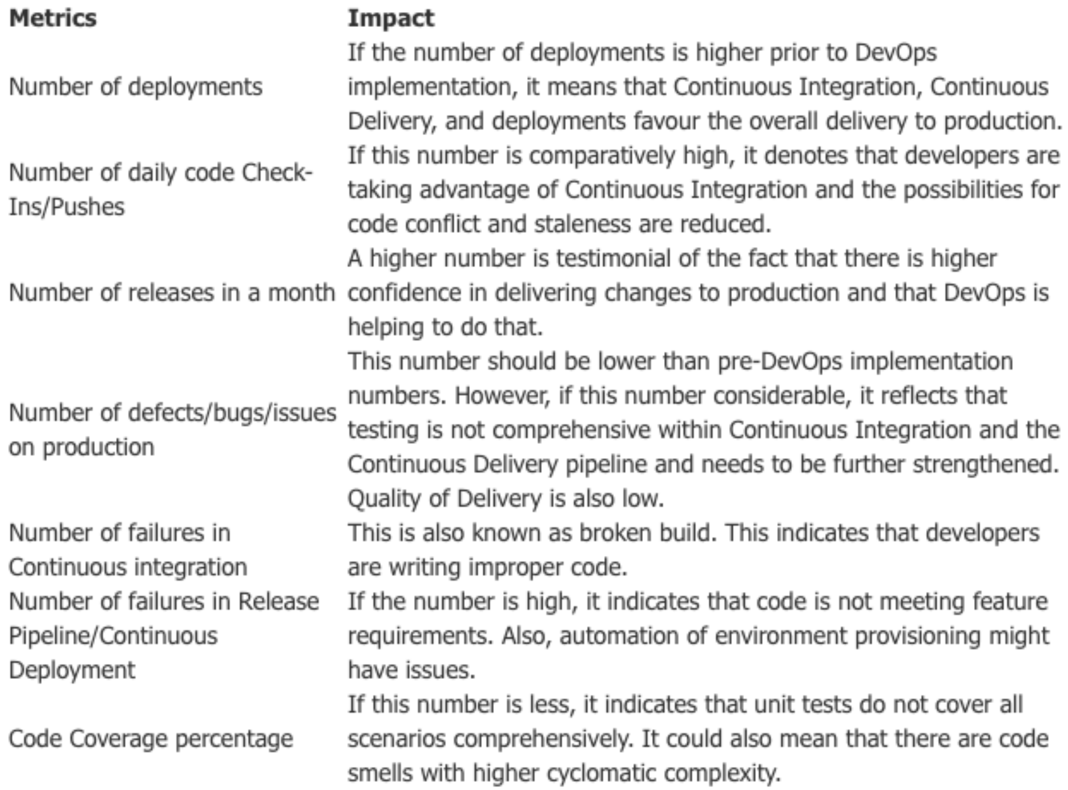
Continuous learning

- The benefits of DevOps will not last for long if a **continuous improvement and feedback** principle is not in place.
  - This means to have real-time feedback about the application's behavior.
- Applications should be built with:
  - Monitoring;
  - Auditing;
  - Telemetry in mind.



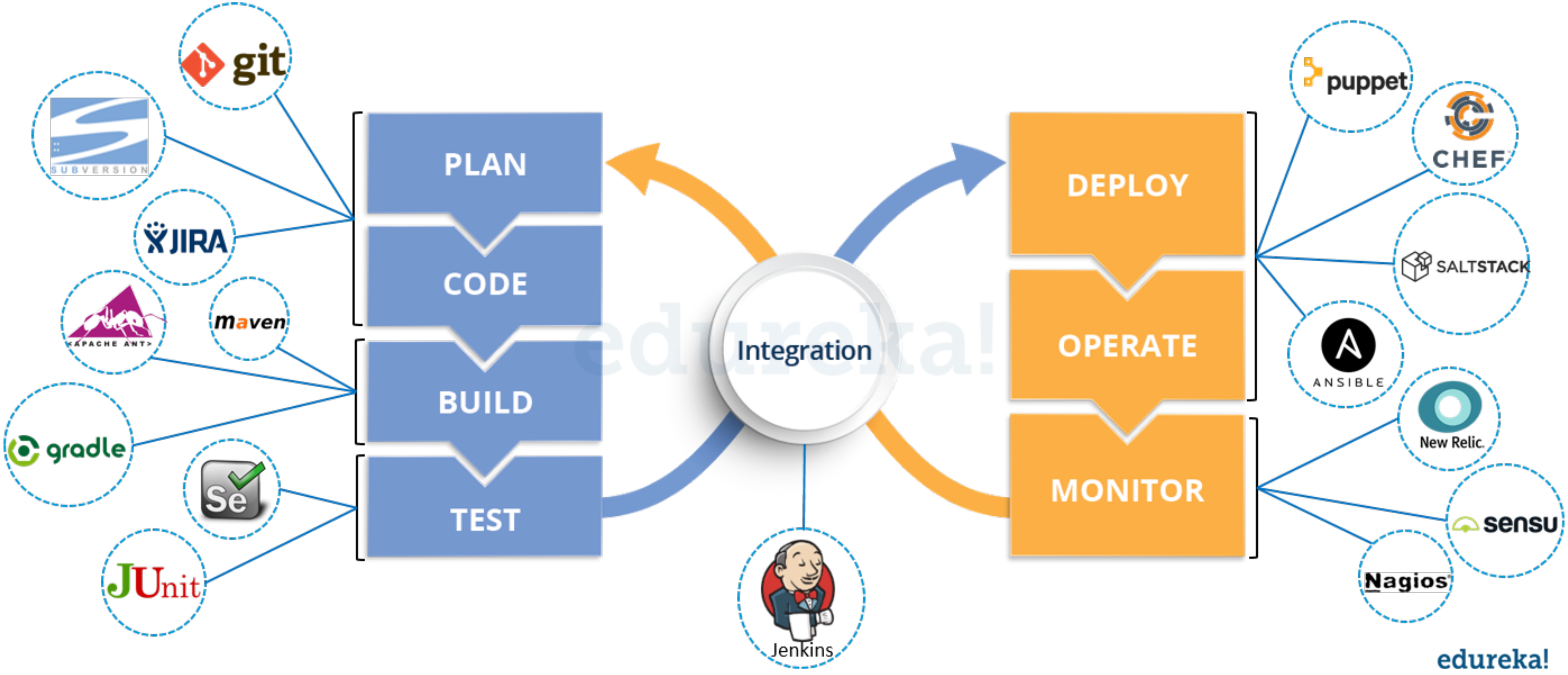
Continuous monitoring

- **Monitoring** starts in the development phase.
  - The same tools that monitor the production environment can be employed in development to spot performance problems *before* they hit production.
- Two kinds of monitoring are required for DevOps:
  - Server monitoring.
  - Application performance monitoring.
- **Measuring DevOps:**
  - **Monitoring, audit and collection of metrics** should be developed and deployed.
  - Regular baselining of data for effective comparison.
  - Metrics should be **captured over a period** and then compared with the baseline.



Metrics	**Impact**
Number of deployments	If the number of deployments is higher prior to DevOps implementation, it means that Continuous Integration, Continuous Delivery, and deployments favour the overall delivery to production.
Number of daily code Check-Ins/Pushes	If this number is comparatively high, it denotes that developers are taking advantage of Continuous Integration and the possibilities for code conflict and staleness are reduced.
Number of releases in a month	A higher number is testimonial of the fact that there is higher confidence in delivering changes to production and that DevOps is helping to do that.
Number of defects/bugs/issues on production	This number should be lower than pre-DevOps implementation numbers. However, if this number considerable, it reflects that testing is not comprehensive within Continuous Integration and the Continuous Delivery pipeline and needs to be further strengthened. Quality of Delivery is also low.
Number of failures in Continuous integration	This is also known as broken build. This indicates that developers are writing improper code.
Number of failures in Release Pipeline/Continuous Deployment	If the number is high, it indicates that code is not meeting feature requirements. Also, automation of environment provisioning might have issues.
Code Coverage percentage	If this number is less, it indicates that unit tests do not cover all scenarios comprehensively. It could also mean that there are code smells with higher cyclomatic complexity.

The DevOps tool chain



Where is my infrastructure?

- We have seen that, through a microservice architecture and some related processes and tools such as DevOps, we are able to **write applications** that are (or at least should be) scalable, reliable and maintainable.
- However, when it comes to **deploying these applications in the Cloud**, we naturally need to *find and configure the resources* that are needed by the application.
- For example, we need to provision the VMs where we can run our containers / microservices, exactly like we did when we created the first VMs on Cloud@CNAF.
- In other words, we need to **explicitly create our infrastructure**.

Container orchestration

- In a previous session, we explored how **containers** help us to easily create applications that are – as the name says – self-contained.
- On the other hand, we just saw that **microservice architectures** are based on the composition of many independent (but communicating) services.
- **Let's combine these two points**: containers can greatly help with the creation of a microservice architecture. Actually, through `docker-compose` we already learned how to create multiple containers linked together in **Application Stacks**.
- However, `docker-compose` is limited to the composition of containers within a single host. On the other hand, in general microservices are deployed across multiple hosts.
- We therefore need to explore how to effectively *orchestrate* many containers across distributed hosts. This is what we call **container orchestration**.

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 main features 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.

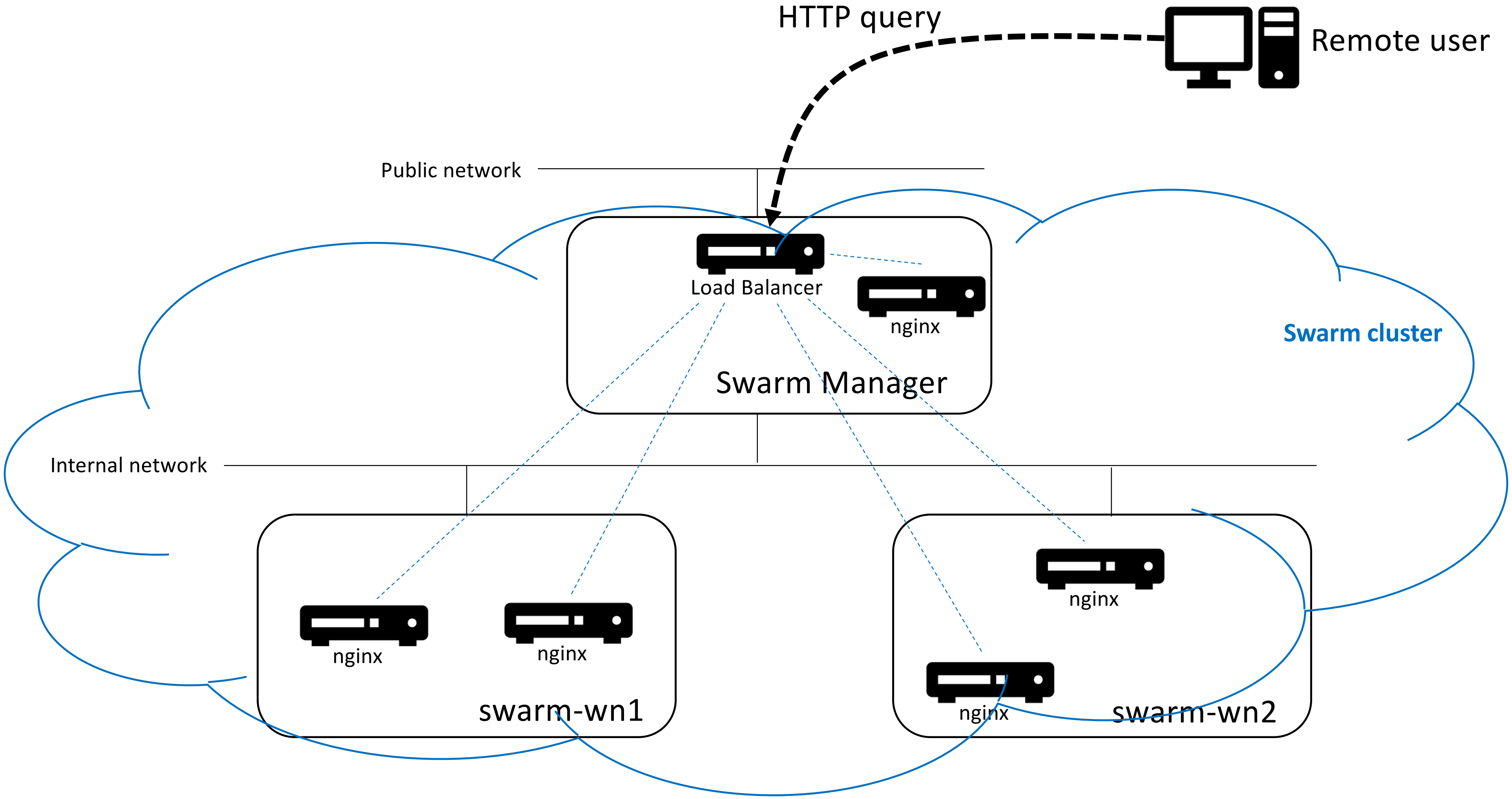
Hands-on with Docker Swarm

- 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

- 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

- 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

- 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;
}
```

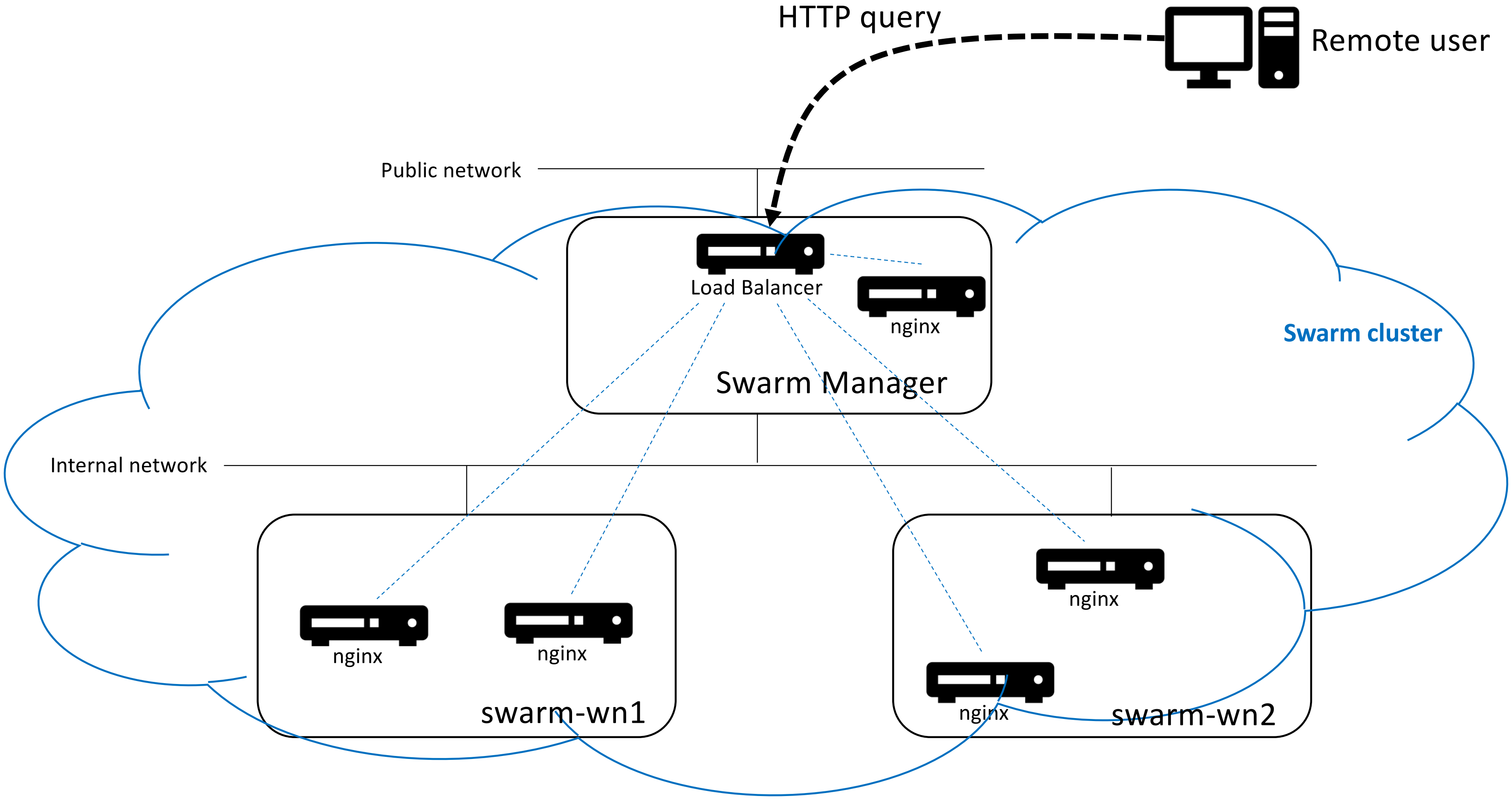
A few notes

- 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>
```

