ITI8801 · Practice 2 (AWS)

Two instances behind a load balancer, by hand

The AWS path. The Azure path is a separate deck; pick one cloud and follow it end to end.

Siim Rebane · TalTech · Autumn 2026

The task

Build this in the AWS console, by hand:

Virtual private cloud (VPC)	`iti8801-vpc`, `10.61.0.0/16`, one region, two availability zones
Subnets	2 public (for the load balancer), 2 private (for the instances); CIDRs assigned by the VPC wizard
Instances	`app-1`, `app-2`, t3.micro, Ubuntu Server LTS, one per private subnet, **no public IP, no key pair**
Security groups	`alb-sg`: TCP 80 from `0.0.0.0/0`. `app-sg`: TCP 80 from `alb-sg` only
Software	the course beacon on both instances, installed by a user-data script at first boot, port 80
Load balancer	Application Load Balancer (ALB) in the public subnets, HTTP 80 → target group of both instances, health check `/hello`
Outbound	none: the private route tables have no NAT gateway

Final state, and how it is proven

The ALB's DNS name is the only address that exists. Port 80 is the only port that answers. Every request lands on one of two instances that have no route to the internet.

The course validator sends requests to that name and checks five things:

- the name resolves to AWS addresses
- exactly two distinct beacons answer behind it
- neither beacon can reach the internet
- port 22 on the address does not answer
- both beacons have an address in `10.61.0.0/16`

All five green: the check is stored on the server with a code, that code goes to Moodle, and you delete everything.

Why the build opens a door and closes it again

The final state has no way in and no way out. A machine like that cannot be built directly: installing the software needs a way out, for `apt` and the Docker image pull.

On this path there is **no way in at any point**: no SSH, no key pair. The instance installs its own software at first boot from a script you give it, a **user-data script**, and pulls the packages through a **NAT gateway** that exists only for that. After the first boot the instance never needs the internet again.

So the build goes: NAT gateway first, launch the instances with the script, wait for them to install themselves, then delete the gateway. The validator looks at the closed state only; having had a NAT gateway during the build costs nothing on the check.

The plan on one slide

Phase	What	Why
1	VPC with the wizard: subnets, route tables, internet gateway, **NAT gateway**	the walls and the temporary way out
2	2 security groups, 2 instances with a **user-data script**	installation happens at boot
3	Confirm the scripts ran	the software
4	Target group + Application Load Balancer	the single entry point; watch it balance
5	**Lock down**: delete the NAT gateway, release its address	the assignment state
6	Validator check → **green**, copy its code	proof
7	**Delete everything**, in order	it bills hourly

Instance size: **t3.micro** (free tier). Region: pick one and stay in it; every page in the console is per region.

Phase 1 — the VPC

VPC → Create VPC → Resources to create: **VPC and more**. The form, top to bottom:

- *Name tag auto-generation*: *Auto-generate* ticked, value `iti8801`. It is a prefix: the wizard names everything it creates after it, `iti8801-vpc`, `iti8801-subnet-private1-<zone>`, `iti8801-rtb-public`
- *IPv4 CIDR block* `10.61.0.0/16` (CIDR: the `/16` range notation), *IPv6 CIDR block* **No IPv6 CIDR block**, *Tenancy* Default
- *Number of Availability Zones (AZs)* **2**, *public subnets* **2**, *private subnets* **2**. Leave *Customize subnets CIDR blocks* closed; the wizard picks the subnet ranges
- *NAT gateways* **Zonal**, in one zone. *Regional* is a newer multi-zone product; more capacity than today needs, and it bills more
- *VPC endpoints* **None**. The pre-selected *S3 Gateway* is a private path to AWS's storage service; nothing today uses it
- *DNS options*: both ticked, the defaults. Instances get names, and names resolve inside the VPC

