Elastic clusters and cloud-aware applications

Dario Berzano

CERN

Scuola RECAS di Cloud Computing - Bari, 27.11.2014

Distributed computing and batch jobs

Use case: HPC and distributed computing

- HPC = High Performance Computing
- Our use case (LHC, Grid, etc.): distributed computing
  - divide a big task into (mostly) independent jobs
  - queue them and run them separately on a batch system
- Geographic distribution
  - several computing centers (tiers) with their batch systems
  - federated: independent but capable of working together

Local batch clusters

- Here we are interested in local batch clusters
  - they can be also made part of a federation (*i.e.* Grid) in case
- Topology of the simplest local batch cluster:
  - one head node
  - several identical workers

Batch system: head node

- Controls job submission and available resources
- Available resources
  - how many workers are available
  - which ones are running jobs, which ones are free
- Job submission and lifecycle
  - you place several jobs in a queue
  - customizable priority: some jobs are more important than others
  - match making: if a worker is capable of executing a certain job and it is idle, dispatches a waiting job there

Batch system: worker nodes

- They are essentially number crunchers
- Controlled by the head node
  - they execute what they are told to do

Cloud aware and cloud independent clusters and applications

IaaS, PaaS, administrative domains

infrastructure

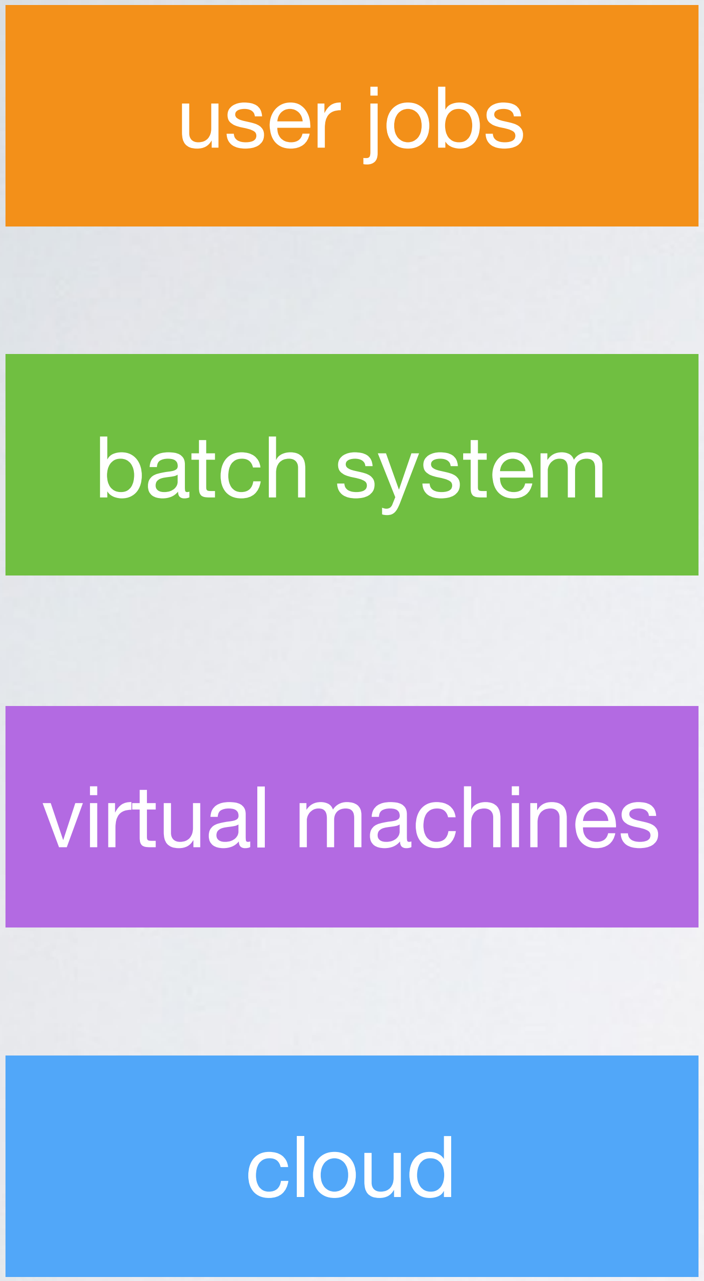
virtual cluster administrator
she manages instantiation of VMs:
sees the virtual cluster as IaaS

platform

user submitting batch jobs
uses a service transparently on VMs:
sees the virtual cluster as PaaS

- The same virtual cluster is seen both as IaaS and PaaS:
  - Infrastructure/Platform as a Service are a matter of perspective
- Batch clusters have been existing way before the cloud!
  - user interaction with the platform: job submission
  - cloud is transparent: user completely unaware of its existence