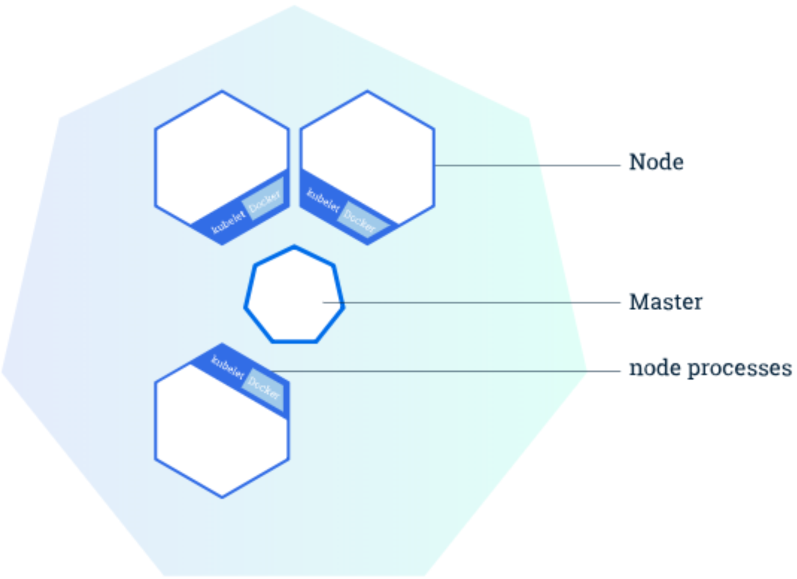
- An interesting point is to combine Docker Swarm with custom Docker images or with Docker Compose. This is left as an exercise.

Docker Swarm: our architecture



Kubernetes

https://kubernetes.io

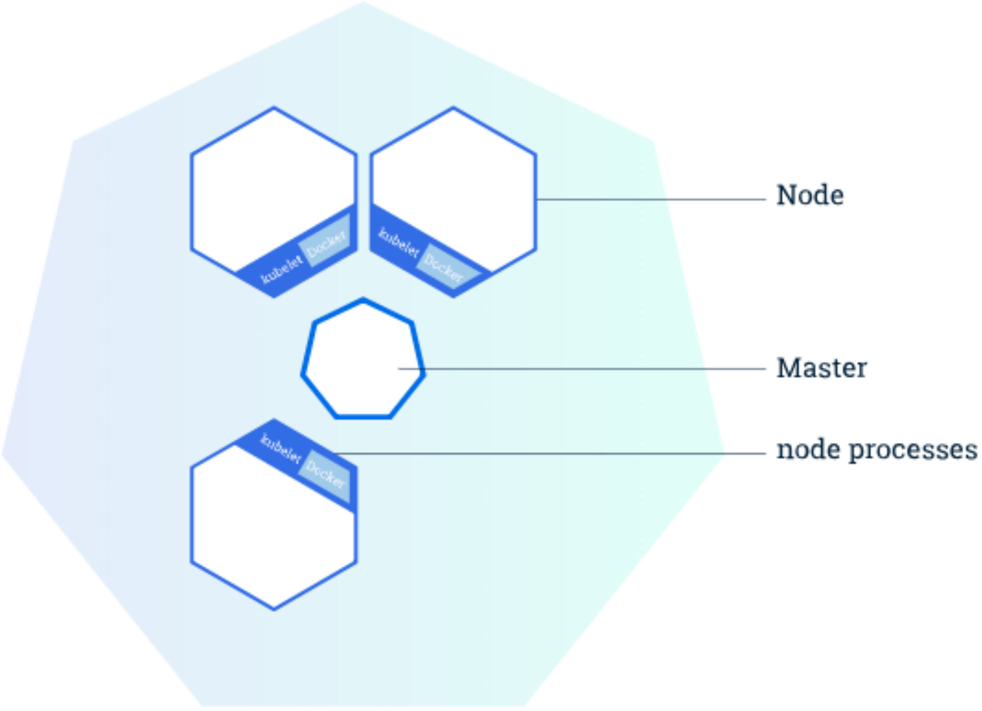


Kubernetes cluster

- Kubernetes is an open-source platform that coordinates a highly available cluster of computers that are connected to work as a single unit. It is backed by Google and RedHat.
- Applications need to be **containerized**.
- Kubernetes automates the distribution and scheduling of application containers across a cluster in a fairly efficient way.
- A Kubernetes cluster can be deployed on either physical or virtual machines.

Kubernetes cluster resources

- A Kubernetes cluster consists of two types of resources:
  - The **Master** coordinates the cluster
  - **Nodes** are the workers that run applications
- **The Master is responsible for managing the cluster**
  - coordinates all activities in your cluster, such as scheduling applications, maintaining applications' desired state, scaling applications, and rolling out new updates.
- **A node** is a VM or a physical computer that **serves as a worker machine in a Kubernetes cluster**



Kubernetes cluster

Kubernetes Master/Nodes processes

- The **Kubernetes Master** is a collection of three processes that run on a single node in your cluster, which is designated as the *master node*. These processes are:
  - **kube-apiserver**
  - **kube-controller-manager**
  - **kube-scheduler**
- Each individual Node in your cluster runs two processes:
  - **kubelet**, which communicates with the Kubernetes Master.
  - **kube-proxy**, a network proxy which reflects Kubernetes networking services on each node.
- Moreover, each Node runs a container runtime (like Docker) responsible for pulling the container image from a registry, unpacking the container, and running the application.
- A Kubernetes cluster that handles production traffic should have a minimum of **three nodes**.

Kubernetes Objects

- Kubernetes contains a number of abstractions that represent the state of your system: deployed containerized applications and workloads, their associated network and disk resources, and other information about what your cluster is doing.
- These abstractions are represented by objects in the Kubernetes API.
- The basic Kubernetes objects include:
  - Volume
  - Namespace
  - Deployment
  - Pod
  - Service

Kubernetes volume

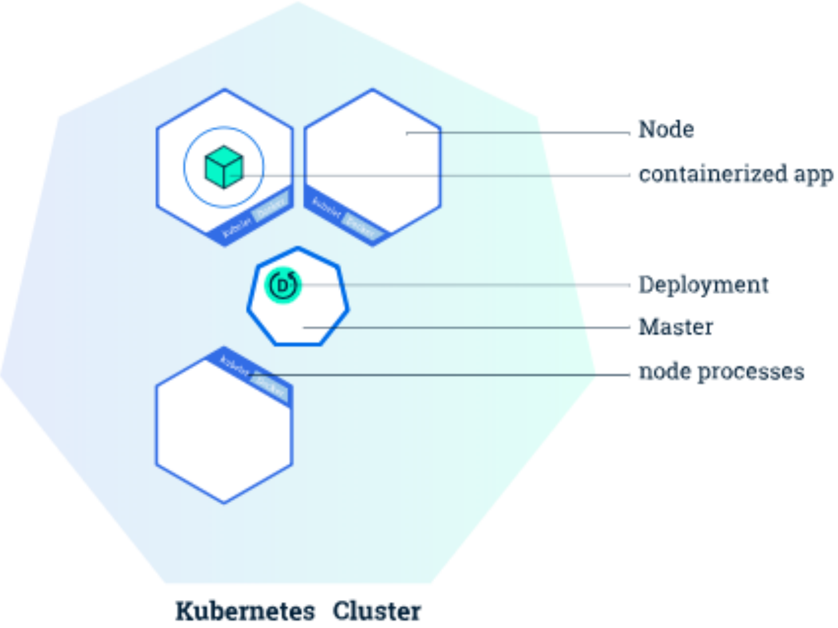
- As we have already seen, on-disk files within a container are ephemeral. This presents some problems for non-trivial applications when running in containers.
  - When a Container crashes, kubelet will restart it, but the internal container files will be lost - the Container starts with a clean state.
  - When running Containers together in a Pod it is often necessary to share files between those Containers.
- The Kubernetes Volume abstraction solves both of these problems.

Kubernetes Namespaces

- Kubernetes supports multiple virtual clusters backed by the same physical cluster.
- These virtual clusters are called namespaces.

Kubernetes Deployment

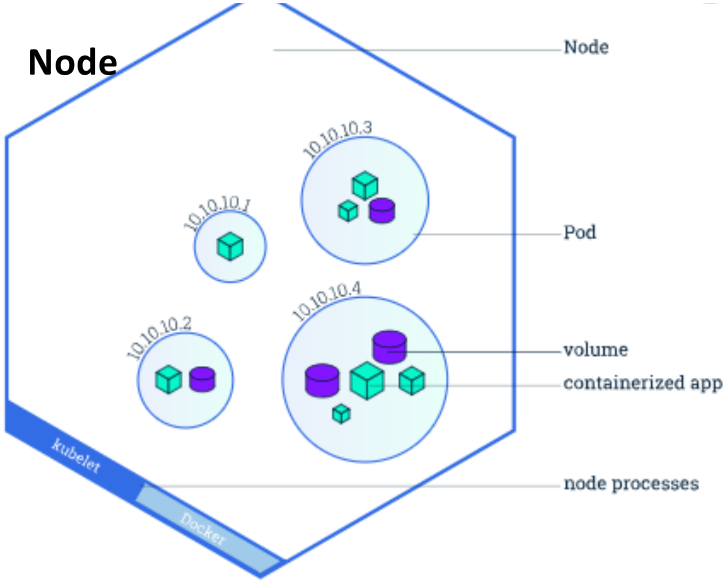
- Once you have a running Kubernetes cluster, you can deploy your containerized applications on top of it. To do so, you create a Kubernetes **Deployment** configuration.
- The Deployment tells Kubernetes how to create and update instances of your application. Once you've created a Deployment, the Kubernetes master schedules application instances onto individual Nodes in the cluster.
- Once the application instances are created, a Kubernetes Deployment Controller continuously monitors those instances. If the Node hosting an instance goes down or is deleted, the Deployment controller replaces it. **This provides a self-healing mechanism to address machine failure or maintenance.**
- In a pre-orchestration world, installation scripts would often be used to start applications, but they did not allow recovery from machine failure. By both creating your application instances and keeping them running across Nodes, Kubernetes Deployments provide a fundamentally different approach to applications.



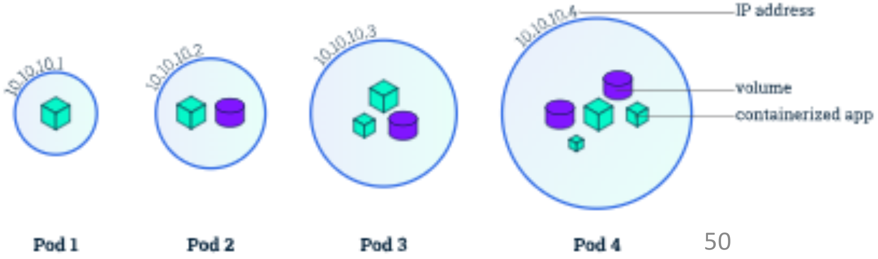
Kubernetes Pod

- A **Pod** is the basic building block of Kubernetes. It represents a running process on your cluster.
- A Pod encapsulates an application container, storage resources, a unique network IP, and options that govern how the container(s) should run.
- **Pods that run a single container**. The “one-container-per-Pod” model is the most common Kubernetes use case; in this case, you can think of a Pod as a wrapper around a single container, and Kubernetes manages the Pods rather than the containers directly.
- **Pods that run multiple containers that need to work together**. A Pod might encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. The Pod wraps these containers and storage resources together as a single manageable entity.