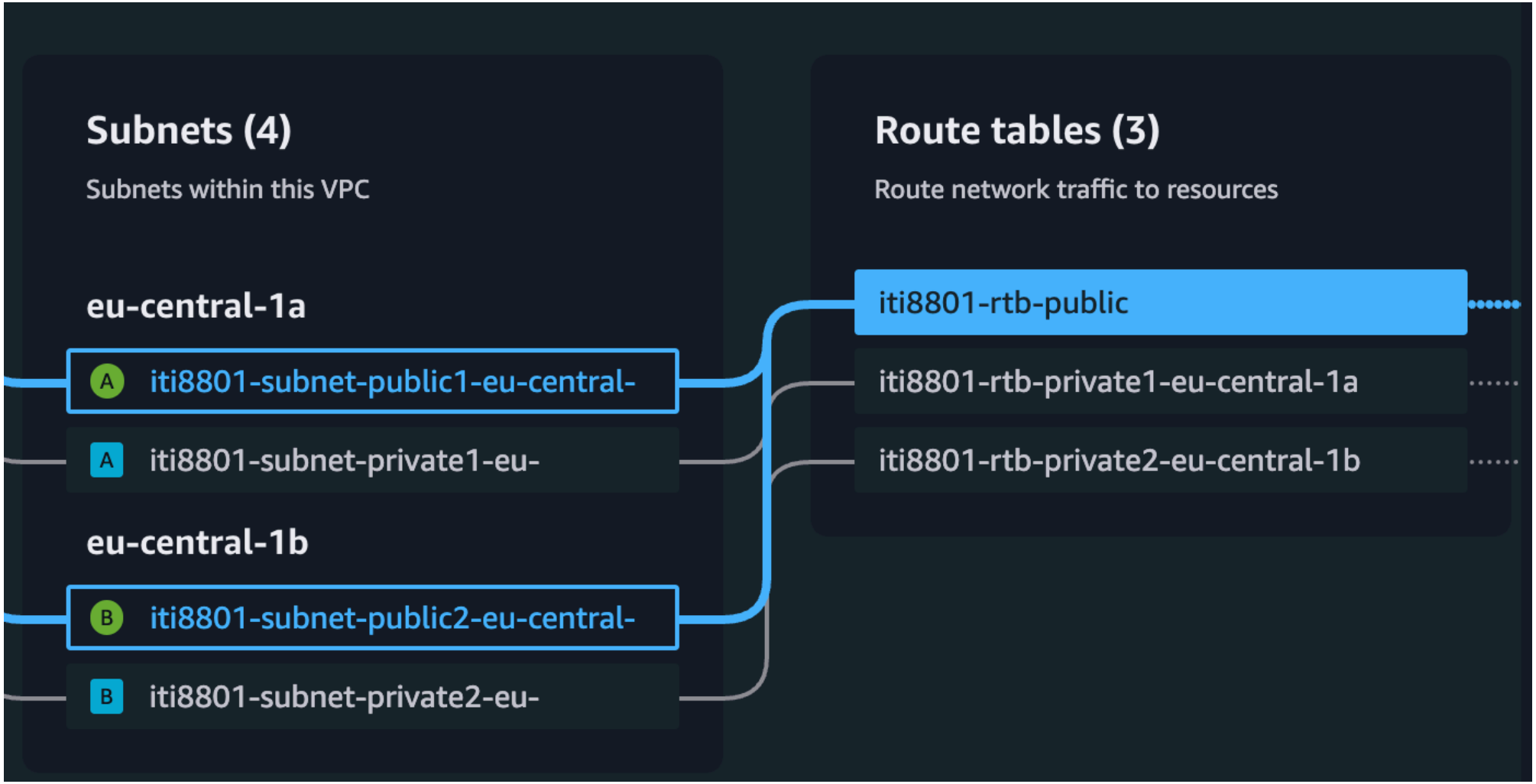
Phase 1 — the preview

The right half of the form is a live preview of what *Create VPC* will make, redrawn at every change. With the settings above it shows:

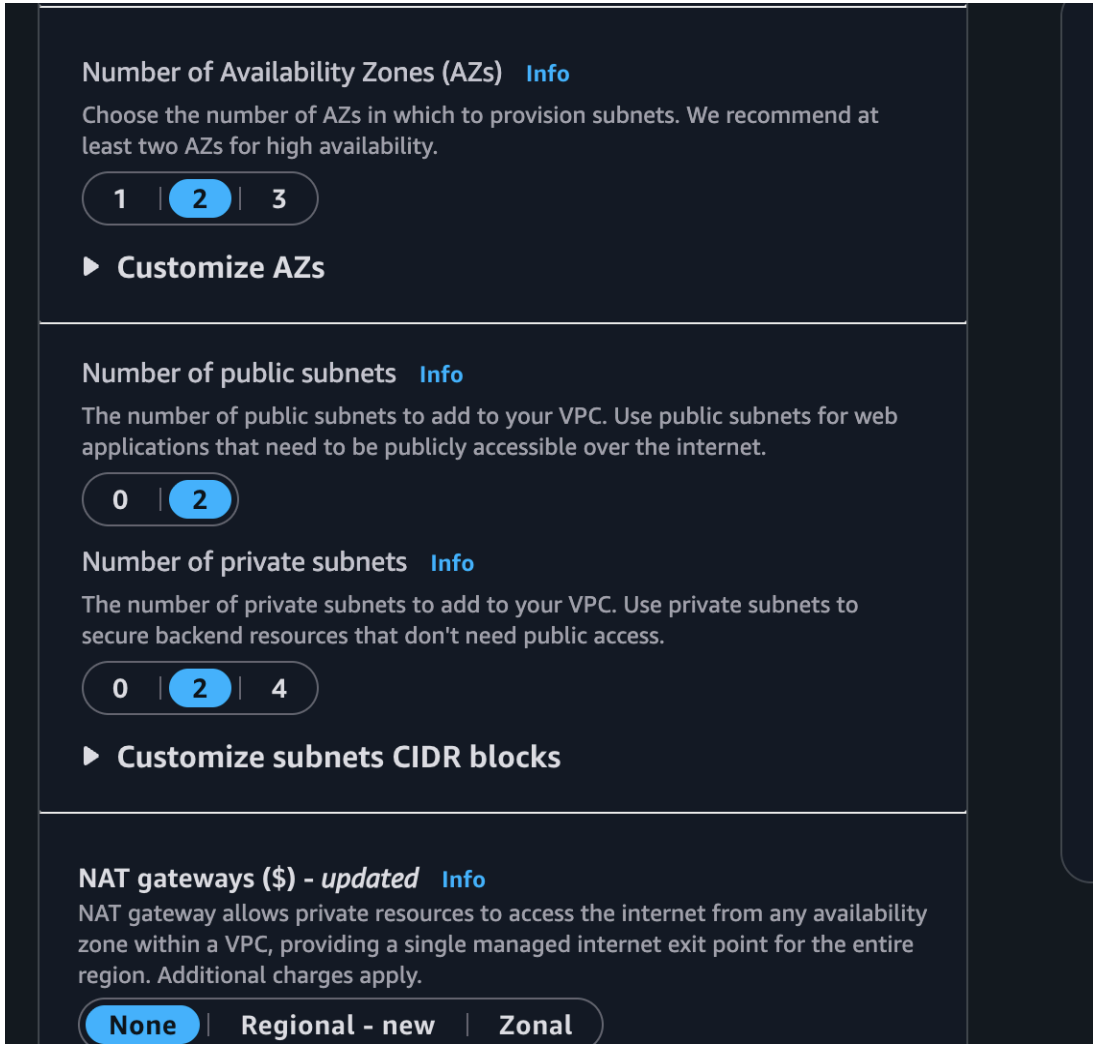
- **Subnets (4)**: per zone one public and one private, named `iti8801-subnet-public1-<zone>`, `iti8801-subnet-private1-<zone>`
- **Route tables (3)**: one `iti8801-rtb-public` shared by both public subnets, and one table per private subnet, `iti8801-rtb-private1-<zone>`. Each private subnet gets its own table because its way out could differ per zone
- **Network connections**: `iti8801-igw`, the internet gateway, and the NAT gateway. If `iti8801-vpce-s3` is listed, *VPC endpoints* is still on *S3 Gateway*