Stacking independent layers

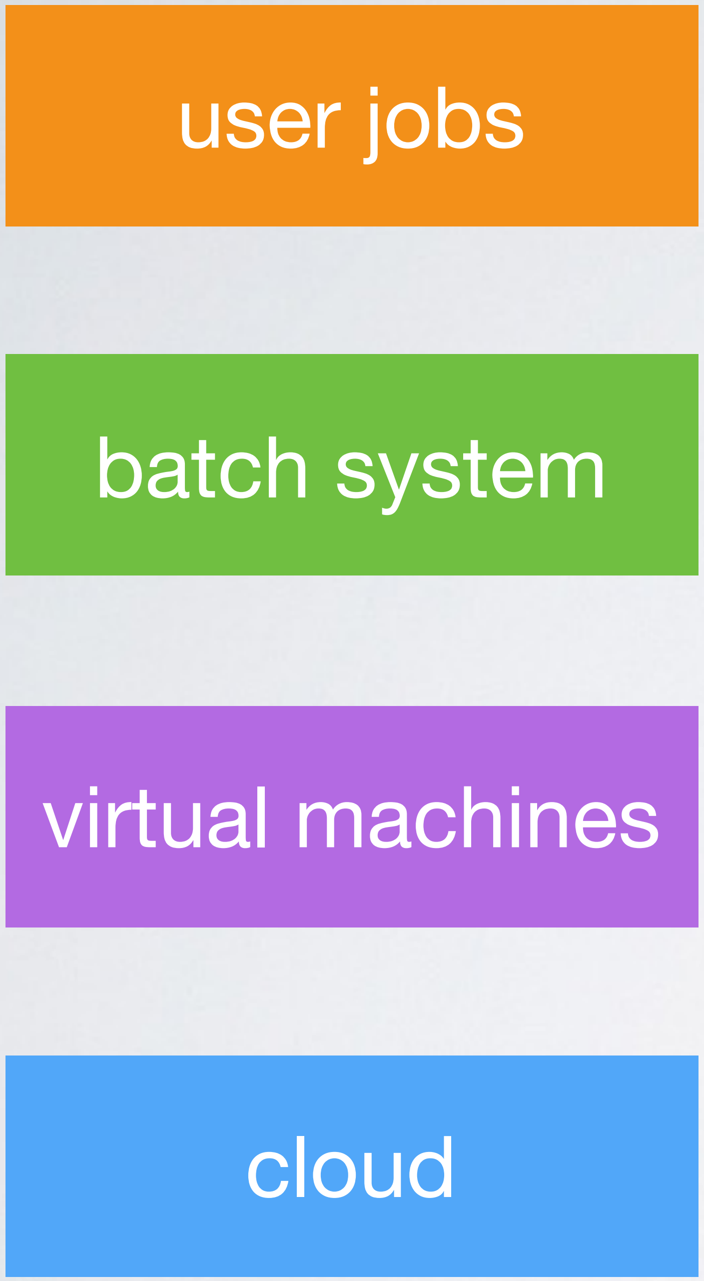


- Architecture composed of *independent* layers
  - we add the cloud
  - use *virtual* machines instead of physical ones
- Why do we add the cloud?
  - keep several use cases (tenants) isolated
  - use resources more efficiently: scale the cluster
  - all of it without changing user's experience
- Layer independency helps us retaining the former user interface (job submission)

Cloud awareness

- Clouds can be a very troubled environment
  - resources are diverse: same profile VMs can behave very differently
  - virtual machines are volatile: appear and disappear with zero notice
- Cloud aware applications
  - scale promptly when underlying resources vary
  - deal smoothly with crashes: automatic failover and clean recovery

Each layer must be cloud aware



- user jobs
  - dependency between jobs well defined
  - jobs must be resubmittable easily
  - something handling the workflow must detect a job error and resubmit it automatically
- batch system
  - workers might disappear without notice
  - dynamic addition and removal of workers

More cloud awareness

- Cloud awareness is about not making assumptions
  - no quality assumptions: be prepared for the worst
  - no quantity assumptions: resources as large as they get
- Clouds are forcing programmers to add robustness to their code
  - this is always a good thing, with or without cloud
- Basic rules for a cloud aware application
  - if you kill a job you do not lose important data
  - important data is saved on an external, secure resource

HTCondor: a cloud aware batch system

HTCondor
High Throughput Computing

- Actively developed and widely used since 1988
- Completely free and open source
- Clouds did not exist in 1988 but its design has always been robust
  - has many features to run opportunistically and deal with failures
- HTCondor supports dynamic addition and removal of workers
  - head node does not have a static list of workers
  - workers self-register to the head node when they are up
  - head node removes them from the list when contact is lost

Cloud independency

- Clouds speak different languages: find the Esperanto of clouds
  - work on several clouds at once (*e.g.* OpenStack, OpenNebula)
  - change our cloud controller without impacts on the applications
- Basic operations (*e.g.* start/stop VMs, check status, list images): EC2
  - *de facto* industry standard developed by Amazon for their cloud
  - widely supported on server side by all major cloud controllers (OpenStack, CloudStack, OpenNebula, Eucalyptus)
  - bindings for several programming languages (*e.g.* boto for Python)
- Alternative: OCCI (Open Cloud Computing Interface)
  - open standard, independent from industry, but less supported

Orchestration of elastic clusters

Elastic virtual clusters

- Value added of a virtual cluster: we can scale it
  - add virtual machines when needed
  - turn off idle virtual machines
- Elastic clusters were born for commercial clouds (*e.g.* Amazon)
  - you pay per use: it's better for your wallet to turn off unused VMs
- Elastic clusters are ideal for non-constant usages
  - less used, or completely idle for some time, but also peak times
  - *e.g.* a web application for selling tickets for a Madonna concert
  - *e.g.* an interactive analysis cluster used at days and idle at nights

