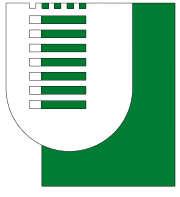
Hands-on Cloud Computing Services

Lecture 1

Gabriele Russo Russo

University of Rome Tor Vergata, Italy

A.Y. 2026/27



TOR VERGATA

UNIVERSITÀ DEGLI STUDI DI ROMA

Overview

- Cloud Computing services in action
  - Amazon Web Services
- Tools for cloud automation
  - Ansible
  - Terraform (and its open-source fork, OpenTofu)
- Examples & Exercises

Course Material

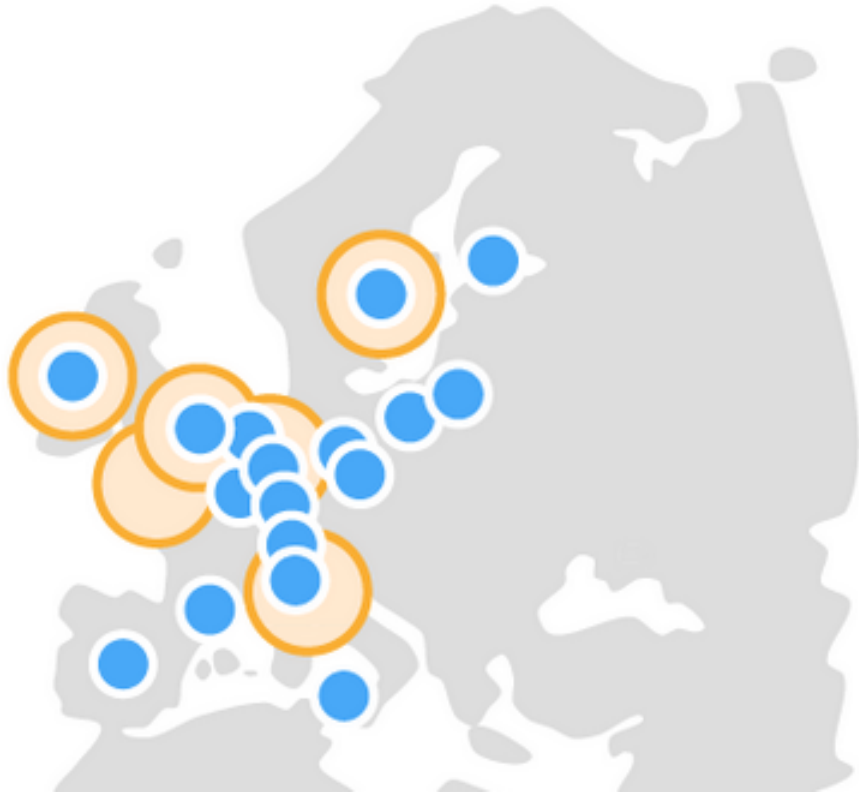
- Relevant slides and code will be published on the web page after the lectures
  - `https://www.ce.uniroma2.it/courses/sdcc2627/`
- Official AWS documentation
  - `https://docs.aws.amazon.com/`
  - most of the content available in Italian as well
- AWS Skill Builder (free courses): `https://skillbuilder.aws/`
- If you are interested in a book that covers these topics (and much more): Arshdeep Bahga, Vijay Madisetti, "Cloud Computing Solutions Architect: A Hands-On Approach", 2019.

Amazon Web Services (AWS)

- One of the major public cloud providers (along with Google Cloud Platform, Microsoft Azure, …)
- 240+ different services
  - **Computation** (EC2, Lambda, …)
  - **Storage** (S3, …)
  - **Machine Learning** (Bedrock, SageMaker, Rekognition, …)
  - **Networking and Content Delivery** (CloudFront, Route 53, …)
  - **Security** (IAM, KMS, Cognito, …)
  - …

AWS Infrastructure

- 39 **regions**, 124 **availability zones** (2 more regions announced)
  - in Italy: Milano (since 2020)
- typically 3+ availability zones in each region
- 500+ Edge locations
  - In Italy: Milano, Roma, Palermo
- Numbers keep growing…



Updated information: `https://aws.amazon.com/it/about-aws/global-infrastructure/regions_az/`

Accessing AWS

- General approach: register an AWS account (with an associated credit card)
  - It is very important to keep AWS credentials safe!
  - Multi-Factor Authentication is required for the root user
- For this course: AWS Academy Learner Lab
  - A temporary AWS account, usable while the lab session is active (*Start Lab* again to reset the session timer)
  - Limited budget (e.g., 100 USD): if you spend all of it, the account is deactivated and all your work is lost
  - Some services are restricted

Interacting with AWS

- Web Console
- AWS CLI
- SDK for various languages (e.g., `Boto3`)
- (Infrastructure-as-Code tools)

Note: regardless of the chosen approach, you need to explicitly select the AWS Region you want to use. Most resources (e.g., EC2 instances) live in a single region and are only visible from that region. A few services are global (e.g., IAM, Route 53, CloudFront).