Hover a box and the lines show what is connected to what. When the preview matches, *Create VPC*.

Phase 1 — the preview, subnets to route tables



Phase 1 — the form: zones, subnets, NAT



Two zones because the ALB insists on two, and because a zone is a datacenter that can fail on its own.

Two public subnets, one per zone, for the load balancer's nodes. Two private subnets, one per zone, so the two instances sit in different zones too.

The NAT gateway holds a reserved public address, an **Elastic IP**, and bills from this moment. It is deleted in phase 5.

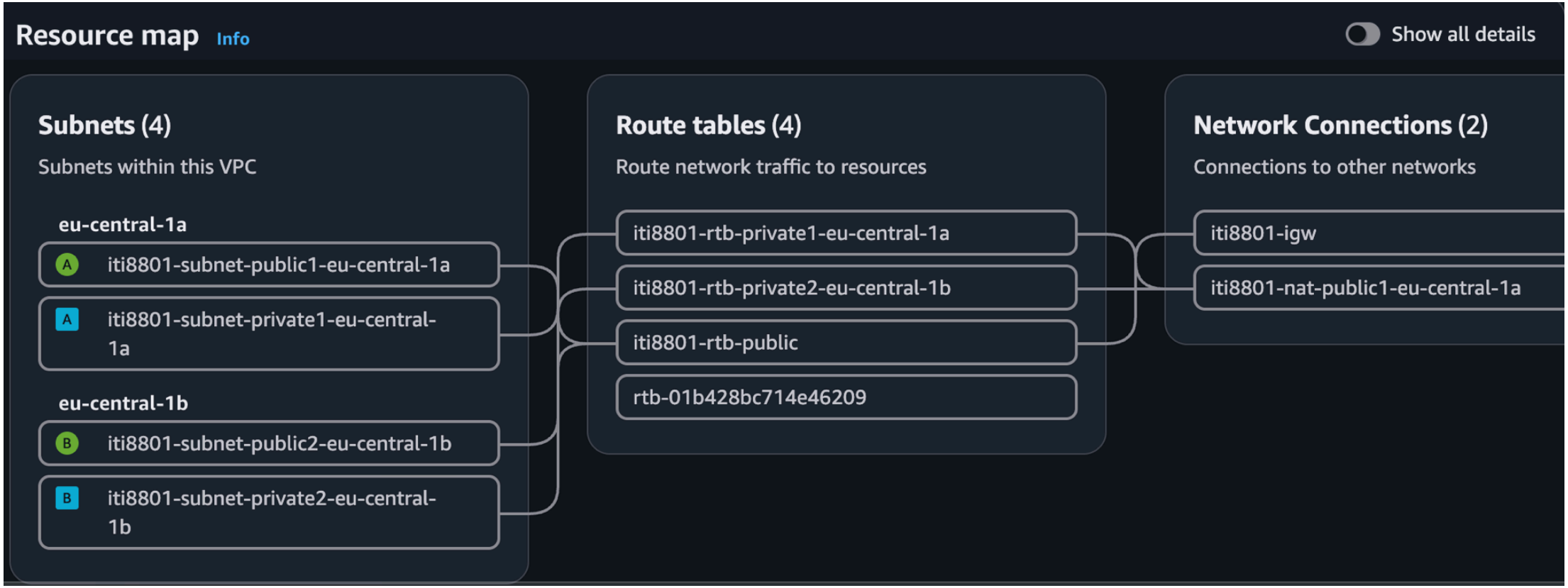
Phase 1 — check yourself

VPC → *Your VPCs* → `iti8801-vpc` → the *Resource map* tab. Lines connect what belongs together:

- Both public subnets → `iti8801-rtb-public` → `iti8801-igw`
- Each private subnet → its own `iti8801-rtb-private…` → `iti8801-nat-public1-…`, the NAT gateway. Its name says where it lives: in the first public subnet, because it needs the internet gateway itself
- A fourth route table with no name, `rtb-0…`: the VPC's **main route table**, created with every VPC. All four subnets have explicit tables, so it is attached to nothing and does nothing. Leave it
- *Show all details* (top right) lists each table's routes: `0.0.0.0/0 → igw-…` in the public one, `0.0.0.0/0 → nat-…` in the private ones

Subnets in the left menu: the wizard's ranges are `10.61.0.0/20` and `10.61.16.0/20` for the public subnets, `10.61.128.0/20` and `10.61.144.0/20` for the private ones. On every subnet *Auto-assign public IPv4 address* is *No*, the public ones included: nothing in this VPC gets a public address unless asked for one.