Orchestration: internal and external

- We can turn on and off virtual machines manually
- Or, we can use an orchestrator to do the work for us
- Two types of orchestration possible
    - Internal orchestration: use something *inside* VMs to control scaling
    - External orchestration: cloud controller is responsible of scaling

External orchestration

- Usually part of the cloud controller (*e.g.* OpenStack Heat)
- Take actions triggered by values of sensors (*e.g.* Ceilometer)
- Cloud-specific: every cloud has its own
- In some cases difficult to get sensor values of what happens *inside* VMs (only external measurements)
- Knowing the cloud status helps orchestrator taking better decisions
  - *e.g.* don't deploy if no resources available instead of trial/error
- *(Next talk will be entirely about OpenStack Heat)*

Internal orchestration

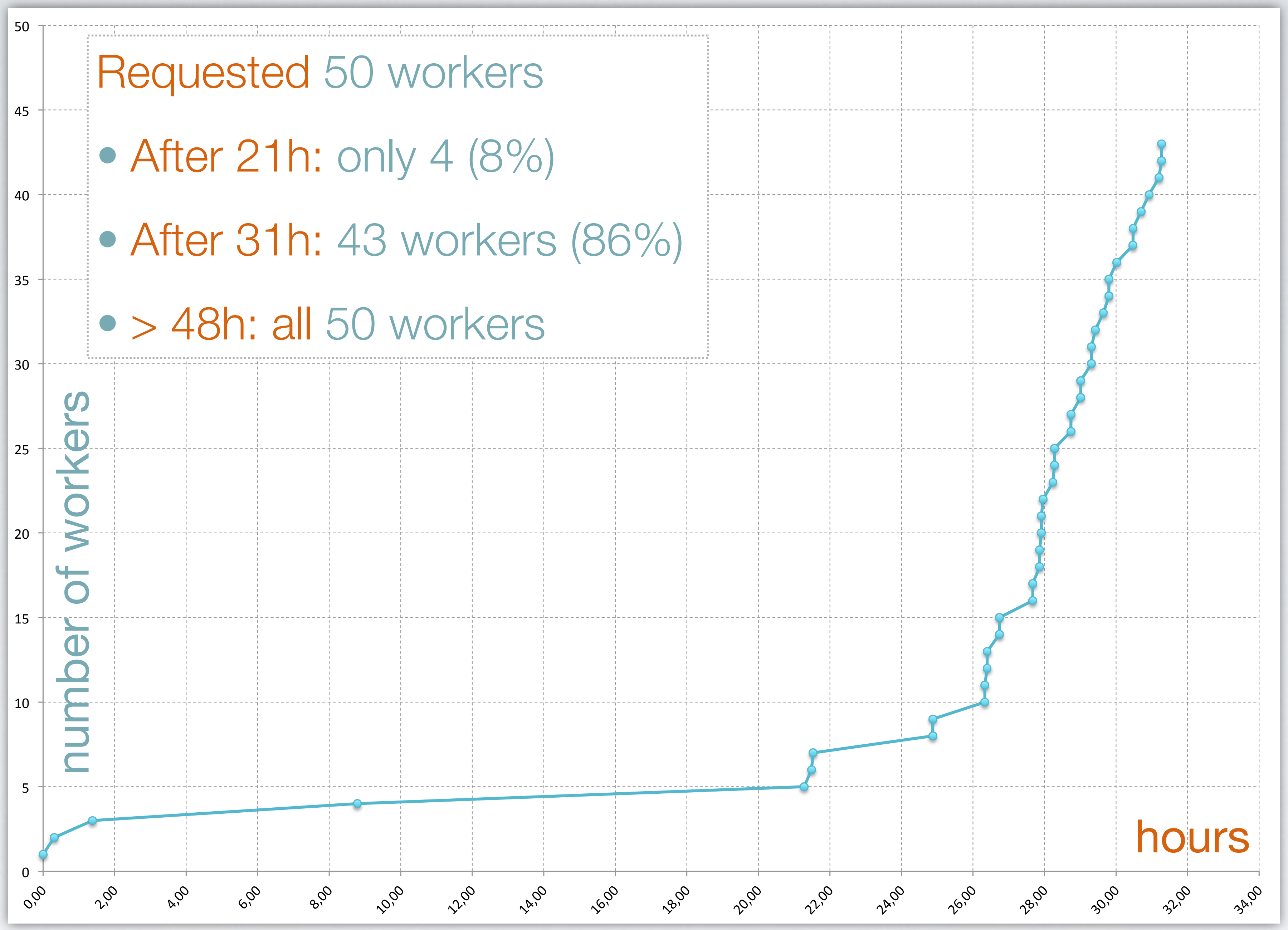
- Much simpler (but simplistic) approach: works for some cases
- Automatic scaling entirely embedded in the virtual cluster
    - no need to install tools outside it: it is self-contained
- Tailor an application-specific sensor to take actions
    - *e.g.* too many waiting jobs → we need more VMs
- Communicate our actions to the cloud via a standard interface
    - *e.g.* use EC2 to tell the cloud controller to start/stop VMs
- Self-contained + standard interface → works on any cloud
- *Following examples are based on this approach*