AWS CLI in practice

- AWS CloudShell: the CLI in the browser, already logged in
  - In the Learner Lab, you can also use the terminal pane of the lab page
- Example: list your EC2 instances

```
$ aws ec2 describe-instances --output table --query \
  'Reservations[].Instances[].[InstanceId,State.Name,PublicIpAddress]'
```

- More on the CLI in the next lectures

Identity and Access Management (IAM)

- IAM controls *who* can do *what* on which resources
- When you register an account, you get a root user
  - Use it only for account-level tasks; MFA is mandatory
- Best practices (not necessary in the Learner Lab of course!):
  - For people: prefer **IAM Identity Center** (SSO, temporary credentials) over IAM users with long-term access keys
  - For applications (e.g., code on EC2 accessing S3): **IAM roles** — no keys stored on the machine[1]

[1] `https://docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html`

Amazon EC2

- Resizable compute capacity in the cloud
- Virtual machines = Instances
- Many different instance types
  - Different amount of vCPUs, memory
  - Different storage
  - Different processors (x86 vs ARM (AWS Graviton) CPUs, GPUs, …)
  - **Different price!**
- Starting from less than 0.01 $/hour

EC2 Purchasing Options

Option	How it works	Discount	Good for
On-Demand	pay per second, no commitment	—	tests, unpredictable loads
Spot	spare capacity; AWS can reclaim the instance with a 2-minute notice	up to 90%	fault-tolerant, flexible jobs (batch, CI)
Savings Plans	commit to a $/hour spend for 1 or 3 years; applies to any instance type (Compute SP)	up to 72%	steady usage
Reserved Instances	commit to a specific instance type in a region for 1 or 3 years	up to 72%	steady usage, fixed configuration

- Discounts are w.r.t. On-Demand prices
- In this course we only use On-Demand instances[2]

[2] `https://aws.amazon.com/ec2/pricing/`

Example

- Let's create an EC2 instance
- **AMI**: we can use Amazon Linux 2023
- **Instance type**: pick something cheap (e.g., `t3.micro` or `t4g.micro` (ARM))[3]
- **Security group**: create a new SG allowing SSH (possibly with source: My IP) and HTTP (source: anywhere)
- Shutdown behavior: `Terminate`
- Create (if necessary) a new **key pair**: store the private key in a safe place! (the public key is installed on the instance)

Connecting to the new instance

```
$ chmod 400 <file.pem>   # else ssh refuses it
$ ssh -i <file.pem> ec2-user@<Public IP/Public DNS>
```

[3] New accounts (since 15 Jul 2025): up to 200 USD of credits for 6 months, instead of 750 free hours/month. `https://aws.amazon.com/free/`

Key pairs

- Key pairs allow EC2 to verify your identity when connecting to an instance via SSH
- Two approaches to setup your key pair:
  1. Key pair created using EC2: you download the private key (`.pem` file) and save it to a **safe** location
  2. Key pair created on your PC (e.g., using `ssh-keygen`): you need to import the **public** key in EC2
- Instructions for both methods: `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/create-key-pairs.html`

Other Ways to Connect

- EC2 Instance Connect
  - SSH from the browser (*Connect* button in the EC2 console)
  - AWS pushes a temporary key to the instance: no `.pem` file needed
  - Port 22 must still be reachable (from the AWS service IP range)
- Session Manager (part of AWS Systems Manager)
  - Shell from the browser or the CLI
  - No open port 22, no key file
  - Needs an instance role with SSM permissions

All options:
`https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/connect.html`

Monitor your Costs!

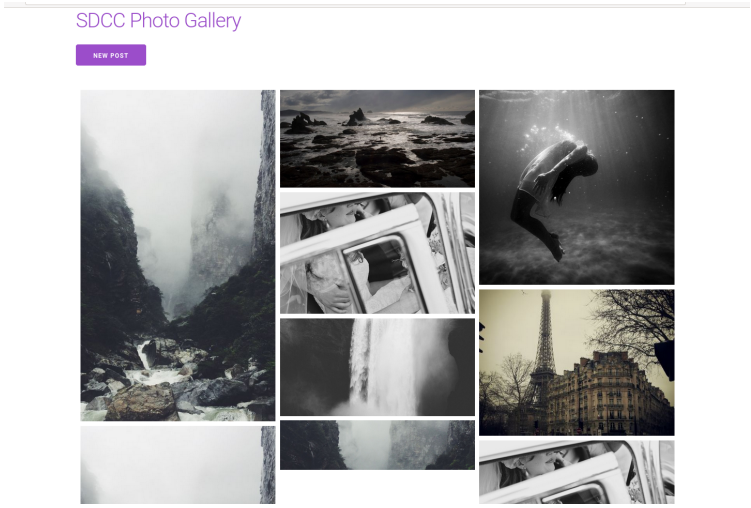
- The **Billing Dashboard** provides useful information to analyze paid and expected costs
- Create a budget with AWS Budgets, to get an e-mail when actual or forecasted costs exceed a threshold
  - Templates: *zero spend budget* (alert at the first cent spent) or *monthly cost budget*
  - `https://docs.aws.amazon.com/cost-management/latest/userguide/budgets-managing-costs.html`
  - Alternative: a CloudWatch billing alarm (`https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/monitor_estimated_charges_with_cloudwatch.html`)
- In the Learner Lab: check your spending at the top of the lab page (updated with a delay of up to 8 hours)