Phase 1 — the resource map



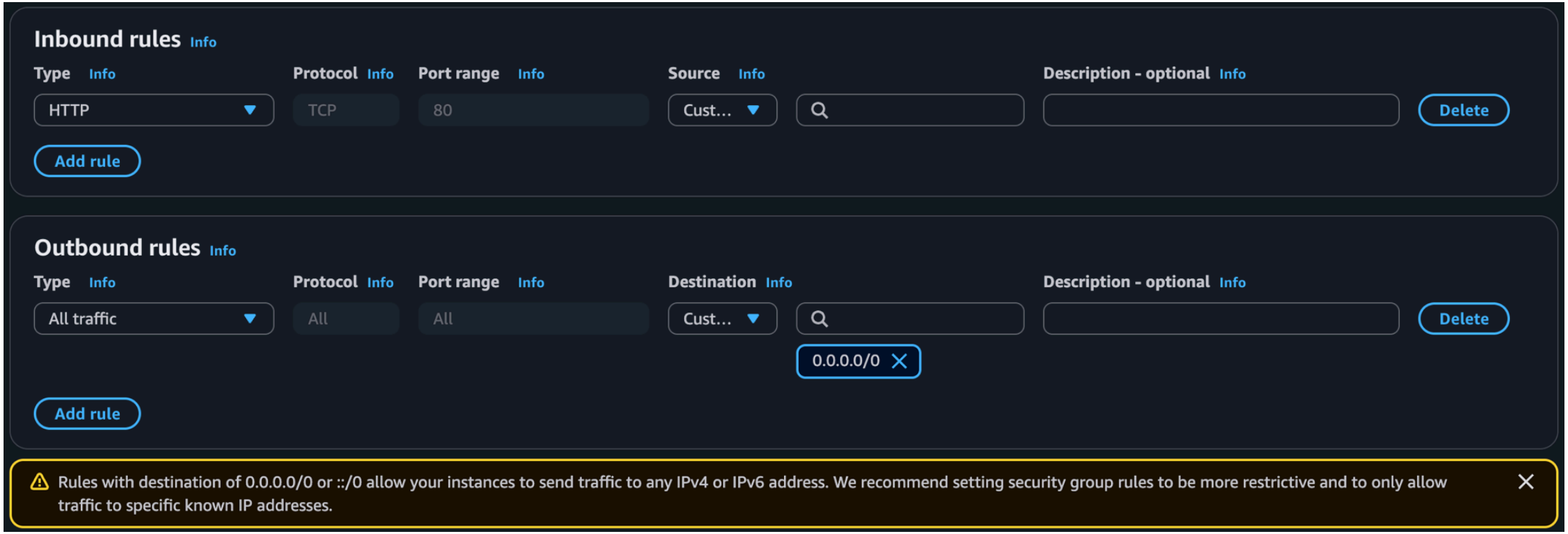
Phase 2 — the security groups

EC2 → *Security groups* → *Create security group*, twice. *Security group name*, *Description* (required; anything), *VPC* `iti8801-vpc` : the dropdown opens on the account's *default VPC*, `vpc-…` with no name, and a group made there cannot be attached to anything in yours. Then one inbound rule each:

Group	Inbound rule: *Type*	*Source*
`alb-sg`	**HTTP** (fills TCP, 80)	**Anywhere-IPv4**, which fills `0.0.0.0/0`
`app-sg`	**HTTP**	**Custom**, and in the search box type `alb-sg` , pick the group. The box then shows its ID, `sg-…` , not the name

Outbound rules stay at the default, *All traffic* to `0.0.0.0/0` , and the yellow warning about it can be ignored: outbound on this path is decided by the route tables, not by the group, and the install at boot needs the way out. Create `alb-sg` first, so it exists when `app-sg` asks for it.

Phase 2 — the rules part of the form



Type HTTP fills protocol and port. *Source* is the dropdown plus the search box next to it: *Anywhere-IPv4* fills the box with `0.0.0.0/0`; *Custom* leaves it for you to type a range or a group name.

Phase 2 — a group as the source

`app-sg` allows port 80 from `alb-sg`, not from an address. A rule whose source is another security group means "from any machine that is a member of that group". The load balancer's nodes are members of `alb-sg`, so the instances accept port 80 from the load balancer, whatever addresses its nodes use now or after AWS replaces them.

Nothing else is open on the instances: AWS security groups deny by default, and there is no SSH rule because there will be no SSH.

Phase 2 — the user-data script

This runs once, at first boot, as root. The same script for both instances:

```
#!/bin/bash
apt-get update
apt-get install -y docker.io
docker run -d --name beacon --restart=always --network host ghcr.io/siimrebane/iti8801-beacon:latest
```

Three things happen: the package list is refreshed and Docker installed through the NAT gateway, then the beacon image is pulled the same way and started on the instance's own network (`--network host`), so it sees the real interface and listens on port 80 directly. After that the instance never needs the internet again, which is what phase 5 relies on.