Starvation

Elastic and inelastic

- On real clouds resources are finite
  - we cannot expand promptly if we do not have free resources
- We want to use resources as much as possible
  - keeping some free resources always is a waste: always saturate
- Elastic clusters do not work in inelastic environments!
  - some tenants must shrink if we want others to expand
  - all, or some, tenants must be polite and relinquish resources
  - or we must force them to do

A real life example: OpenStack @ CERN



- Everybody gets free resources, nobody relinquishes them
- Resources ended quickly: huge inelastic wall when scaling

Self-release and preemptiveness

- Build an elastic tool that turns off unused virtual machines
  - this is almost needed if we pay for resources
  - when we get them for free we are not pushed for using it
- In addition we can have preemptiveness
  - turn off some virtual machines periodically (even if we need them)
  - we are giving others the room to run their VMs

Rolling updates

- Rolling updates: update a running cluster without stopping operations
  - we turn periodically off all VMs running image version 0
  - we request new VMs with image version 1
- Preemptiveness can be used for rolling updates
- Easier if done by internal orchestration
  - internal orchestrator knows how many jobs are running on a specific virtual machine
  - external orchestrator commonly unaware of jobs (layers isolation)

Turn off VMs in a virtual batch cluster

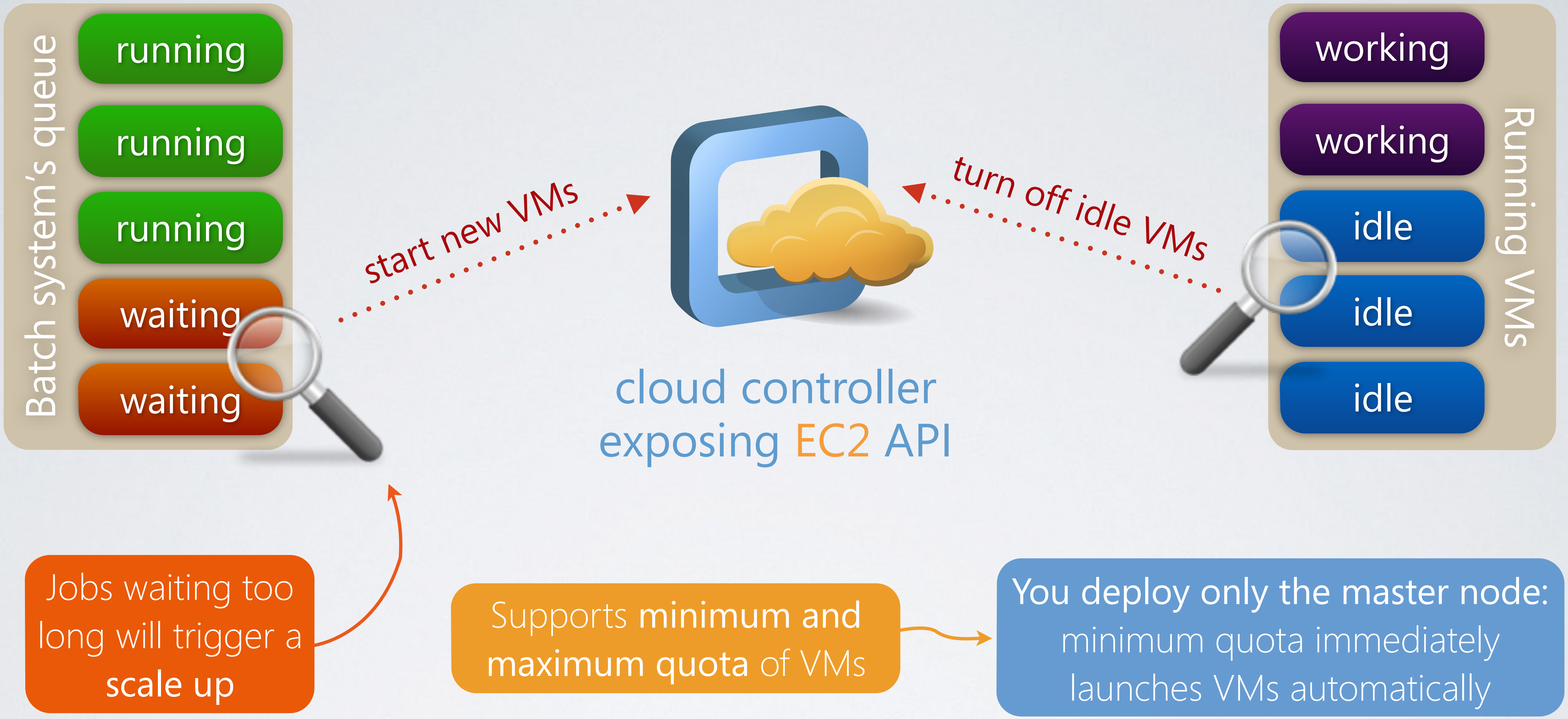
- Put the VM in drain mode
  - VM does not accept new jobs
- Wait for every job on that VM to finish
- When last job has gone, ask the cloud to turn it off
- Never shutdown VM from the inside:
  - cloud controller might think VM is still there and never frees the slot
  - might lead to stale resources