Node

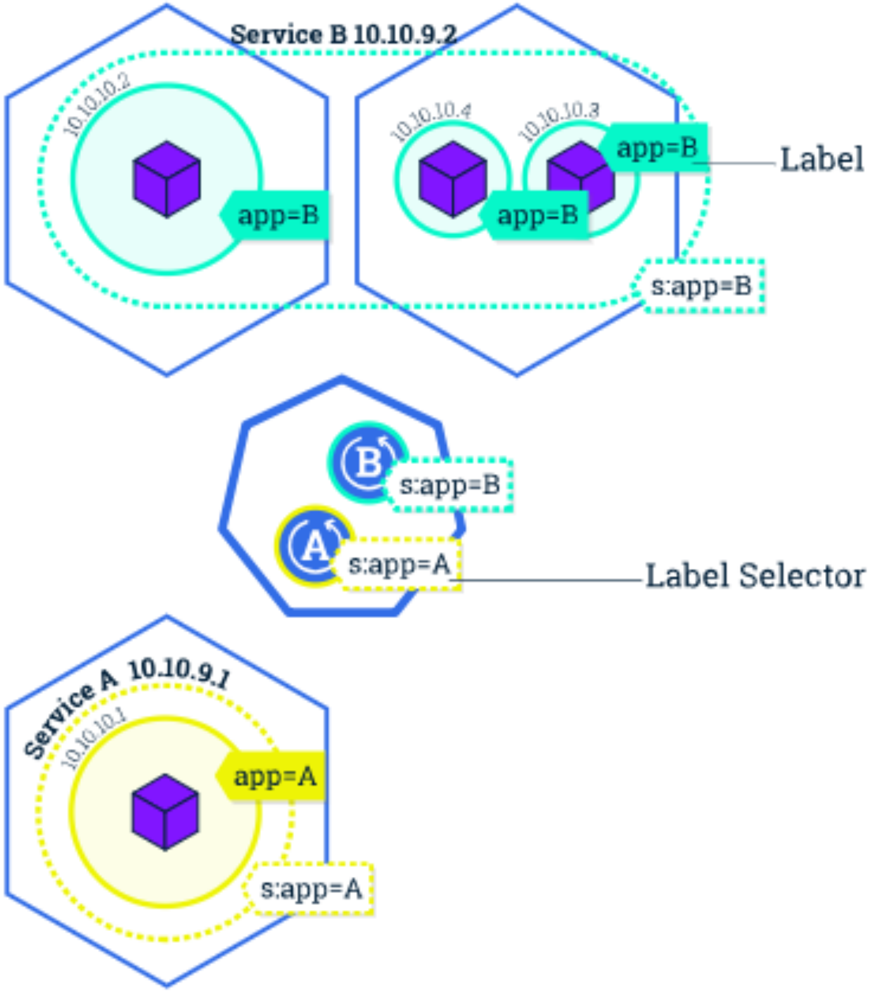


Pod

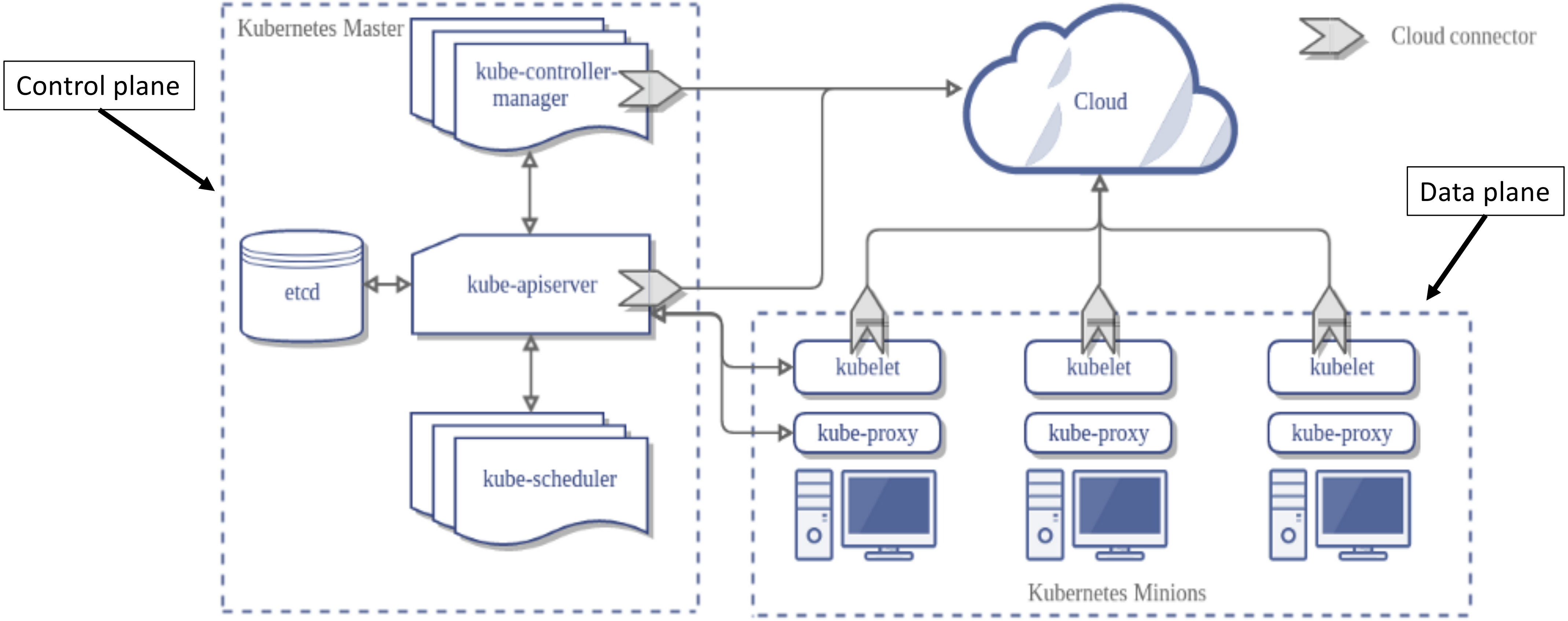


Kubernetes Services

- A Kubernetes Service is an abstraction which defines a logical set of Pods and a policy by which to access it.
- Although each Pod has a unique IP address, those IPs are not exposed outside the cluster without a Service. Services allow your applications to receive traffic.
- Services match a set of Pods using labels and selectors, a grouping primitive that allows logical operation on objects in Kubernetes. Labels are key/value pairs attached to objects and can be used in any number of ways:
  - Designate objects for development, test, and production
  - Embed version tags
  - Classify an object using tags



Kubernetes architecture



Kubernetes hands-on

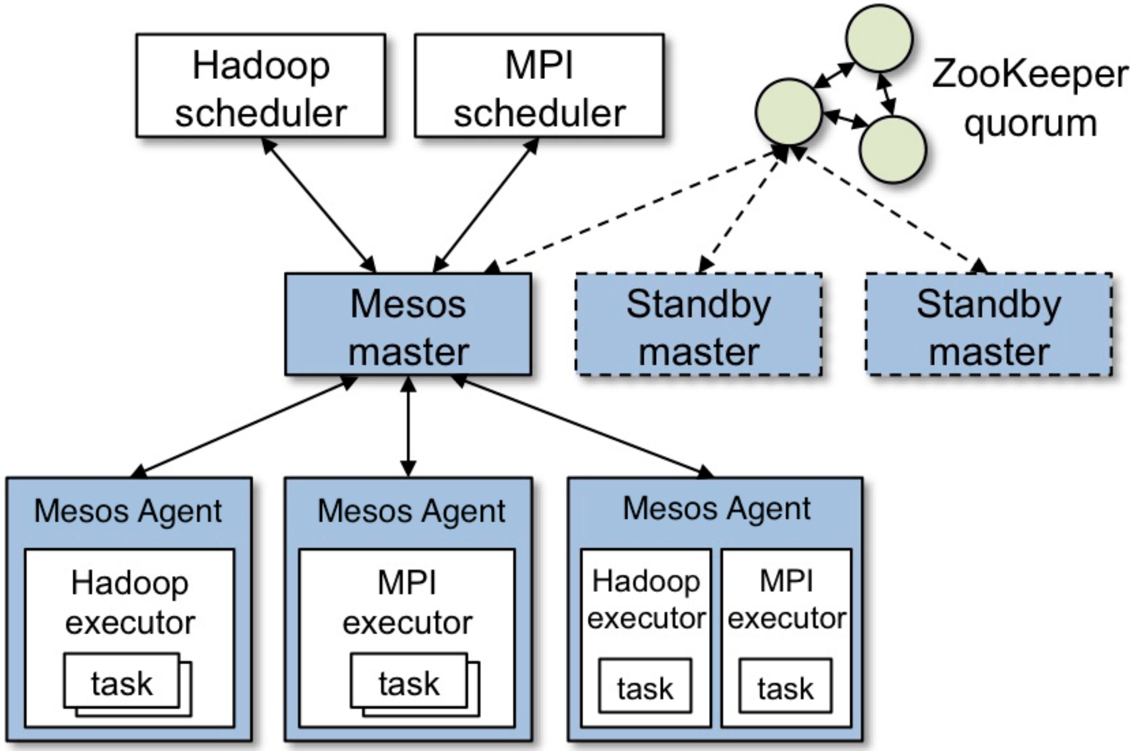
- **https://baltig.infn.it/corsi_formazione_ccr/corso_bd_2019/tree/master/docker**
  - Leveraging a K8S cluster already created on https://rancher.cloud.cnaf.infn.it/
- In addition you can use **Minikube -** to deploy a Kubernetes cluster on your own on your laptop
- Minikube is a **lightweight Kubernetes implementation** that creates a VM on your local machine and deploys a simple cluster containing only one node. See https://github.com/kubernetes/minikube for details.

Kubernetes as a Service

- Deploying and managing a Kubernetes cluster is generally not trivial (that’s why Minikube was introduced), since it requires effort and several skills.
- It would be nice to automatize this part as well and focus just on deploying our containers on a Kubernetes cluster that somebody else instantiates for us.
- Many Cloud providers give us just that: a **Kubernetes as a Service**.
  - Amazon provides what they call the “**Elastic Container Service for Kubernetes**”, or **EKS** for short. Other providers have similar offerings.
  - We have seen that Kubernetes cluster consist of a control plane, where the masters are running, and of a data plane, where we have our worker nodes and containers. **EKS provides a managed control plane**, deployed in a fully highly available setup. Since this a service managed by AWS, we don’t need to care about updates to the Kubernetes software itself.

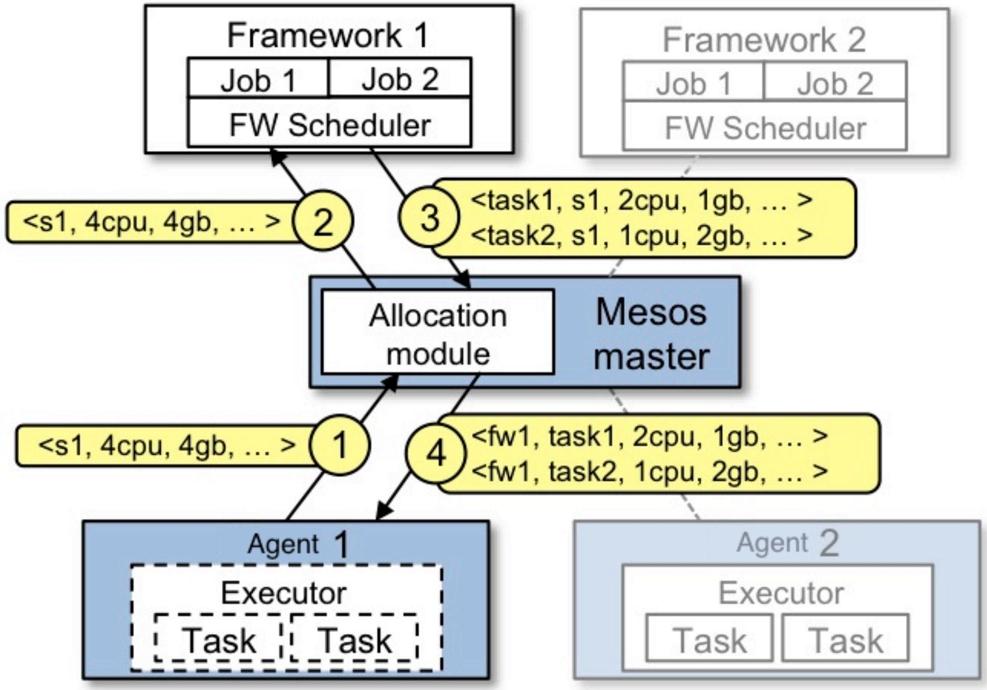
Apache Mesos

- **Apache Mesos** (http://mesos.apache.org) is a software layer over which diverse *frameworks* can run.
- In some way, Mesos is **the opposite of virtualization**: while virtualization divides a single physical resource into many virtual ones, Mesos allows you to share a large cluster of machines between different frameworks.



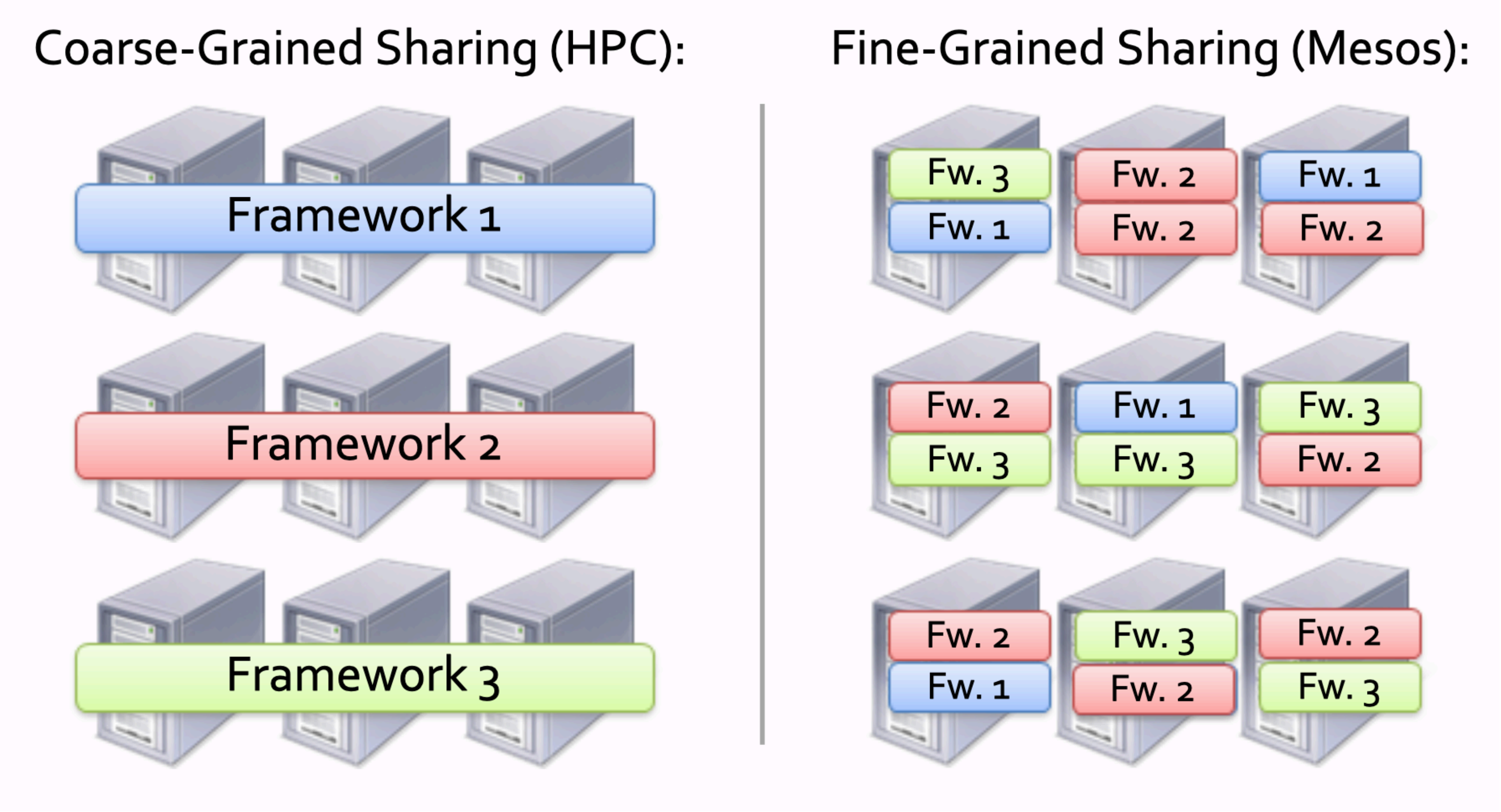
How Mesos works

1. Agent 1 reports to the master that it has 4 CPUs and 4 GB of memory free. The master then invokes the allocation policy module, which tells it that framework 1 should be offered all available resources.
2. The master sends a resource offer describing what is available on agent 1 to framework 1.
3. The framework's scheduler replies to the master with information about two tasks to run on the agent, using <2 CPUs, 1 GB RAM> for the first task, and <1 CPUs, 2 GB RAM> for the second task.
4. Finally, the master sends the tasks to the agent, which allocates appropriate resources to the framework's executor, which in turn launches the two tasks (depicted with dotted-line borders in the figure). Because 1 CPU and 1 GB of RAM are still unallocated, the allocation module may now offer them to framework 2.