How much does this lecture cost?

- Estimated cost to keep the resources of this lecture (2 $\times$ `t3.micro` + Application Load Balancer) running for a whole day in `us-east-1`: about 2 USD/day
  - instances: $\approx$ 0.50 USD
  - load balancer: $\approx$ 0.55 USD + usage (LCU)
  - public IPv4 addresses (4 $\times$ 0.005 USD/hour): $\approx$ 0.48 USD
- **Note**: since Feb 2024 every public IPv4 address is billed (0.005 USD/hour), even on a free-tier instance
- Stopped instances are not billed, but their EBS volumes are

Example Application: Photogallery

- Web application, written in Python using Flask
- Just a toy: no security, no robustness, …
- We'll use and extend it during the lectures
- You may develop your own customized version



Photogallery: Roadmap

Each version adds 1+ AWS services.
We will try to show all of them during the lectures…otherwise, you will be able to try to extend the app yourself!

Ver.	AWS services	What changes
v1	EC2, AMI, ELB	static photos; footer shows the instance
v2	S3, IAM roles	photos uploaded to S3
v3	DynamoDB	titles and tags; search by tag
v4	Lambda	thumbnails created on upload
v5	Step Functions, Rekognition	workflow; automatic tags
v6	SQS	job queue; workers, retries
v7	Bedrock	captions written by an LLM (demo)

Plus: auto-scaling (ALB + Auto Scaling + load test)

Deploying Photogallery on EC2

Running Photogallery (`code/photogallery_v1`)

```
$ pip install -r requirements.txt   # Flask
$ PORT=5000 ./run.sh   # http://localhost:5000
$ ./run.sh             # port 80 (run.sh uses sudo)
```

`run.sh` simply runs:

```
$ python3 -m flask --app galleryApp run \
      --host 0.0.0.0 --port <port>
```

- Create a new EC2 instance to deploy the app
- Copy the app files from your PC using `scp`:

```
$ scp -i <key.pem> -r photogallery_v1 \
      ec2-user@<Public IP/Public DNS>:photogallery
```

Deploying Photogallery on EC2 (contd.)

- Connect via SSH to the instance, install the required software and start the application:

```
$ ssh -i <key.pem> ec2-user@<Public IP/Public DNS>
$ sudo dnf install python3-pip
$ cd photogallery/
$ sudo pip3 install -r requirements.txt
$ ./run.sh
```

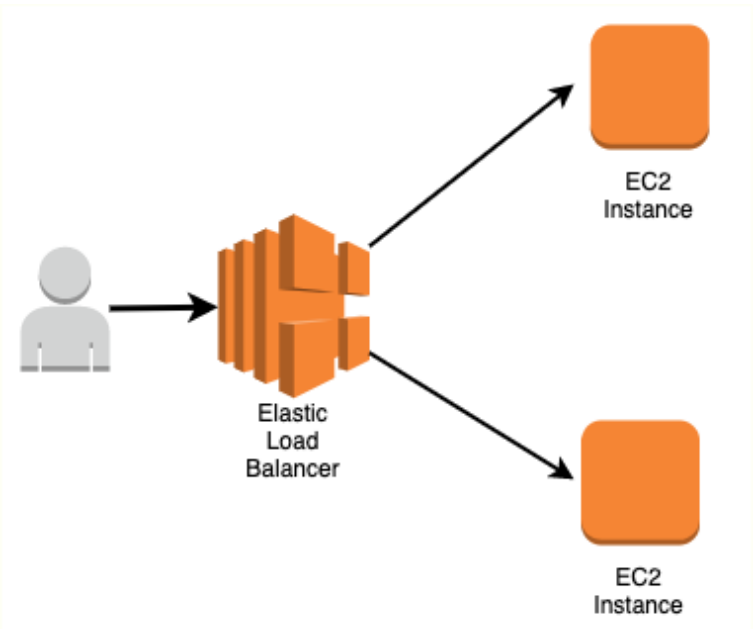
- Installing packages system-wide with pip is fine for this toy example; real deployments use a virtual environment
- `flask run` starts a development server: fine for testing, not for production (e.g., use `gunicorn` behind a proxy)

Deploying Photogallery on EC2 (contd.)