Virtual machines lifetime

- Most practical solution: enforce a maximum lifetime for VMs
  - cloud controller will turn them off after that time
  - does not care if something is running inside or not
- Works better if our use case also relinquishes resources automatically
  - we shut down VM *before* someone forces its shutdown
  - forces our tenant to well behave
  - if she does not, kill VMs in any case

Internal orchestration in practice: elastiq

elastiq



- elastiq: internal orchestrator for HTCondor-based virtual clusters
- github.com/dberzano/elastiq

Installing elastiq

- Do it on the head node: can also be non-virtual
- Prerequisites: HTCondor and boto installed
  - yum install htcondor python-boto
- RPM available: github.com/dberzano/elastiq/releases/latest
  - yum localinstall elastiq-<version>.noarch.rpm
- All configuration in /etc/elastiq.conf (very simple)

Configuration file

- [elastiq]
  - configure main parameters
  - time between checks, idle timeout before killing VMs…
- [ec2]
  - EC2 authentication information
  - user-data to contextualize workers
- [quota]
  - configure minimum and maximum running virtual machines

Section [elastiq] / 1

- sleep_s = 5
- check_queue_every_s = 15
  - how often to check for waiting jobs
- check_vms_every_s = 45
  - how often to check for idle virtual machines
- check_vms_in_error_every_s = 20
  - how often to check for virtual machines in error

Section [elastiq] / 2

- waiting_jobs_threshold = 10
  - have at least 10 jobs waiting before asking for new virtual machines
- waiting_jobs_time_s = 100
  - more than 10 jobs must be waiting for at least 100 s
- n_jobs_per_vm = 4
  - each virtual machine can run 4 jobs

Section [elastiq] / 3

- idle_for_time_s = 3600
  - a VM idle for more than 3600 seconds is killed
- estimated_vm_deploy_time_s = 600
  - if a booted VM does not join cluster within 600 seconds it is killed
- batch_plugin = htcondor
- log_level = 0

Section [quota]

- min_vms = 0
  - minimum number of running virtual machines
  - can be used to automatically deploy a virtual cluster (also static!)
- max_vms = 3
  - maximum number of virtual machines to have