- From http://mesos.apache.org/documentation/latest/architecture/

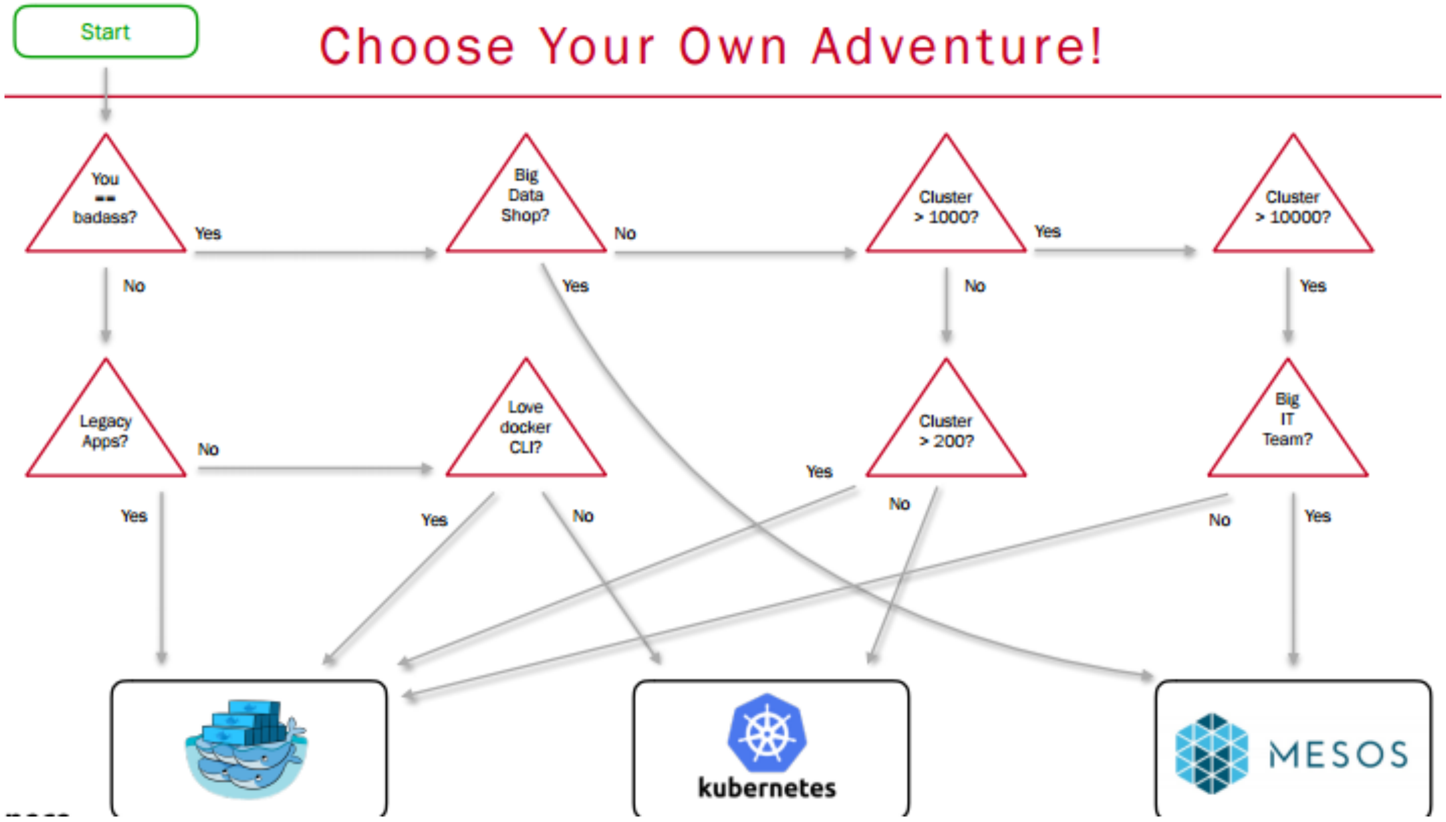
Mesos fine-grained sharing



Docker Swarm, Kubernetes, Mesos: which one to choose? (1)

- We have seen (with different degrees of in-depth analysis) the three current major solutions for **container or resource orchestration**, which is a topic that sooner or later normally comes up with anything but the simplest big data problems.
- Some general considerations on when to use what:
  - **Docker Swarm** for smaller projects and for testing purposes. Easy to use if you are already familiar with Docker.
  - For larger, enterprise-like solutions, **Kubernetes**. It’s also “the Google way of doing it”. But mind the rather steep learning curve.
  - **Mesos** for very large clusters and for workflow-based solutions. It can be fairly complex, so it might need a sizeable support team.

Docker Swarm, Kubernetes, Mesos: which one to choose? (2)



- From https://www.bogotobogo.com/DevOps/DevOps-Docker-Swarm-vs-Kubernetes-vs-Apache-Mesos.php

Infrastructure as Code (1)

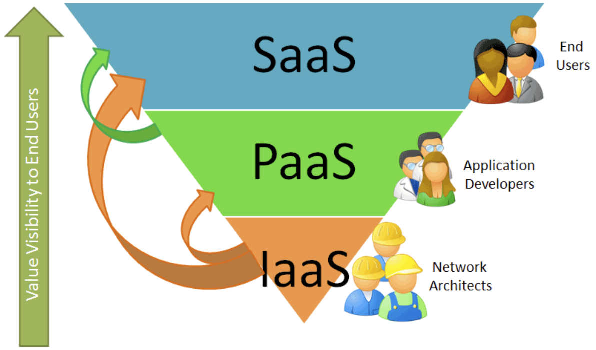
- With the idea of **Infrastructure as Code (IaC)**, instead of manually creating the infrastructure we need for our applications (e.g. virtual machines, disk volumes, installations, configurations), we **define what we want** through machine-readable definition files.
  - IaC is based on the realization 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)

- 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).

Serverless technologies

- With **serverless technologies**, we perform another step toward automating and facilitating the use of Cloud resources.
  - Remember that - **what eventually matters are the applications, not the infrastructure.**



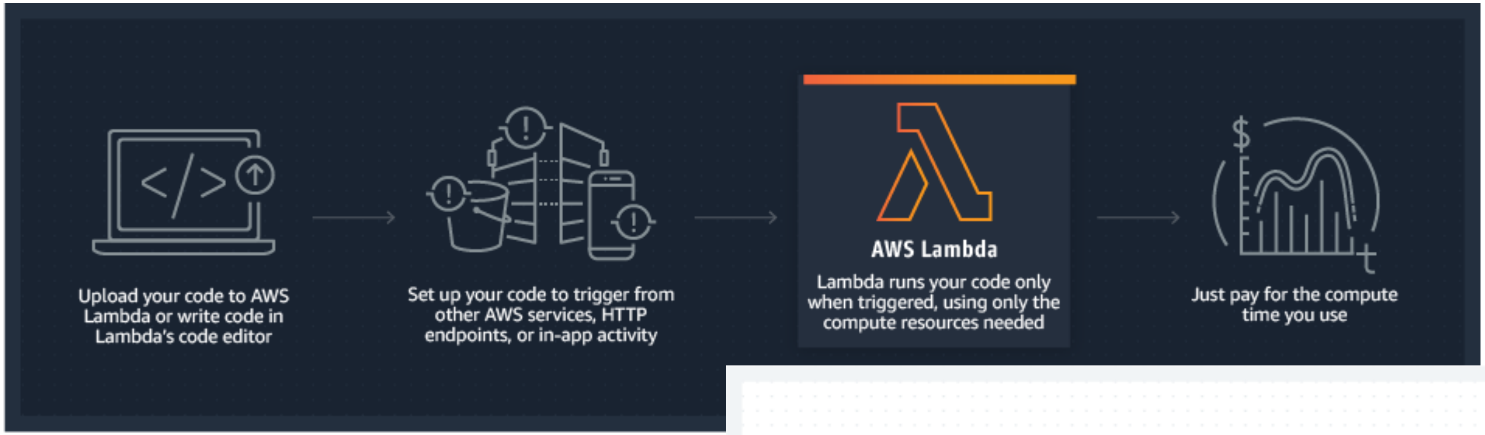
- Recall what happens with traditional Cloud applications, of which we have already seen several examples:
  - We need to **provision and manage the resources** (e.g. VM1, VM2, the disks, the S3 buckets, etc.) for our applications.
  - We are **charged if we keep the resources up**, even if they are doing nothing.
  - We are responsible to **apply all the updates and security patches** to our servers.

What is serverless, or FaaS

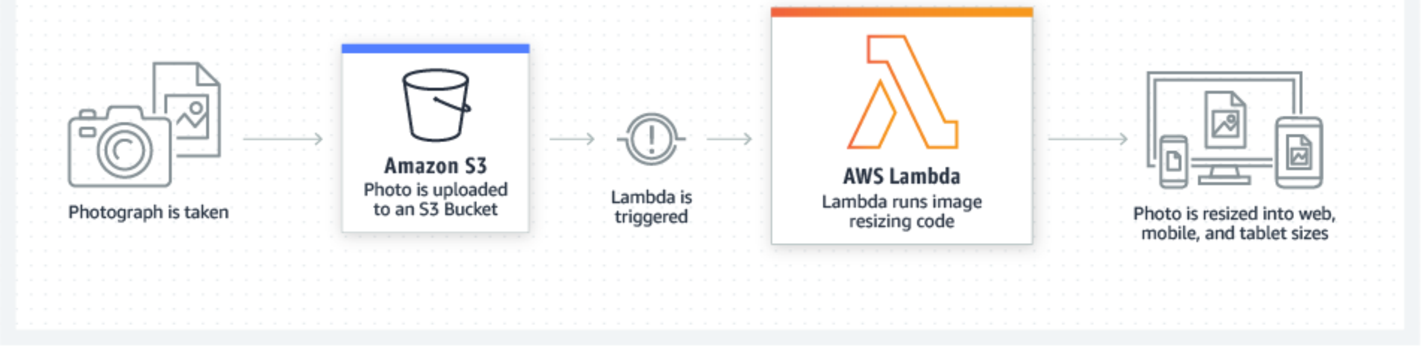
- With **serverless**, a Cloud provider is responsible for executing a piece of code (written by you) by **dynamically allocating the resources needed by the code**.
- You are only charged for the resources used to run the code and only when the code runs.
- This code is typically a function. Thus, serverless computing is also called **Functions as a Services, or FaaS**.
- The running of these functions can be **triggered** depending on some **conditions**, such as for example database events, queueing services, file uploads, scheduled events, various alerts, etc.
- Your applications should therefore be **structured around a set of stateless functions** → this is consistent with the idea of **microservices** we have already seen.

AWS Lambda

- In the Amazon world, serverless computing is called **AWS Lambda**.
- This is how it works (picture from Amazon):

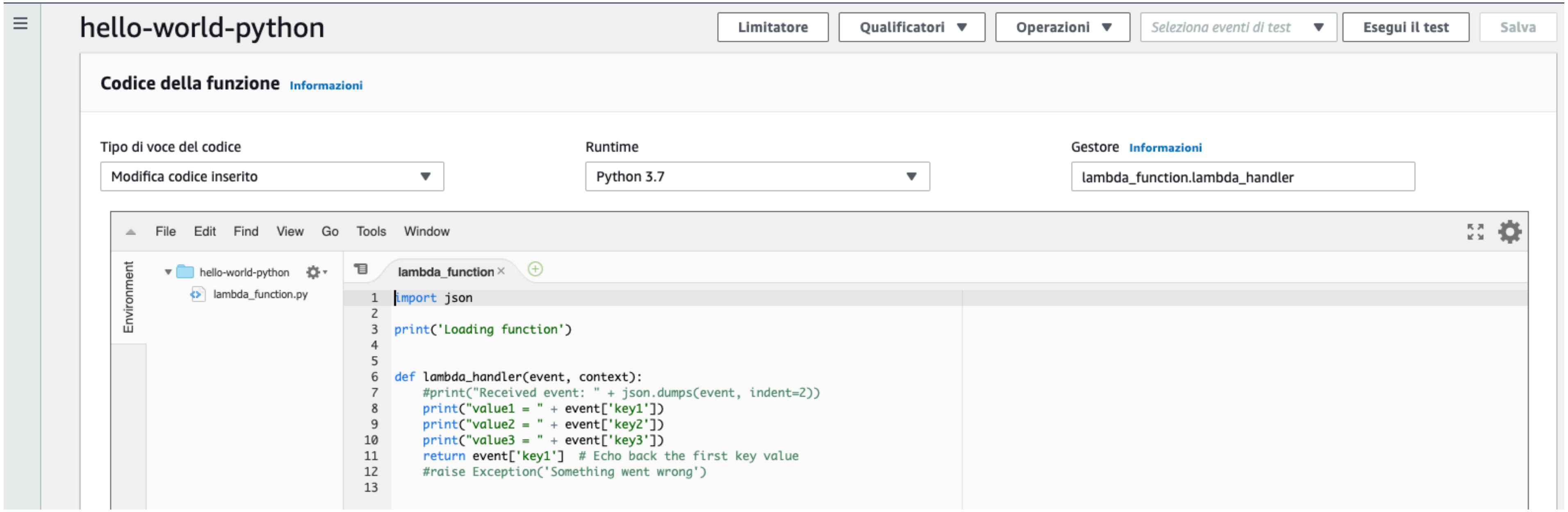

Upload your code to AWS Lambda or write code in Lambda's code editor
Set up your code to trigger from other AWS services, HTTP endpoints, or in-app activity
AWS Lambda
Lambda runs your code only when triggered, using only the compute resources needed
$
t
Just pay for the compute time you use

A simple AWS Lambda example:


Photograph is taken
Amazon S3
Photo is uploaded to an S3 Bucket
Lambda is triggered
AWS Lambda
Lambda runs image resizing code
Photo is resized into web, mobile, and tablet sizes

How an AWS Lambda looks like

- We won’t do a direct hands-on with Lambda.
- However, this how a sample Python function would look like in the AWS Lambda Console:



Tipo di voce del codice: Modifica codice inserito

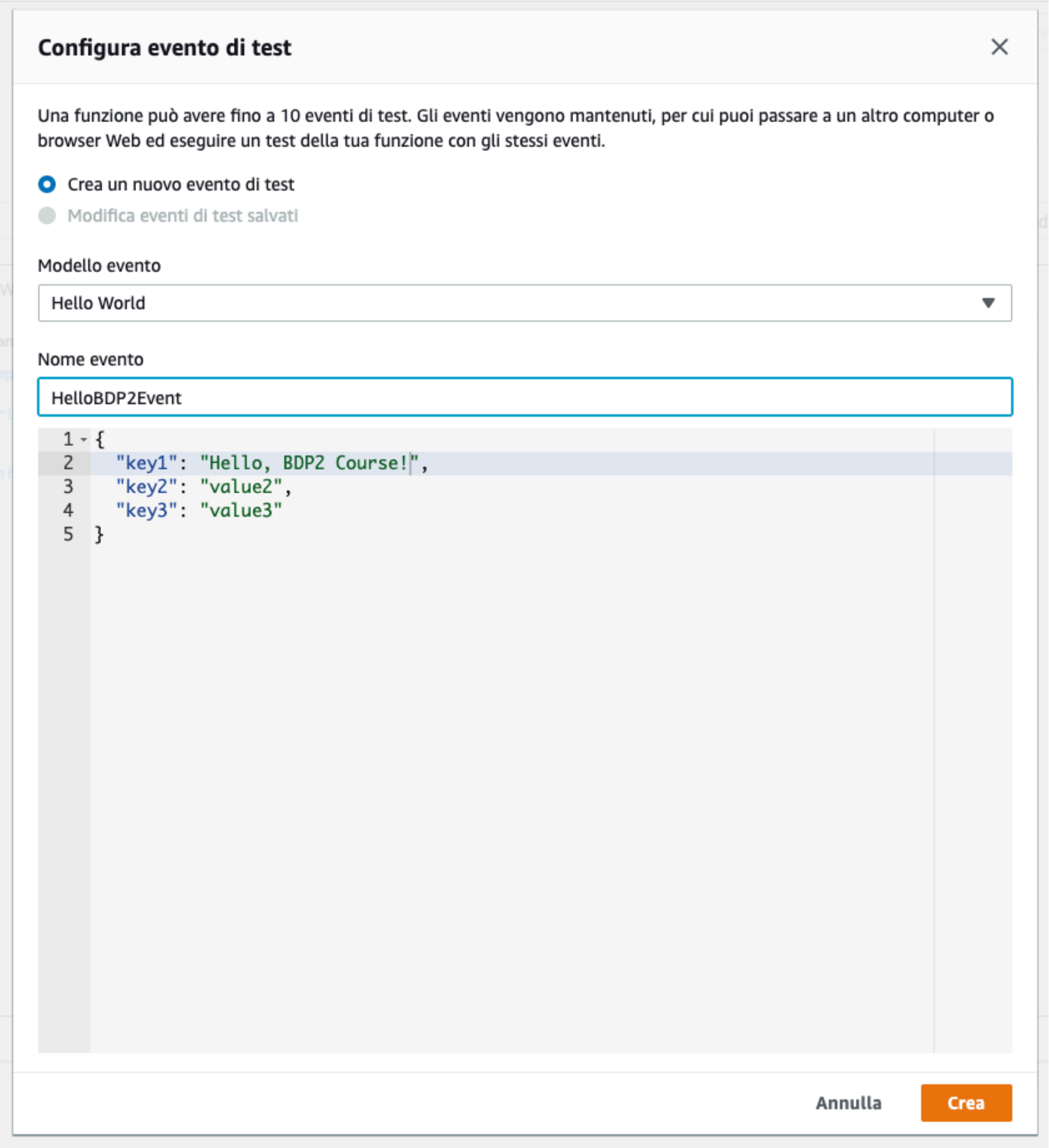
Runtime: Python 3.7

Gestore: lambda_function.lambda_handler

```
import json

print('Loading function')


def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))
    print("value1 = " + event['key1'])
    print("value2 = " + event['key2'])
    print("value3 = " + event['key3'])
    return event['key1']  # Echo back the first key value
    #raise Exception('Something went wrong')
```

Testing an AWS Lambda function

- My Python function just replies to some events, printing out their content. I am testing it creating a *test event* with some dummy values, as shown in the picture on the right.