Phase 2 — launch the instances

EC2 → *Instances* → *Launch instances*, twice. Top to bottom:

- *Name* `app-1` / `app-2`
- *Application and OS Images*: **Ubuntu**. The version the console preselects, the current Ubuntu Server LTS, is the one to use
- *Instance type* **t3.micro**, marked *Free tier eligible*
- *Key pair*: leave it empty. *Launch instance* then asks once more; answer **Proceed without key pair**. Nothing will log in

Phase 2 — launch: the network part

- *Network settings*: as it opens it says the account's default VPC, a public IP *Enable*d, and a new security group `launch-wizard-…` with **SSH from anywhere** ticked. All three are wrong for today. Press **Edit**: *VPC* `iti8801-vpc`, *Subnet* `iti8801-subnet-private1-…` for `app-1` and `…private2-…` for `app-2`. *Auto-assign public IP* switches to **Disable** by itself, since the subnet does not hand them out; check that it did. *Firewall*: **Select existing security group** → `app-sg`
- *Advanced network configuration*: nothing to change. AWS picks a free address in the subnet; the validator only requires it to be inside `10.61.0.0/16`, and every address in your subnets is
- *Advanced details*, near its bottom: *User data*, a text box. Paste the script. The box wraps long lines for display; the `docker run` command must still be one line, so after pasting put the cursor after `host` and press End: it should land after `latest`

Two instances because a load balancer with one machine behind it balances nothing; with two you will see requests alternate in phase 4.

Phase 3 — did the script run?

You cannot log in, so you read what the instance reports: *Instances* → the instance → *Actions* → *Monitor and troubleshoot* → **Get system log**. It is the console output of the boot, long; the interesting part is the last screen. About a minute after launch it contains, in this order:

- a `ci-info` table with the instance's address, `ens5 … 10.61.138.130`, and its routes: `0.0.0.0` via the subnet's gateway `.1`, and one line for `10.61.0.2`, the VPC resolver
- `Cloud-init … running 'modules:final'`: your script starts here
- `Get: … docker.io …`, then `Setting up docker.io`: the package came in through the NAT gateway
- `Pulling from siimrebane/iti8801-beacon` … `Status: Downloaded newer image`, and a long container id: the beacon is running
- `Cloud-init … finished`

Phase 3 — what the log may also say

One line to ignore: `SSM Agent unable to acquire credentials … AccessDeniedException`. That is AWS's remote-management agent, part of the image, finding no permission to phone home. Nothing today uses it.

If the `Get:` lines fail or hang, the way out is missing: the NAT gateway or the private route table (phase 1). If nothing after `modules:final` mentions docker, the user data was not attached: terminate and launch again.

Phase 4 — what a health check is

A load balancer has a list of targets and no idea which of them work. So it asks, all the time: every few seconds it sends each target one request of its own and looks at the answer.

- **TCP check**: opens a connection to the port. Proves that something listens, nothing more
- **HTTP check**: sends `GET <path>` and requires status **200**. Proves the application itself answers. Today's path is `/hello`, the beacon's own endpoint

A target that fails the check *unhealthy threshold* times in a row gets no new requests until it passes *healthy threshold* times in a row. The target group's defaults: every **30 s**, unhealthy after **2** failures, healthy after **5** passes. So a fresh target is *healthy* about 2.5 minutes after it starts answering, and a dead one is dropped after about a minute. Under *Advanced health check settings*, interval 10 and healthy threshold 2 cut the wait to 20 seconds.

The checks come from the ALB itself, which is why `app-sg` allows port 80 from `alb-sg`: without that rule every target would be unhealthy and the ALB would forward nothing.