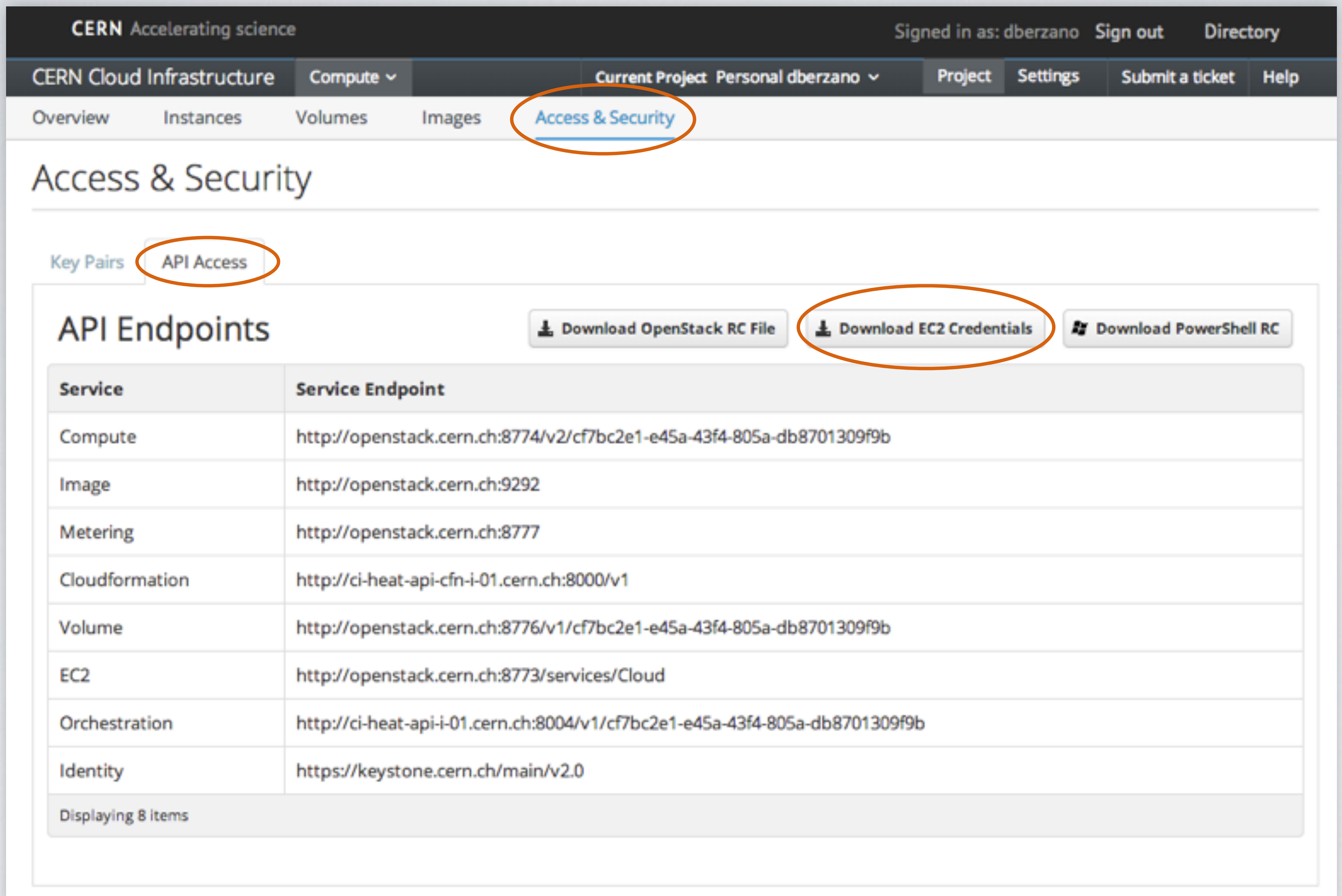
Section [ec2] / 1

- api_url = https://dummy.ec2.server/ec2/
  - server's EC2 endpoint
- aws_access_key_id = my_username
  - your EC2 access key ID
- aws_secret_access_key = my_password
  - your EC2 password

Section [ec2] / 2

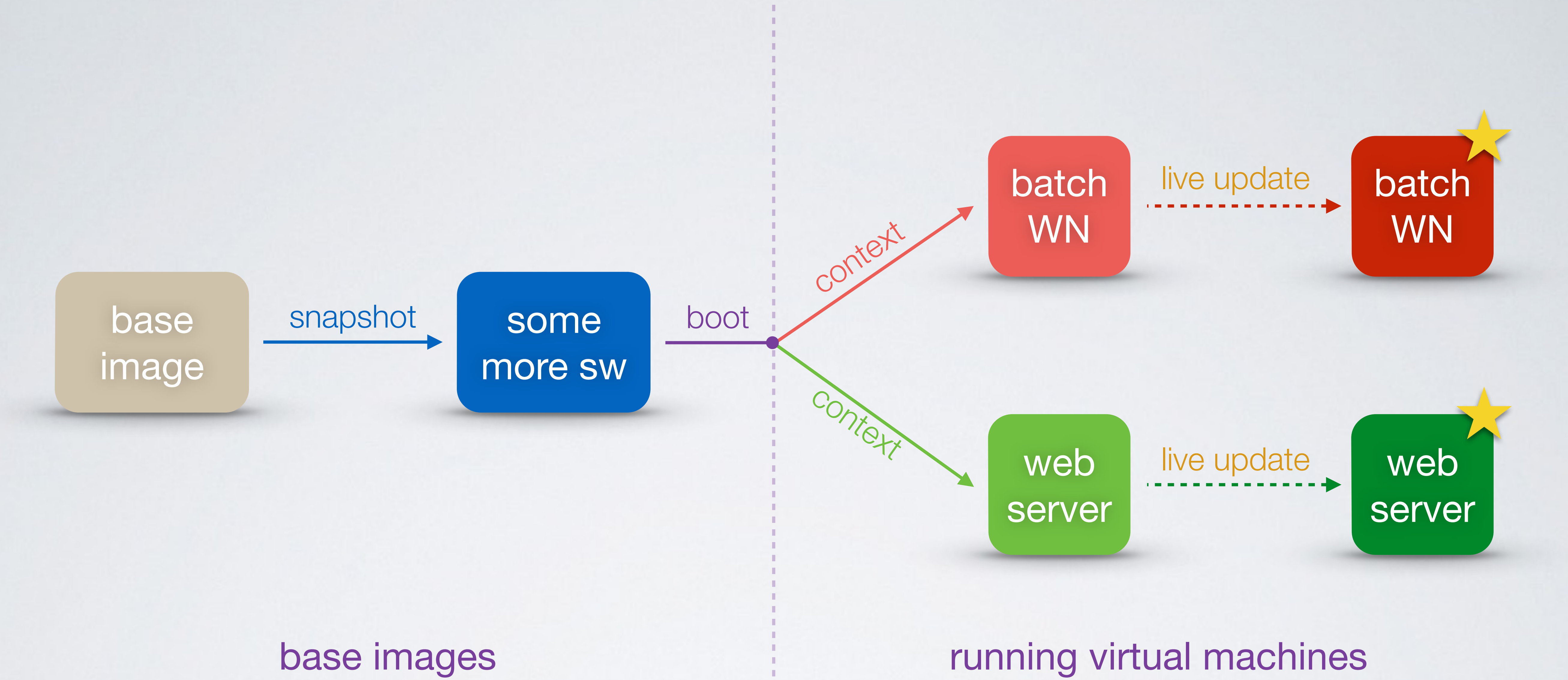
- image_id = ami-00000000
  - AMI ID of workers image *(use euca-describe-images for that)*
- api_version = 2013-02-01
- key_name = your_ssh_key
- flavour = m1.large
- user_data_b64 =
  - contextualization file (user-data) to provide to the new VMs
  - encode text file in a one-line base64
  - cat user-data-workers.txt | base64 | tr -d '\n'