Configura evento di test

Una funzione può avere fino a 10 eventi di test. Gli eventi vengono mantenuti, per cui puoi passare a un altro computer o browser Web ed eseguire un test della tua funzione con gli stessi eventi.

● Crea un nuovo evento di test

○ Modifica eventi di test salvati

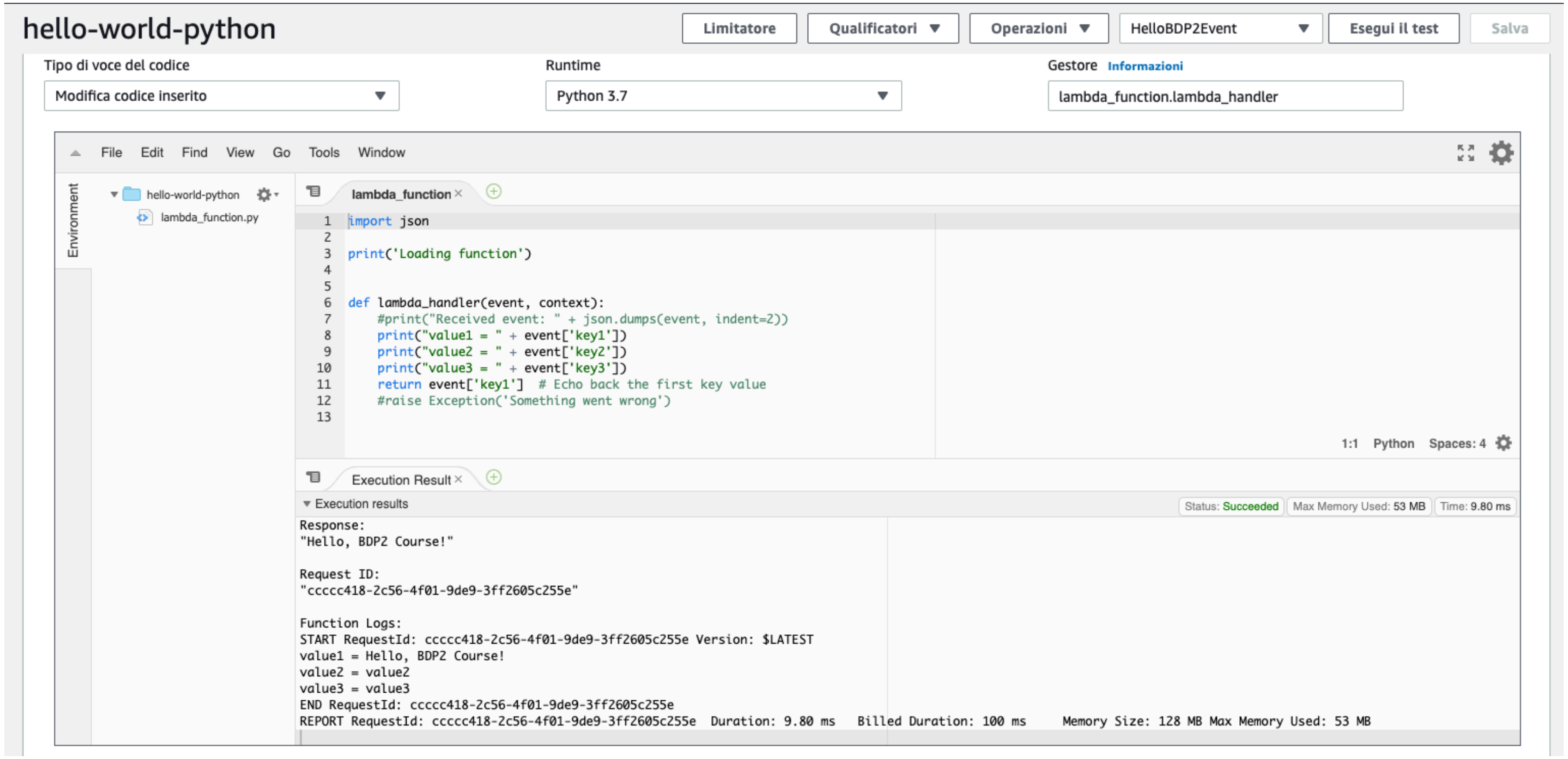
Modello evento: Hello World

Nome evento: HelloBDP2Event

```
{
  "key1": "Hello, BDP2 Course!",
  "key2": "value2",
  "key3": "value3"
}
```

Test results

- When I now actually run the test, this is what is shown:



Tipo di voce del codice: Modifica codice inserito

Runtime: Python 3.7

Gestore: lambda_function.lambda_handler

```python
import json

print('Loading function')


def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))
    print("value1 = " + event['key1'])
    print("value2 = " + event['key2'])
    print("value3 = " + event['key3'])
    return event['key1']  # Echo back the first key value
    #raise Exception('Something went wrong')

```

1:1 Python Spaces: 4

Execution results

Status: Succeeded | Max Memory Used: 53 MB | Time: 9.80 ms

```
Response:
"Hello, BDP2 Course!"

Request ID:
"ccccc418-2c56-4f01-9de9-3ff2605c255e"

Function Logs:
START RequestId: ccccc418-2c56-4f01-9de9-3ff2605c255e Version: $LATEST
value1 = Hello, BDP2 Course!
value2 = value2
value3 = value3
END RequestId: ccccc418-2c56-4f01-9de9-3ff2605c255e
REPORT RequestId: ccccc418-2c56-4f01-9de9-3ff2605c255e  Duration: 9.80 ms   Billed Duration: 100 ms     Memory Size: 128 MB Max Memory Used: 53 MB
```

Why is this useful?

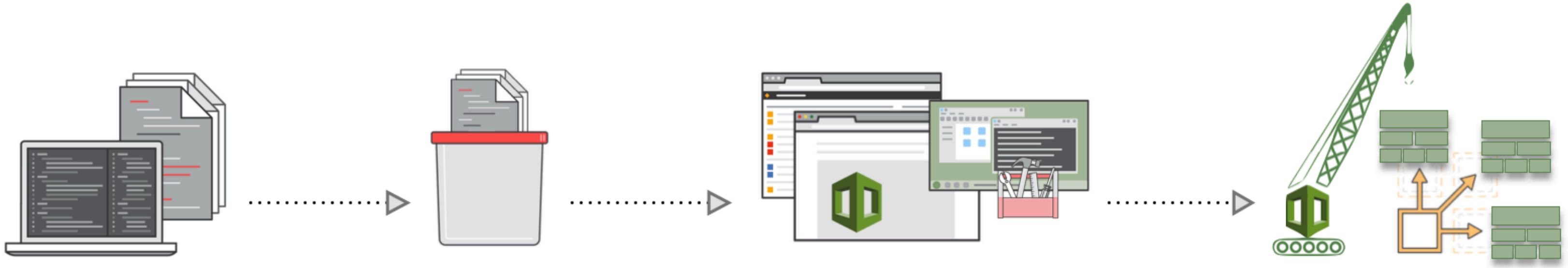
- Because **I could attach a Lambda function to any event that interests me**.
  - For example, instead of writing a dummy function such as the above, I could have put in the code that transforms a picture from color to grayscale, using e.g. our well-known `single.py` program.
- I could have then connected my Lambda function to an S3 bucket, so that any time a new picture is uploaded to the bucket, my function runs on some (dynamically provisioned) AWS resource, and automatically generates a grayscale version of the image.
  - Note that in this case I do not have to explicitly start up any VMs nor containers!
  - You could connect a Lambda e.g. to S3, to Alexa, to changes in a DB, to something being published in a queue, to an IoT device, to a MapReduce workflow, etc.
- Imagine for instance to connect a Lambda function to a DNA sequencer: the function could dynamically process the received data and do *something* (which could also be rather complex) with it, as data is being produced, all without you requiring to explicitly instantiate, run or monitor anything!

Template-based orchestration

- We have seen Function as a Service as a way of abstracting from resource descriptions in Cloud computing. This is handy and useful, but sometimes applications need to have a higher-level description than a “function”, because they have several components.
- 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.



Code your infrastructure from scratch with the CloudFormation template language, in either YAML or JSON format, or start from many available sample templates

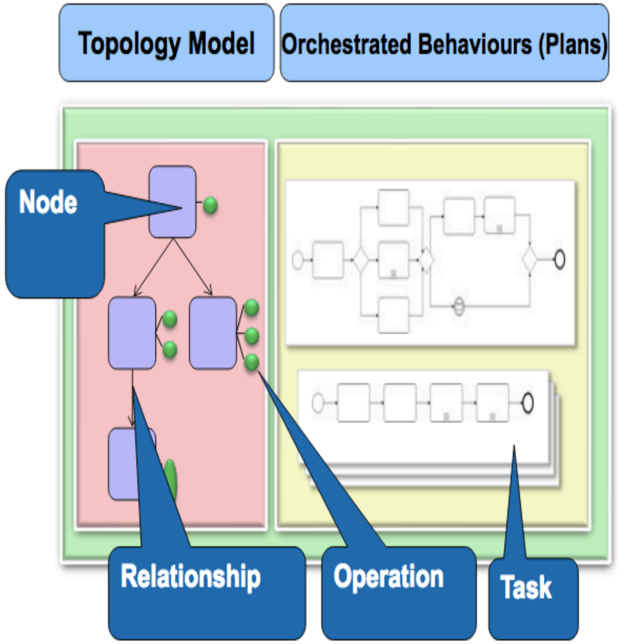
Check out your template code locally, or upload it into an S3 bucket

Use AWS CloudFormation via the browser console, command line tools or APIs to create a stack based on your template code

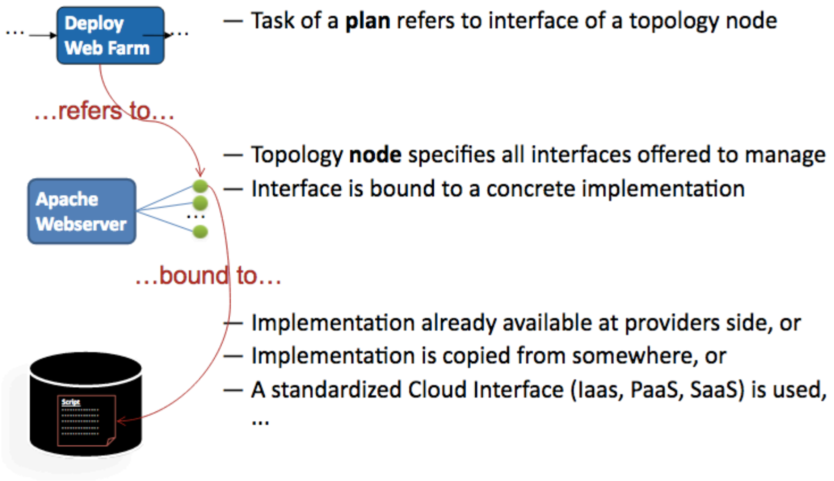
AWS CloudFormation provisions and configures the stacks and resources you specified on your template

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



— Task of a **plan** refers to interface of a topology node

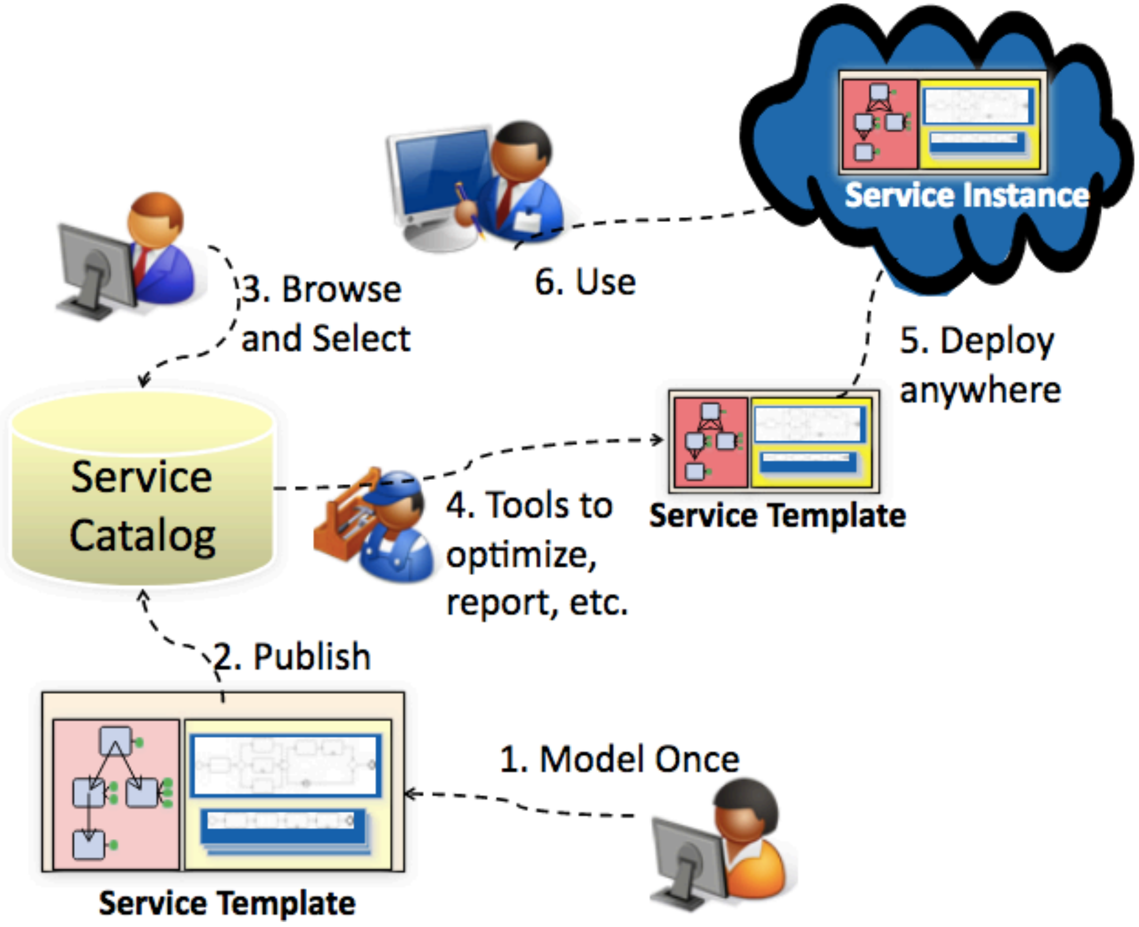
— Topology **node** specifies all interfaces offered to manage

— Interface is bound to a concrete implementation

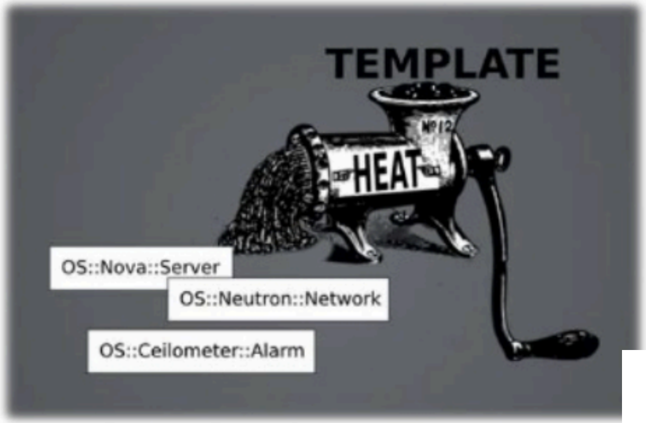
— Implementation already available at providers side, or