Phase 4 — the target group

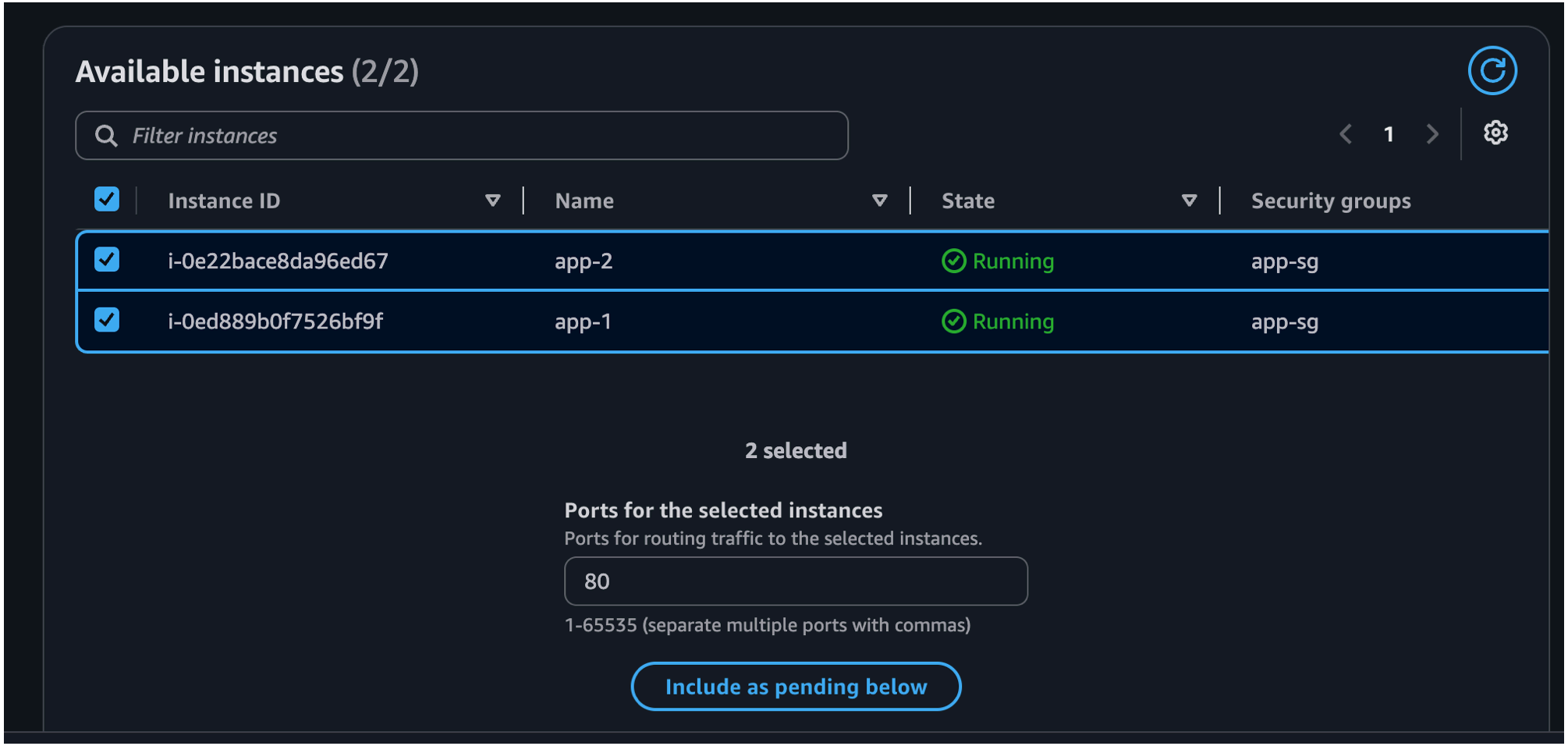
A **target group** is the list of machines a load balancer sends to, with the health check that decides which of them count. It exists on its own; the load balancer refers to it.

EC2 → *Target groups* → *Create target group*. Page 1, top to bottom:

- *Target type* **Instances**; name `app-tg`; *Protocol* **HTTP**, *Port* **80**; *IP address type* IPv4
- *VPC* `iti8801-vpc`; *Protocol version* HTTP1
- *Health checks*: protocol **HTTP**, *Health check path* `/hello`. *Advanced health check settings* stays folded unless you want the shorter wait: interval 10, healthy threshold 2
- *Target optimizer* Off, *Tags* empty → *Next*

Page 2, *Register targets*: tick `app-1` and `app-2`, port 80 as offered, **Include as pending below** → *Next*. Page 3, *Review targets*, lists both with their zone and private address → *Next*, and create.

Phase 4 — register targets

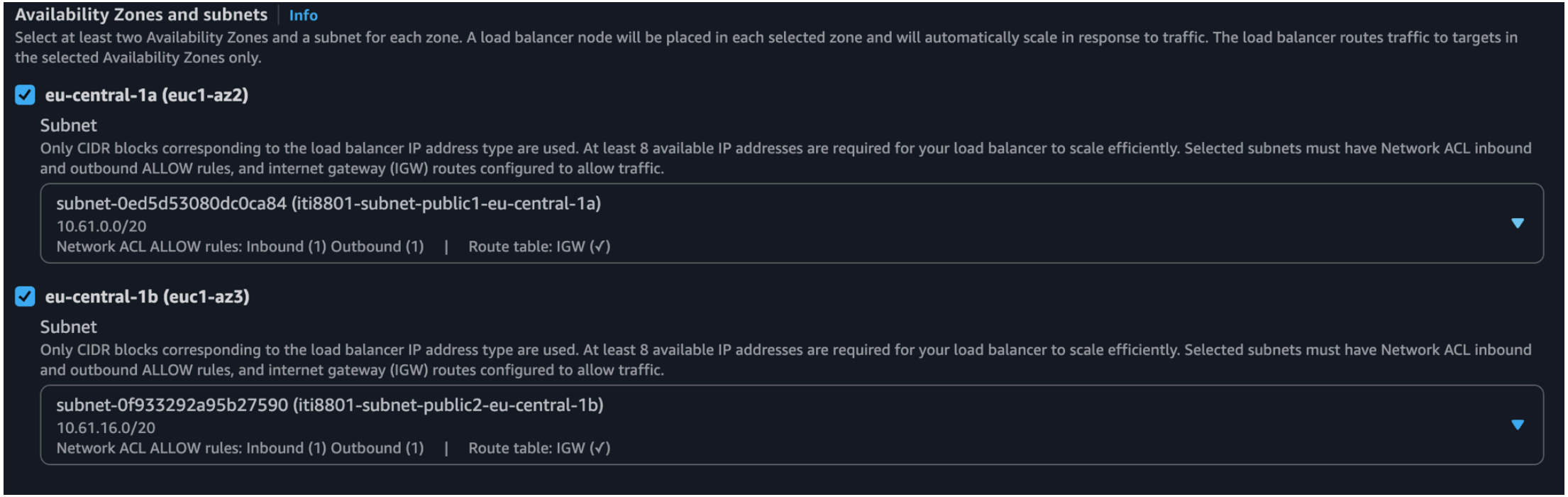


Phase 4 — the ALB

Load balancers → *Create load balancer* → **Application Load Balancer** → *Create*. Top to bottom:

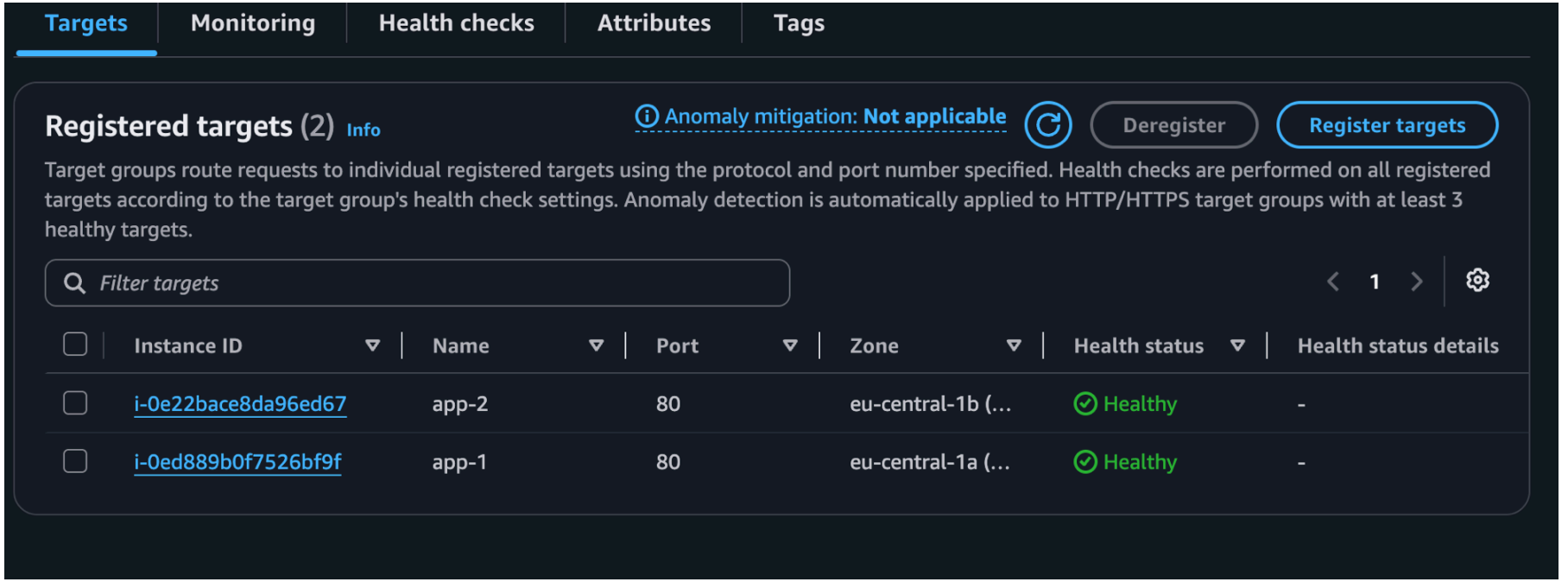
- *Basic configuration*: name `app-lb`, *Scheme* **Internet-facing** (the other choice, *Internal*, gets private addresses only), *IP address type* IPv4
- *Network mapping*: VPC `iti8801-vpc`; *IP pools* unticked; under *Availability Zones and subnets* tick **both** zones, and in the dropdown each one reveals pick the **public** subnet of that zone, `iti8801-subnet-public1-…` and `…public2-…`. The dropdown says which is which: a public subnet shows *Route table: IGW* ✓. The form refuses fewer than two zones
- *Security groups*: `alb-sg` only; remove the *default* group the form pre-fills
- *Listeners and routing*: one listener, HTTP **80**; *Default action* → *Routing action* **Forward to target groups**, *Target group* `app-tg`, weight 1. Everything below (attributes, tags) stays at its defaults
- *Create load balancer*. State *Provisioning* for a minute or two, then *Active*

Phase 4 — the ALB's zones and subnets



Phase 4 — both targets healthy

`app-tg` → *Targets*: wait for both to show **Healthy**. With the default thresholds that is a few minutes after the beacons start.



Phase 4 — the first request through the ALB

The ALB's overview shows its **DNS name**, `app-lb-….eu-central-1.elb.amazonaws.com`. An ALB has no fixed IP: the name resolves to one address per zone, and those can change. From your laptop:

```
curl http://<ALB-DNS-NAME>/hello
```

That name is also what you enter on the validator.

Phase 4 — watch it balance

Ask the load balancer ten times and look only at which beacon answered:

```
for i in $(seq 10); do curl -s http://<ALB-DNS-NAME>/hello | grep -o '"beacon_id":"[^"]*"'; done
```

Two different IDs, in no fixed order. The ALB is two nodes, one per zone, and its DNS name resolves to both; each node hands its own requests to the targets in turn, and your laptop picks a node per connection. Over ten requests both IDs appear, about half each.