How to find EC2 information in OpenStack



- Download EC2 Credentials > look for variables into .sh file

Specializing a VM: contextualization



- Boot-time procedure to specialize VM: single image for different uses
- Using scripts or standard tools inside the VM (*e.g.* cloud-init)
- VM will find its contextualization file in a known place (*e.g.* webserver)

Application example:
the Virtual Analysis Facility

Inside a Virtual Analysis Facility

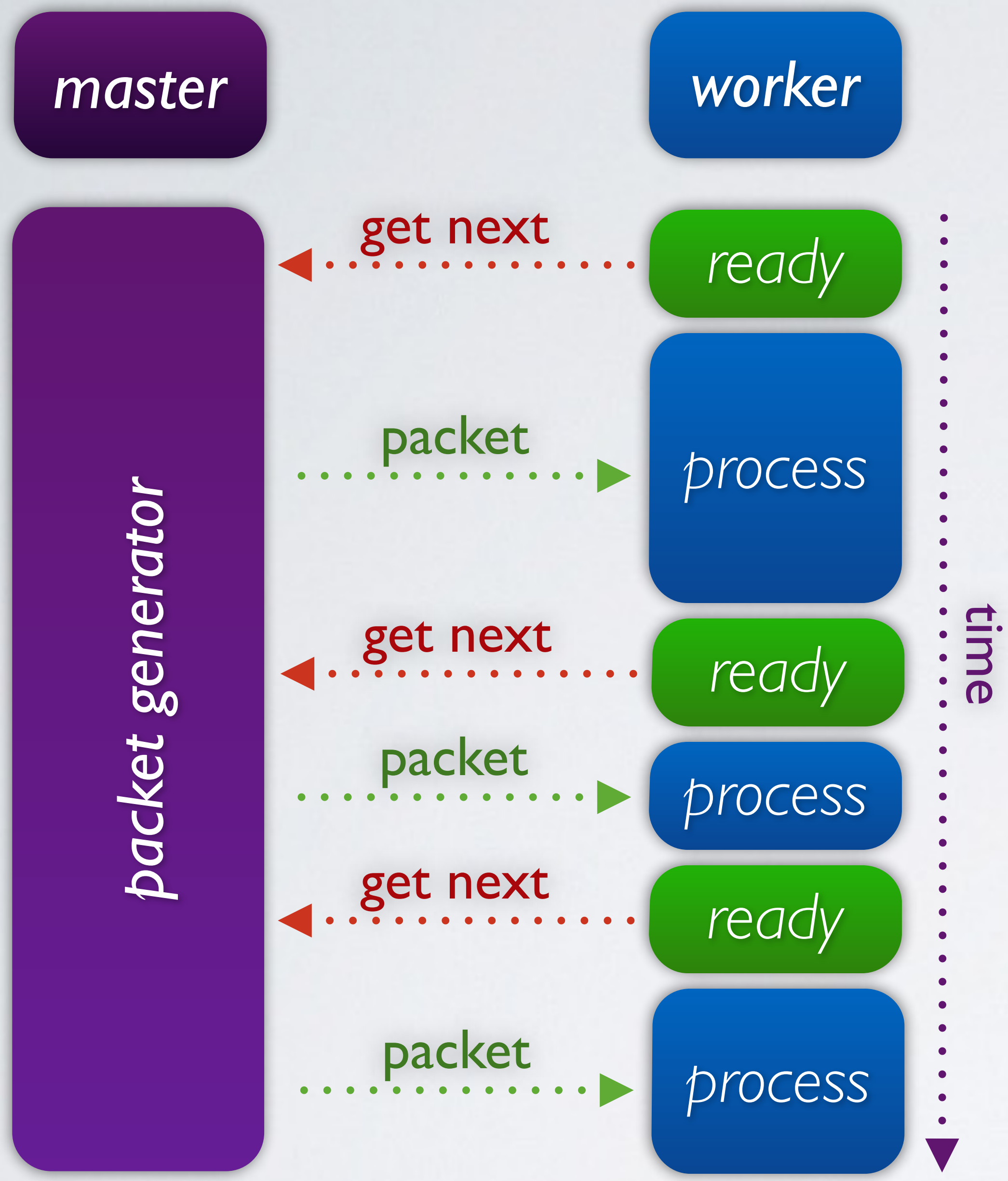
your jobs | virtual machine | HTCondor | elastiq

- A virtual analysis facility is
  - a set of virtual machines
  - running the HTCondor batch system
  - capable of growing and shrinking autonomously with elastiq
- You launch only the head node with elastiq: the rest is automatic
- User interacts only by submitting jobs
- Elasticity is embedded: no external tools are needed
  - only requires EC2 on the hosting cloud