— Implementation is copied from somewhere, or

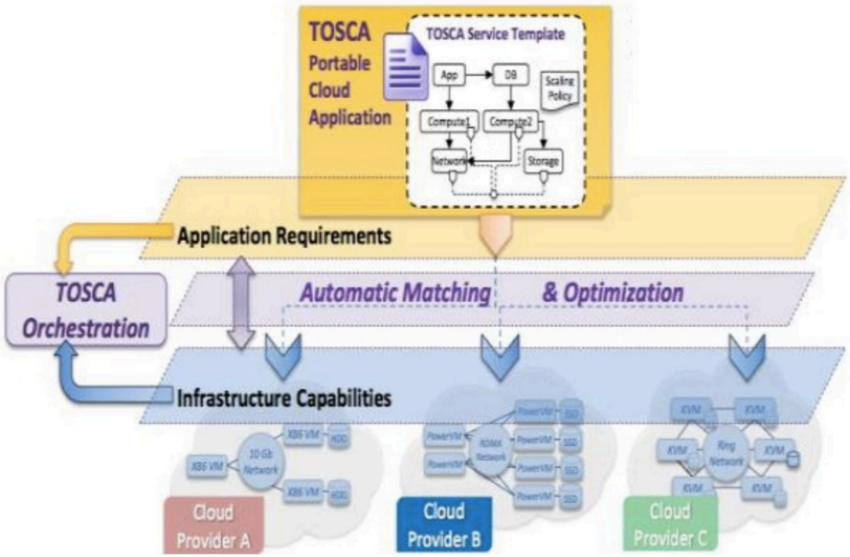
— A standardized Cloud Interface (Iaas, PaaS, SaaS) is used, …



HEAT vs TOSCA

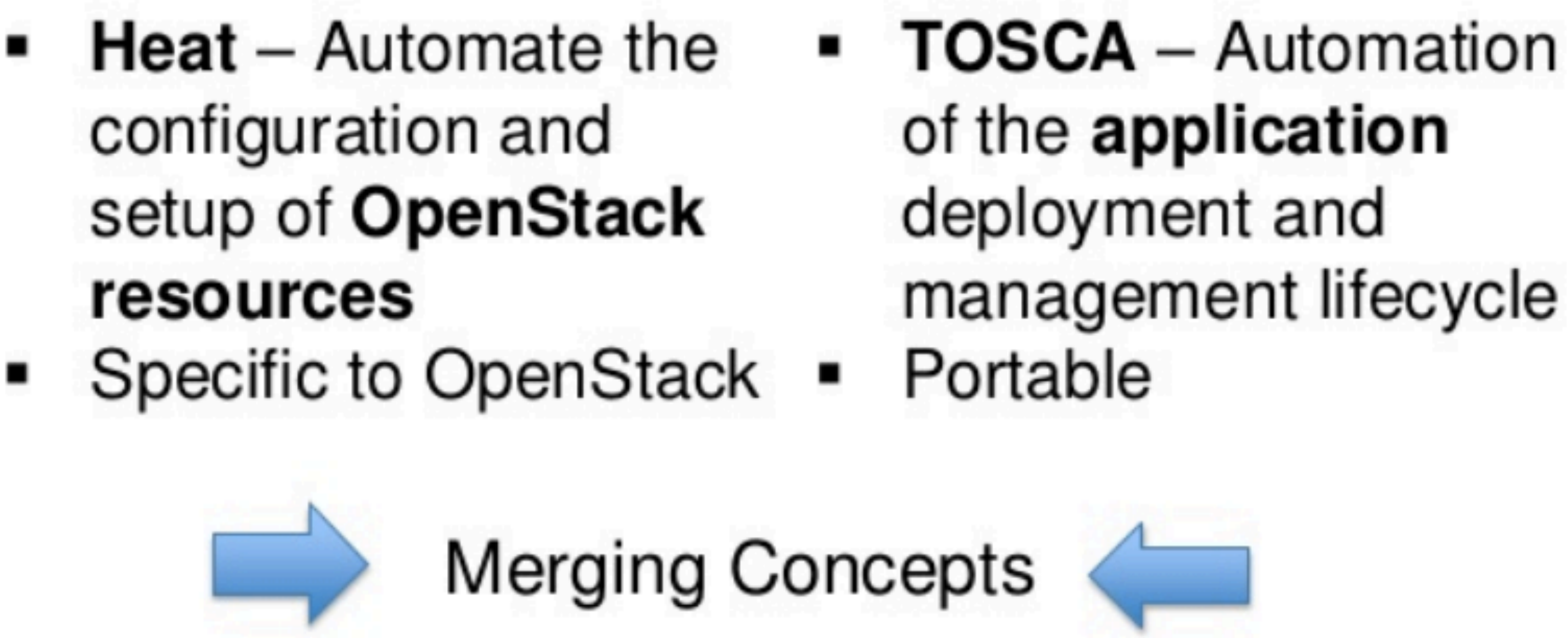


Heat provides a mechanism for orchestrating OpenStack resources through the use of modular templates.



TOSCA defines the interoperable description of applications; including their components, relationships, dependencies, requirements, and capabilities….

Comparing TOSCA & HEAT



- **Heat** – Automate the configuration and setup of **OpenStack resources**
- Specific to OpenStack

- **TOSCA** – Automation of the **application** deployment and management lifecycle
- Portable

Merging Concepts

What can you do with a TOSCA-driven solution?

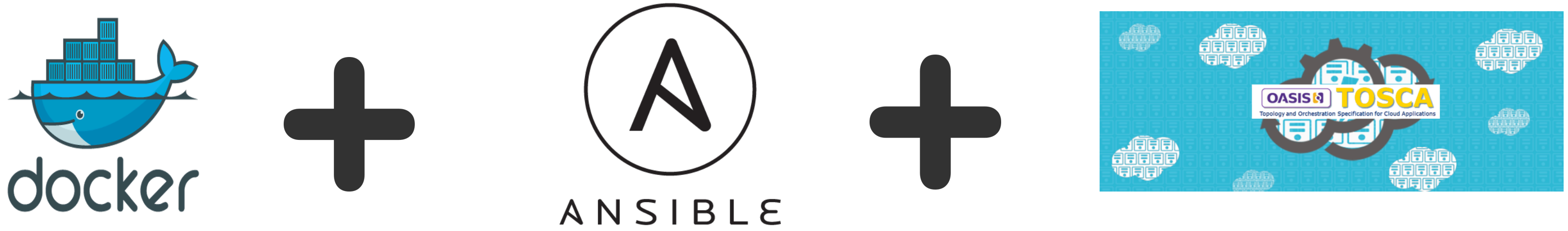
- TOSCA and other template-driven orchestration mechanisms allow us to realize **service composition**, i.e. to combine different services to implement complex topologies.
- An example of service composition developed within INFN is **DODAS** (Dynamic On-Demand Analysis Service), where we facilitate the deployment of relatively complex set-ups on *any* cloud provider with almost zero effort.
- DODAS currently provides support to generate:
  - **An HTCondor-based batch system as a Service**
  - **A Big Data platform for Machine Learning as a Service**
  - **Plus extensions of these two** integrating community-specific services.

Why is this useful

- Regardless of the details of DODAS, what is important is often to have solutions in science that allow:
  - Creation and management of **on-demand systems** (batch or Spark, for example) for data processing.
    - But these should not be tied to a specific cloud provider (e.g. Amazon).
  - Exploitation of **opportunistic computing**.
    - Intended as resources not necessarily or permanently dedicated to a specific experiment and/or activity.
  - **Elastic extension** of existing facilities.
    - To absorb peaks of resource usage.
    - To accommodate workflows with special requirements.

The DODAS architectural pillars

- Everything is cloud and experiment agnostic as much as possible.
- There are three major handles that make service composition in DODAS highly customizable:

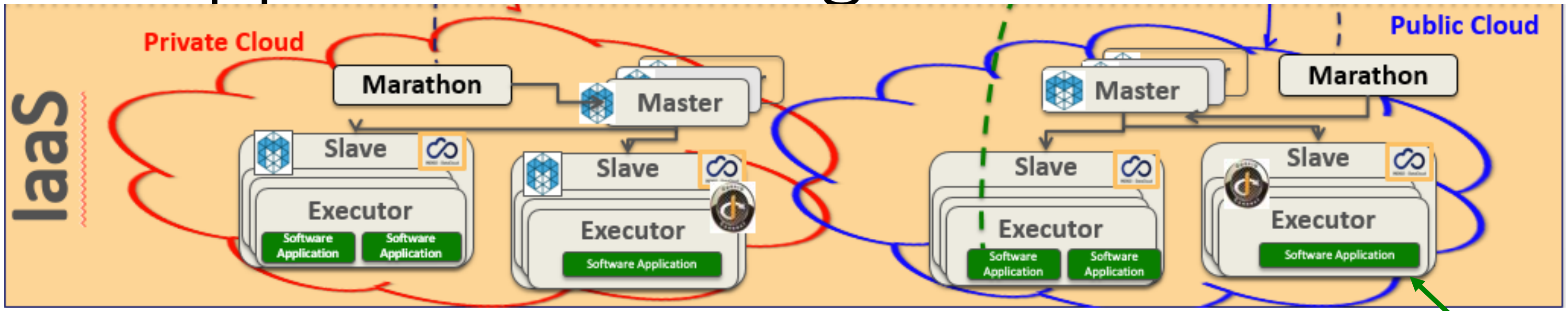

docker
ANSIBLE
OASIS TOSCA
Topology and Orchestration Specification for Cloud Applications

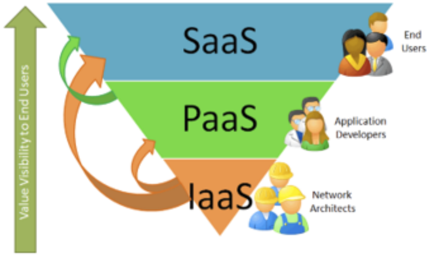
To support user tailored computing environments

To automate configuration and deployment of custom services and/or dependencies

To define input parameters and customize the workflow execution

Application management



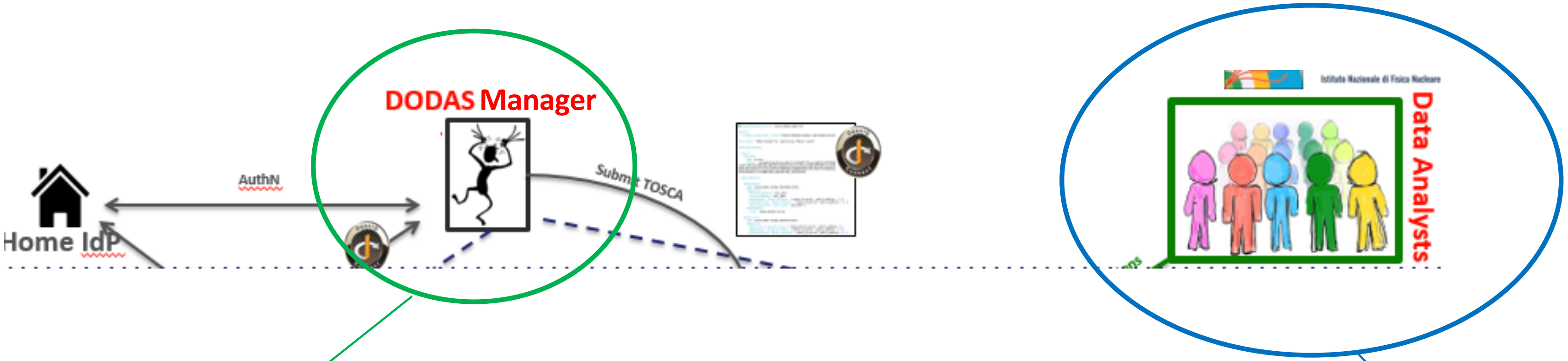


- In DODAS, we use Mesos for resource management and Marathon, a *Mesos framework* to launch long-running applications, for container orchestration (Kubernetes is also supported).
  - Container orchestrator is the layer responsible for the execution of end user services.
  - Any framework and/or software application can run on a DODAS-provided cluster.
    - DODAS provides by default two set of recipes, to create HTCondor & Spark clusters.

docker

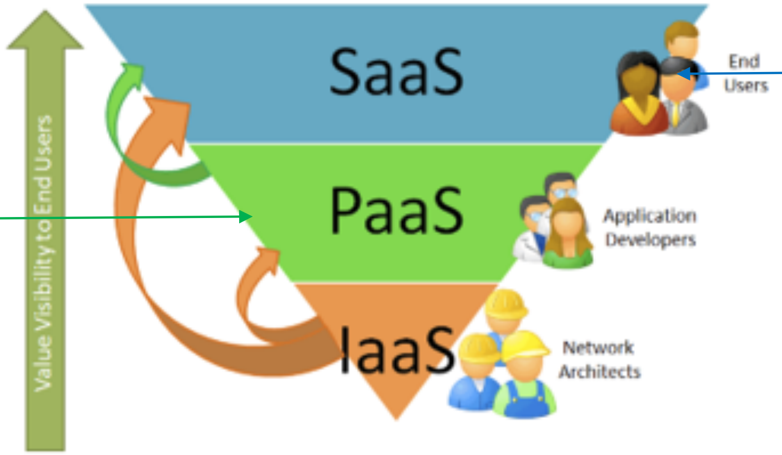
To support user tailored computing environments