- Open `http://EC2-PUBLIC-IP/` in a browser
- The page footer shows the instance ID and the Availability Zone that served the request (read from the Instance Metadata Service, at `http://169.254.169.254/`, reachable only from the instance itself)
- Test: what if we "close" port 80 in the security group?

Replicating App Instances

- Current configuration is neither scalable nor fault-tolerant
- Let's run multiple replicas of the web server
- We need a **load balancer**
- The app is ready for this: `GET /health` answers `200 OK` (for the load balancer health checks), and the footer tells us which replica served each page



How Do We Prepare Each Replica?

- Every new instance must have the app installed and started at boot
- Doing it by hand (`scp` + `ssh`) for each replica does not scale

Two alternatives we will discuss:

1. Custom AMI

Set up one instance, then save its disk as a new **image**.
New instances boot with the app already installed.

2. User Data

Launch from the stock Amazon Linux image, plus a **script** that EC2 runs at first boot to install the app.

In both cases, the app must start automatically $\Rightarrow$ first, we turn it into a **service**

Starting the App at Boot

- We run the app as a systemd **service**, automatically started at boot

/etc/systemd/system/photogallery.service

```
[Unit]
Description=SDCC Photo Gallery
After=network-online.target
Wants=network-online.target
[Service]
Type=simple
WorkingDirectory=/home/ec2-user/photogallery
# Must be the same on every replica
Environment=FLASK_SECRET_KEY=sdcc-demo-key-change-me
ExecStart=/usr/bin/python3 -m flask --app galleryApp run --host 0.0.0.0 --port 80
Restart=on-failure
[Install]
WantedBy=multi-user.target
```

Starting the App at Boot (contd.)

Starting and enabling the service

```
$ sudo cp photogallery.service /etc/systemd/system/
$ sudo systemctl daemon-reload
$ sudo systemctl enable --now photogallery.service
```

The `photogallery.service` file is included in `photogallery_v1/`

Option 1: Custom AMI

- Start from the instance where the app is installed and the service is enabled
- *Actions* $\rightarrow$ *Image and templates* $\rightarrow$ *Create image*
- The new **AMI** is a copy of the instance's disk
- Launch new instances from this AMI: the application is already installed and configured to start

Note: each AMI is associated with a snapshot of the root EBS volume attached to the instance. Keeping this snapshot has a (small) cost: `https://repost.aws/knowledge-center/ebs-snapshot-billing`

Option 2: User Data

- A script that EC2 runs (as root) at the first boot of a new instance, launched from the stock Amazon Linux AMI
- Set it in *Advanced details* → *User data* when launching the instance (CLI: `-user-data file://user_data.sh`)

`photogallery_v1/user_data.sh` (excerpt)

```
#!/bin/bash
APP_URL=https://.../photogallery_v1.tar.gz
APP_DIR=/home/ec2-user/photogallery
dnf install -y python3-pip
mkdir -p $APP_DIR
curl -fsSL $APP_URL | tar -xz -C $APP_DIR --strip-components=1
chown -R ec2-user:ec2-user $APP_DIR
pip3 install -r $APP_DIR/requirements.txt
cp $APP_DIR/photogallery.service /etc/systemd/system/
systemctl daemon-reload
systemctl enable --now photogallery.service
```

Log of the script on the instance: `/var/log/cloud-init-output.log`

Custom AMI vs User Data

	Custom AMI	User Data
Setup done	once, before launch ("baked" image)	at every launch (first boot)
Boot time	fast: app ready right away	slower: install and download at boot
Updates	create a new AMI for each change	edit the script (or fetch latest code)
Reproducible	yes: identical image on every replica	depends on what is downloaded at boot
Dependencies at boot	none	package repos, app URL must be reachable
Cost	snapshot storage	none

In practice, often combined: a custom AMI with the heavy, stable parts (OS packages, runtime) + a small user data script for per-instance configuration

Later in the course: **Ansible**, a further alternative (configure running instances remotely)

Next Steps: Load Balancing, Auto Scaling

- So far, replicas are created by hand
- Next lecture: let AWS create (and remove) replicas for us
- A load balancer will route requests to replicas

Clean up!

- Terminate instances
- Deregister AMIs **and** delete their snapshots

Learner Lab

- *End Lab* stops the session but does not delete resources!
  - EC2 instances are stopped (and restarted with the next session)
  - Other resources (e.g., NAT gateways, RDS databases) keep running and consume your budget
- *Reset* deletes all resources, but does not restore the budget