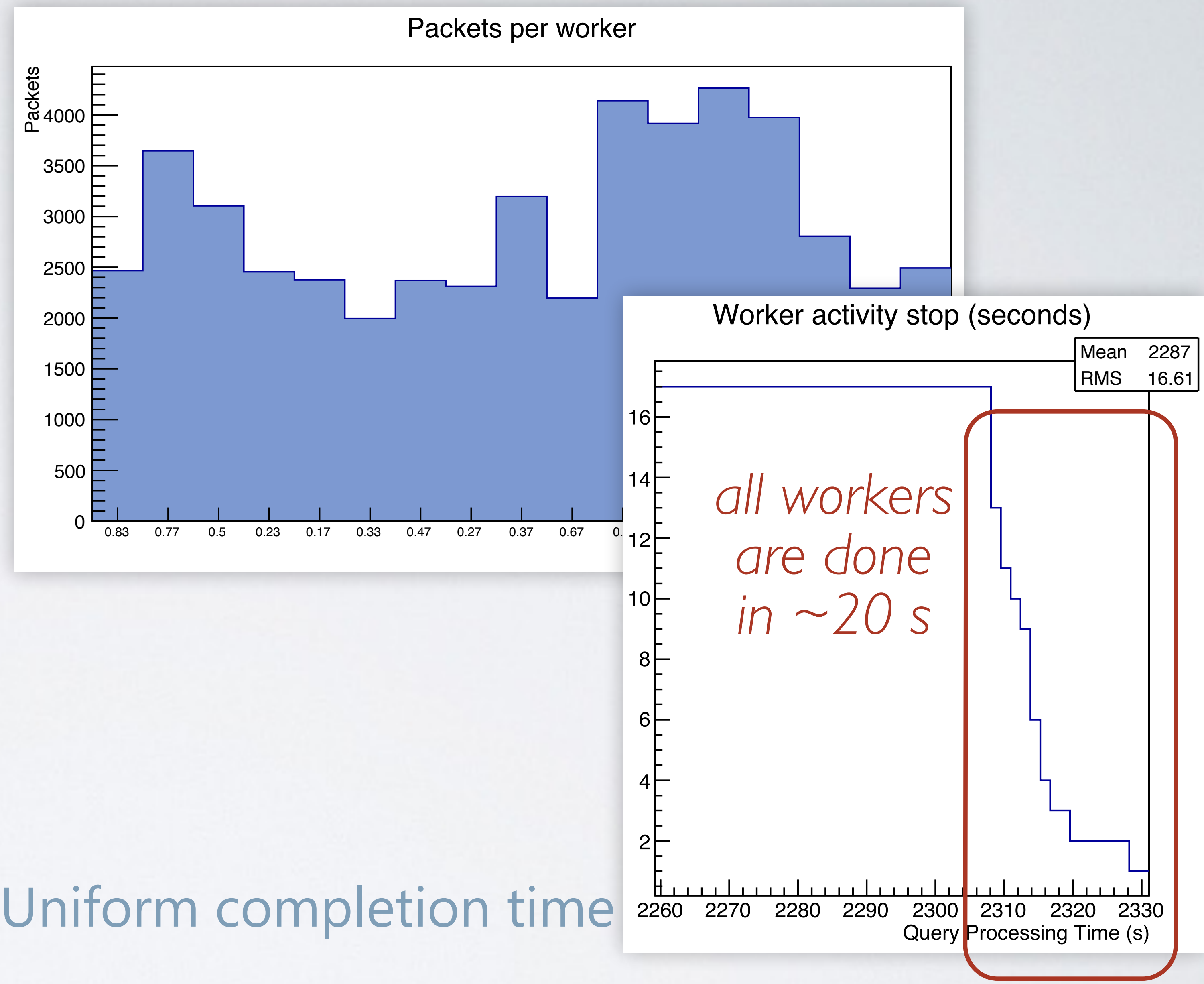
PROOF and PoD

- PROOF: analysis tool specific to High Energy Physics for computing
  - concepts similar to Hadoop: map + reduce
  - distributed analysis: event-based parallelism
  - computing nodes communicate with each other
  - efficient *pull* scheduler: nodes ask for workload when idle
- PROOF on Demand (PoD):
  - submit normal jobs on batch systems
  - they are pilot jobs interactively receiving from PROOF the workload
  - use batch resources in an interactive way

Pull scheduling in PROOF



Nonuniform workload distribution



Uniform completion time

- Also: new PROOF workers can join a processing at any time
- PROOF is cloud aware: assigns workload opportunistically

Grid site on opportunistic clouds

- Some dedicated high performance farms are unused sometimes
  - *e.g.* the High Level Trigger of LHC experiments
- Opportunistic use with no messing with special hardware setup
  - turn them into clouds → run virtual farms on top
- On the ALICE HLT @ LHC we run an opportunistic Grid site
  - based on elastiq: scales when needed
  - seen as a normal Grid site from the outside
  - only with a variable number of resources
- We have introduced elasticity in a transparent way!

ALICE

Makeflow: jobs with dependencies

- Batch jobs are dumb and independent
- Makeflow: describe a workflow of jobs through a manifest
  - text file similar to Makefiles: *target: dependency1 dependency2…*
  - each section is a job whose input is some other job's output
  - controls job submission (and failures) over HTCondor (or others)
- Example: process many physics events and merge all results
  - several jobs, then a single long merge job
- We use elastiq to turn off VMs when only this job is running
- Used for running software tests on the ALICE software

ALICE