DODAS roles

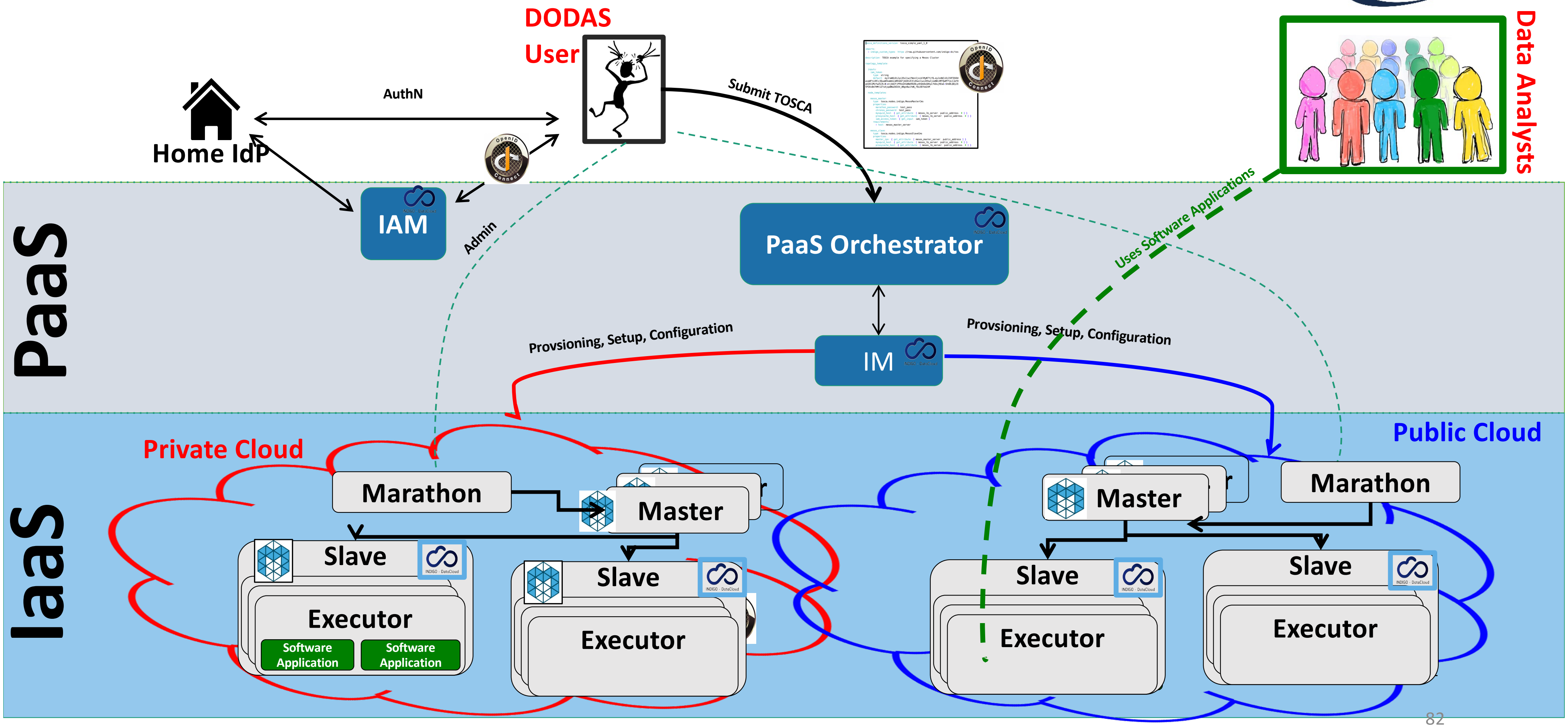


Care only about software applications

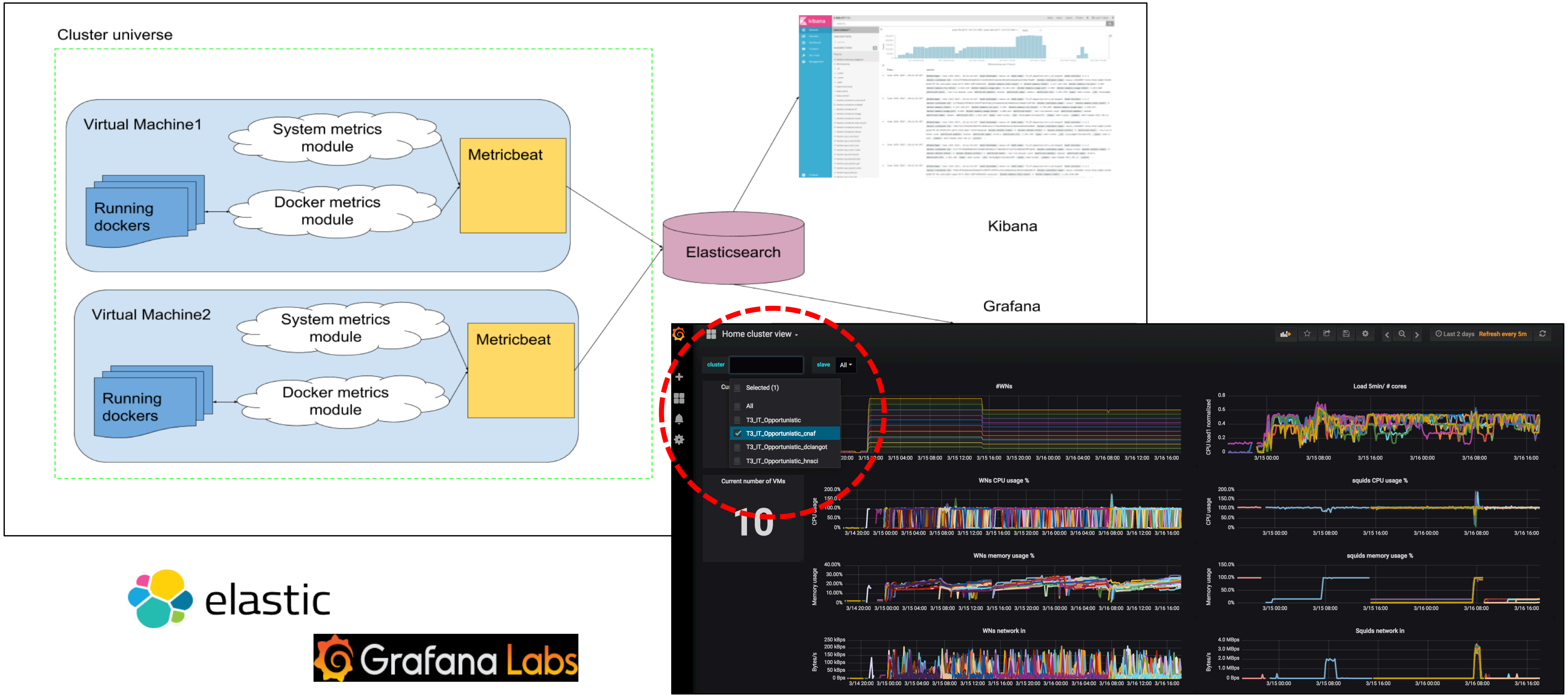
Interact with DODAS PaaS services



The Big Picture



The DODAS Monitoring System



Cluster universe

Virtual Machine1

System metrics module

Docker metrics module

Running dockers

Metricbeat

Virtual Machine2

System metrics module

Docker metrics module

Running dockers

Metricbeat

Elasticsearch

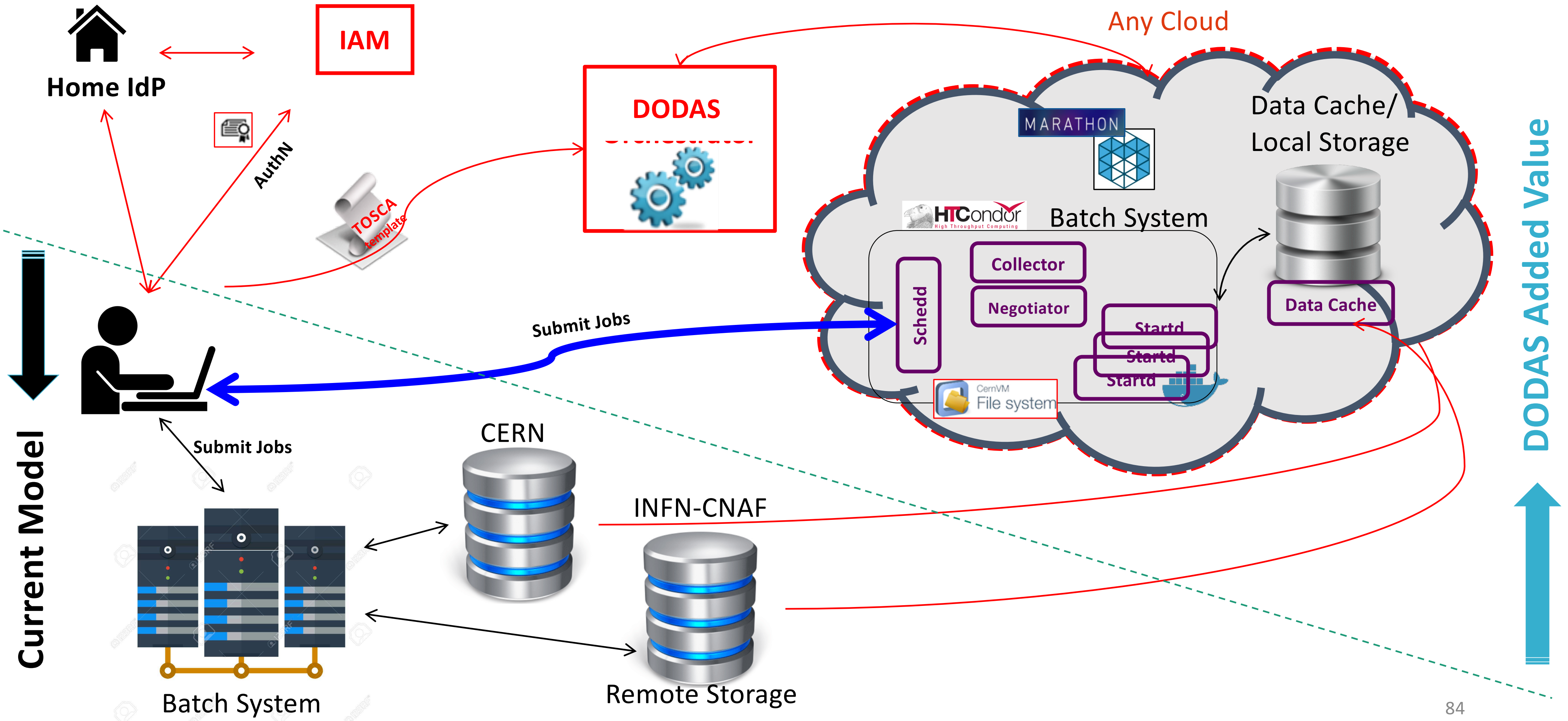
Kibana

Grafana

elastic

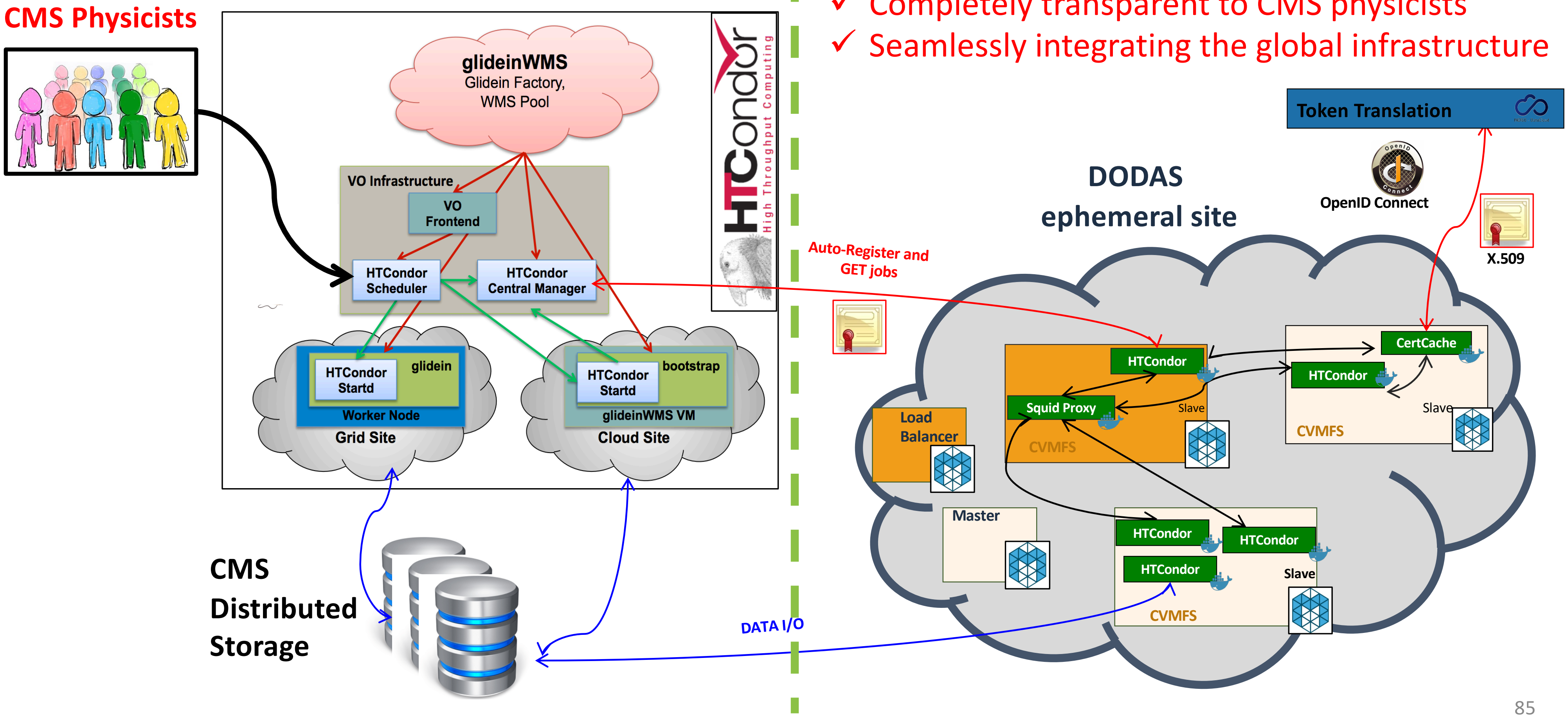
Grafana Labs

Batch System as a Service: the AMS use case

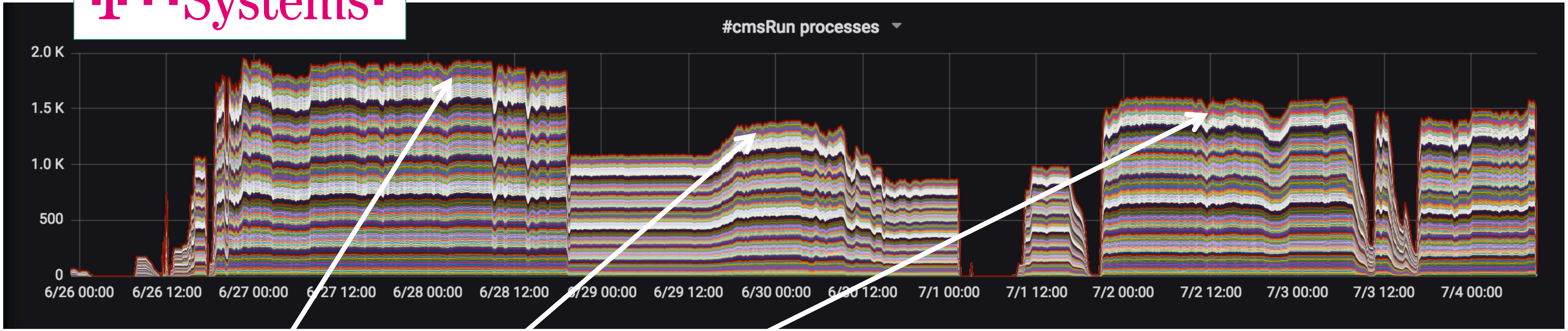


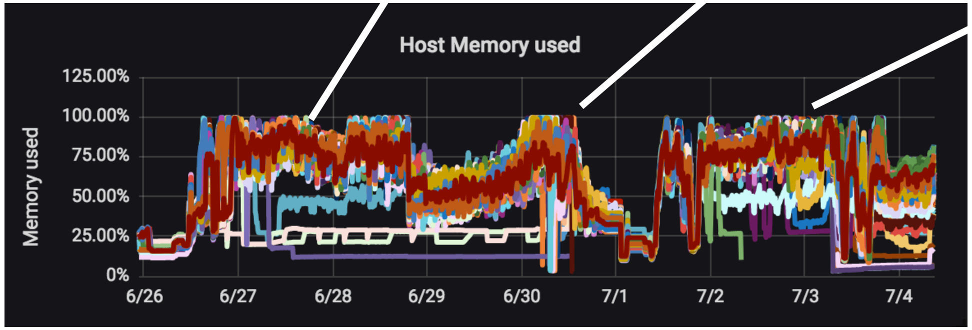
HTCondor Pool Extension: the CMS use case

- ✓ Completely transparent to CMS physicists
- ✓ Seamlessly integrating the global infrastructure



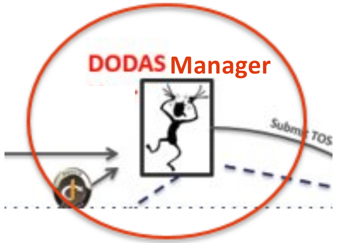
Elastic use of resources





- Elasticity and self-healing
- Stability over days/weeks (120k jobs)
- Handling "special requirements"
high memory jobs

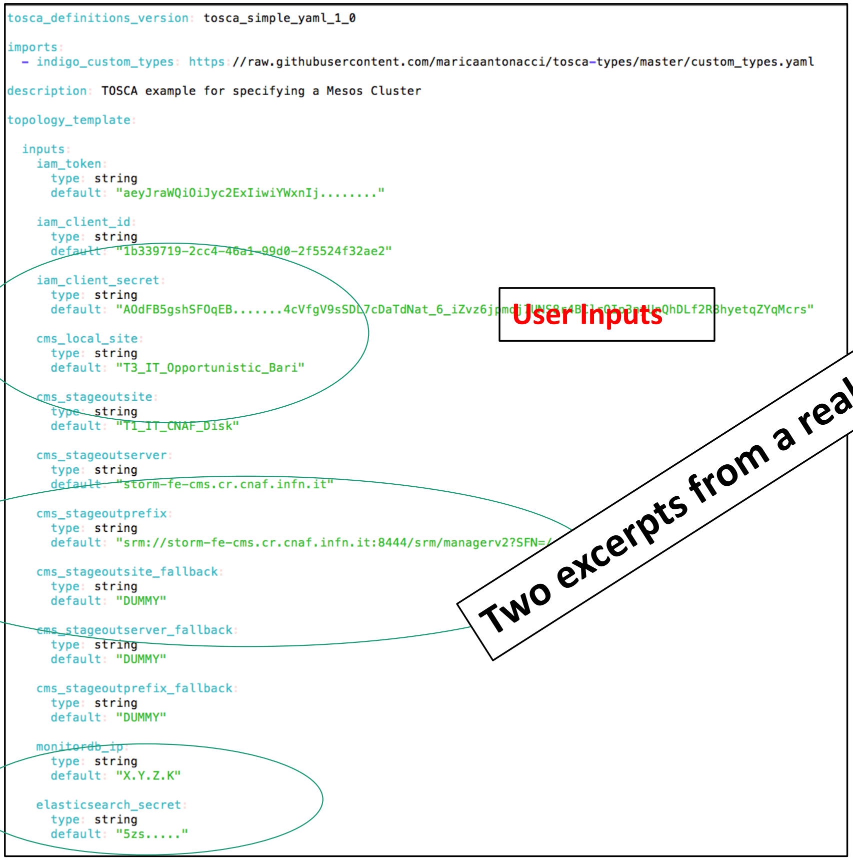
A sample DODAS TOSCA template



To define input parameters and customize the workflow execution

ANSIBLE

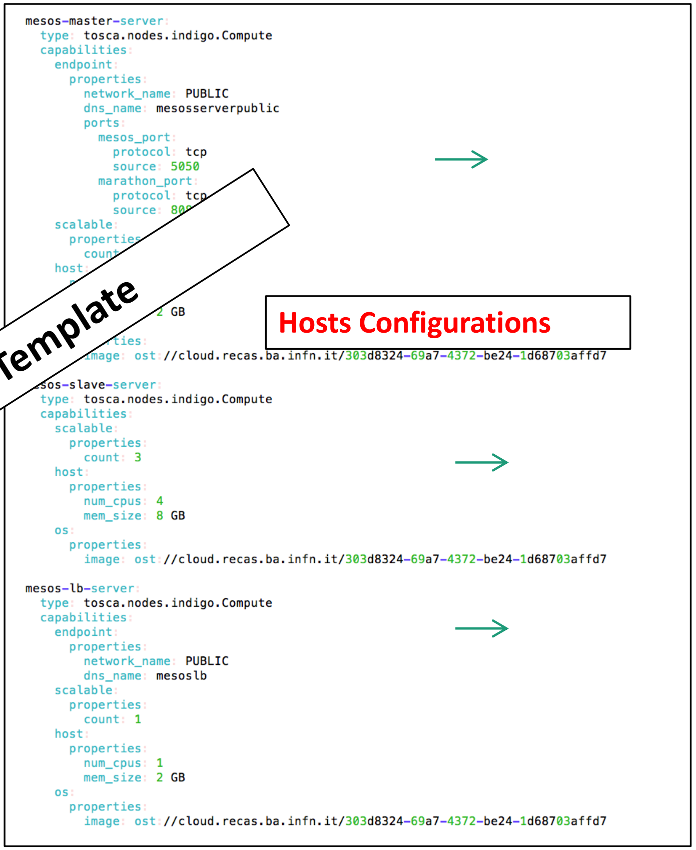
To automate configuration and deployment of custom services and/or dependencies



```
tosca_definitions_version: tosca_simple_yaml_1_0

imports:
  - indigo_custom_types: https://raw.githubusercontent.com/maricaantonacci/tosca-types/master/custom_types.yaml

description: TOSCA example for specifying a Mesos Cluster

topology_template:

  inputs:
    iam_token:
      type: string
      default: "aeyJraWQiOiJyc2ExIiwiYWxnIj........"

    iam_client_id:
      type: string
      default: "1b339719-2cc4-46a1-99d0-2f5524f32ae2"

    iam_client_secret:
      type: string
      default: "AOdFB5gshSFOqEB.......4cVfgV9sSDL7cDaTdNat_6_iZvz6jpmdj[illegible]QhDLf2R8hyetqZYqMcrs"

    cms_local_site:
      type: string
      default: "T3_IT_Opportunistic_Bari"

    cms_stageoutsite:
      type: string
      default: "T1_IT_CNAF_Disk"

    cms_stageoutserver:
      type: string
      default: "storm-fe-cms.cr.cnaf.infn.it"

    cms_stageoutprefix:
      type: string
      default: "srm://storm-fe-cms.cr.cnaf.infn.it:8444/srm/managerv2?SFN=/

    cms_stageoutsite_fallback:
      type: string
      default: "DUMMY"

    cms_stageoutserver_fallback:
      type: string
      default: "DUMMY"

    cms_stageoutprefix_fallback:
      type: string
      default: "DUMMY"

    monitordb_ip:
      type: string
      default: "X.Y.Z.K"

    elasticsearch_secret:
      type: string
      default: "5zs....."
```

User Inputs

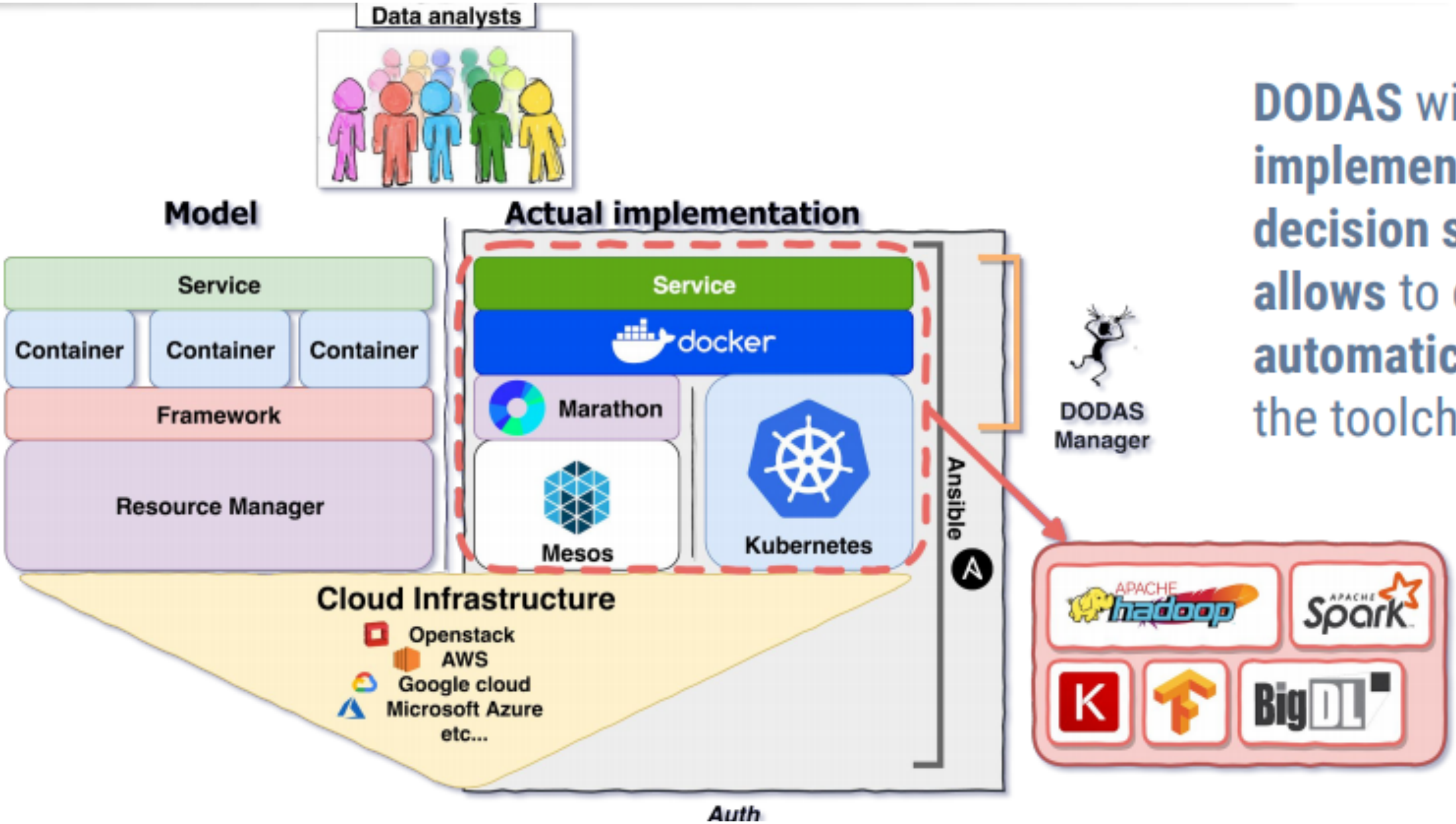
Two excerpts from a real Template



```
mesos-master-server:
  type: tosca.nodes.indigo.Compute
  capabilities:
    endpoint:
      properties:
        network_name: PUBLIC
        dns_name: mesosserverpublic
        ports:
          mesos_port:
            protocol: tcp
            source: 5050
          marathon_port:
            protocol: tcp
            source: 80
    scalable:
      properties:
        count
    host:
      [illegible]
        [illegible] 2 GB
    [illegible]
      [illegible]ties:
        image: ost://cloud.recas.ba.infn.it/303d8324-69a7-4372-be24-1d68703affd7

[illegible]sos-slave-server:
  type: tosca.nodes.indigo.Compute
  capabilities:
    scalable:
      properties:
        count: 3
    host:
      properties:
        num_cpus: 4
        mem_size: 8 GB
    os:
      properties:
        image: ost://cloud.recas.ba.infn.it/303d8324-69a7-4372-be24-1d68703affd7

mesos-lb-server:
  type: tosca.nodes.indigo.Compute
  capabilities:
    endpoint:
      properties:
        network_name: PUBLIC
        dns_name: mesoslb
    scalable:
      properties:
        count: 1
    host:
      properties:
        num_cpus: 1
        mem_size: 2 GB
    os:
      properties:
        image: ost://cloud.recas.ba.infn.it/303d8324-69a7-4372-be24-1d68703affd7
```

Hosts Configurations

Not only HTCondor

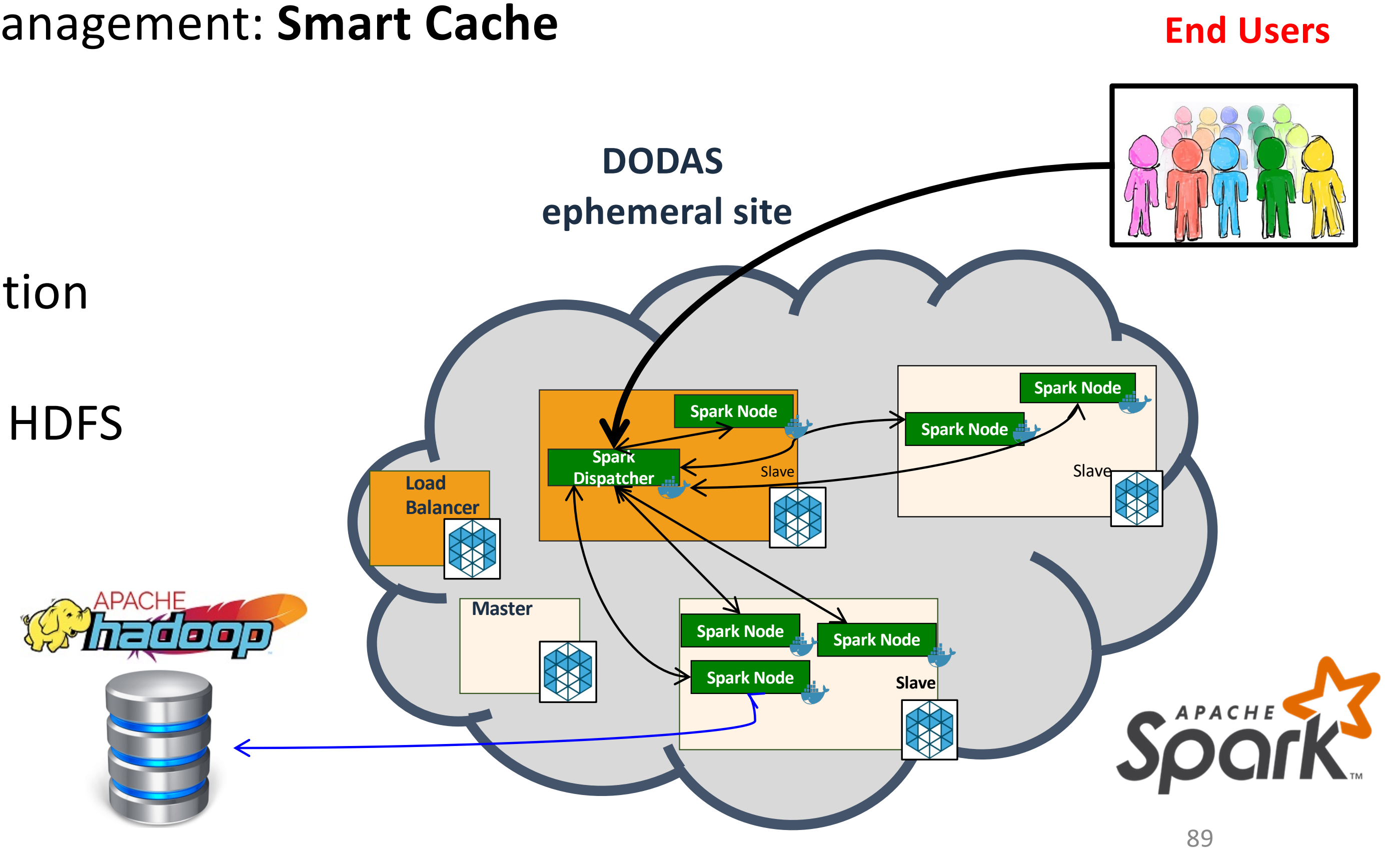


DODAS will be used to **implement** a **smart cache decision service** because it **allows** to **compose automatically the blocks** of the toolchain.

DODAS for Machine Learning as a Service

- Analysis of “Data Cache” related metadata flow
  - To improve caching layer management: **Smart Cache**

1. Reading HDFS@CERN data
2. Data enrichment and reduction with Spark jobs
   - Storing of output data to HDFS
3. Analysis of structured data



Recap of Cloud Automation

- We covered some basic concepts about **Cloud Automation**.
- After explaining what Cloud Automation is, we discussed the difference between **microservices** and **monoliths**.
- We then explored the **DevOps principles**, and discussed the "**Continuous Approach**" (Continuous Integration, Development, Learning, etc.).
- We then moved on to discuss container orchestration, starting with **Docker Swarm**. As hands-on, we created a Swarm cluster load balancing multiple web servers, distributed across multiple nodes.
- After Swarm, we described **Kubernetes and Mesos**, and provided a comparison between these three orchestration tools.
- Simplification in Cloud Automation led us to consider **Infrastructure as Code**, followed by the new paradigm of **Serverless Technologies (or FaaS)**. We showed a simple FaaS example with AWS Lambda.
- We finished our Cloud Automation journey considering two high-level languages for complex template-based orchestration of resources: **AWS Cloud Formation and TOSCA**, providing an example (DODAS) of **service composition**.