Now the health check. You cannot stop a beacon by hand here (no SSH), so use the target group instead: *Targets* → select `app-1` → **Deregister**. Within a moment the loop shows only `app-2`'s ID. Register it again and both are back. When a real target stops answering, the health check does the same deregistering on its own, and that is what keeps a service up while one machine is down.

Phase 4 — read what the beacon says

Nobody can log in, so the beacon's answer is your only view of the machine from inside. Ask for it through the load balancer, from your laptop:

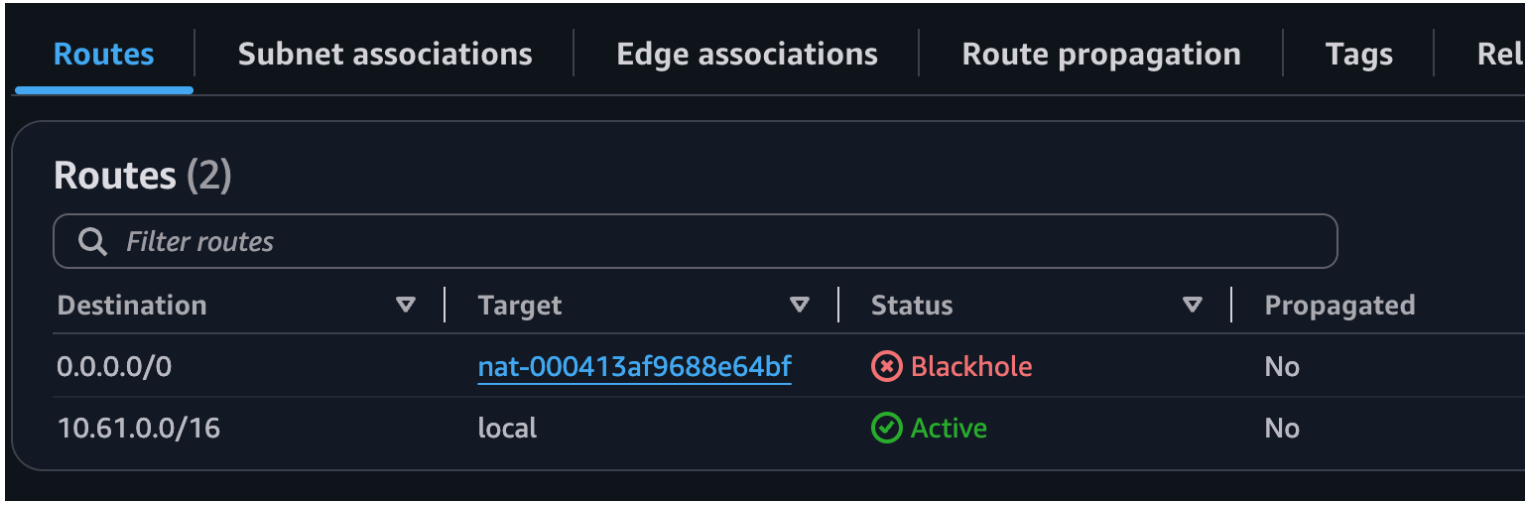
```
curl http://<ALB-DNS-NAME>/hello
```

One instance's JSON. Right now it should show:

- `interfaces`: `ens5`, the instance's network card, with the private IP AWS gave it; and `docker0`, Docker's own bridge. No public IP anywhere
- `hostname`: `ip-10-61-138-130` or similar. AWS names the machine after its private address; the *Name* tag `app-1` is a label in the console, not the hostname
- `outbound`: the two TCP targets **succeed**, expected; the NAT gateway is still there. `dns:example.com` succeeds too, and keeps succeeding after the lock-down: the VPC's resolver at its second address (`10.61.0.2`) is a platform service, not the internet, and the validator ignores it

Phase 5 — remove the temporary way out

1. *VPC* → *NAT gateways* → select → *Actions* → **Delete**. Wait for the state *Deleted*
2. *Elastic IPs* → `iti8801-eip-…`, the address the NAT gateway used → *Actions* → **Release Elastic IP addresses**. Deleting the gateway never releases its address, and a reserved public address bills while you hold it, used or not. Two more addresses with no name are listed here: the ALB's own, one per zone. Leave them; they go with the ALB
3. *Route tables* → a private one → *Routes*: `0.0.0.0/0 → nat-…` now in state **Blackhole**, a route that leads nowhere. That is the lock



Phase 6 — the check

Course validator: **https://iti8801.digivader.com/lab2-validator/**

1. Enter your student code + `<ALB-DNS-NAME>` (the name only, no `http://`)
2. The server checks, through your ALB:

Check	What it proves
ALB address resolves to AWS	it is a real cloud load balancer
Many requests reach **exactly two** beacons	the ALB leads to your two instances, both healthy
Beacons: no outbound anywhere	the private subnets have no way out
Port 22 on the ALB address does not answer	no SSH through the entry point
Every beacon has an address in `10.61.0.0/16`	the instances are in the VPC this deck builds

3. Five green boxes → **passed**. Under them is the **check code**, a long random id: copy it, that code is your submission

Phase 6 — not green?

Each box maps to one phase of today: the address to phase 4, the two machines to phases 2 to 4, the way out to phase 5, the private range to phases 1 and 2. Fix, press *Check again*.

Every check is stored with its own code and nothing is ever overwritten, so check as often as you like and submit the code of a green one.

Phase 7 — delete everything, in this order

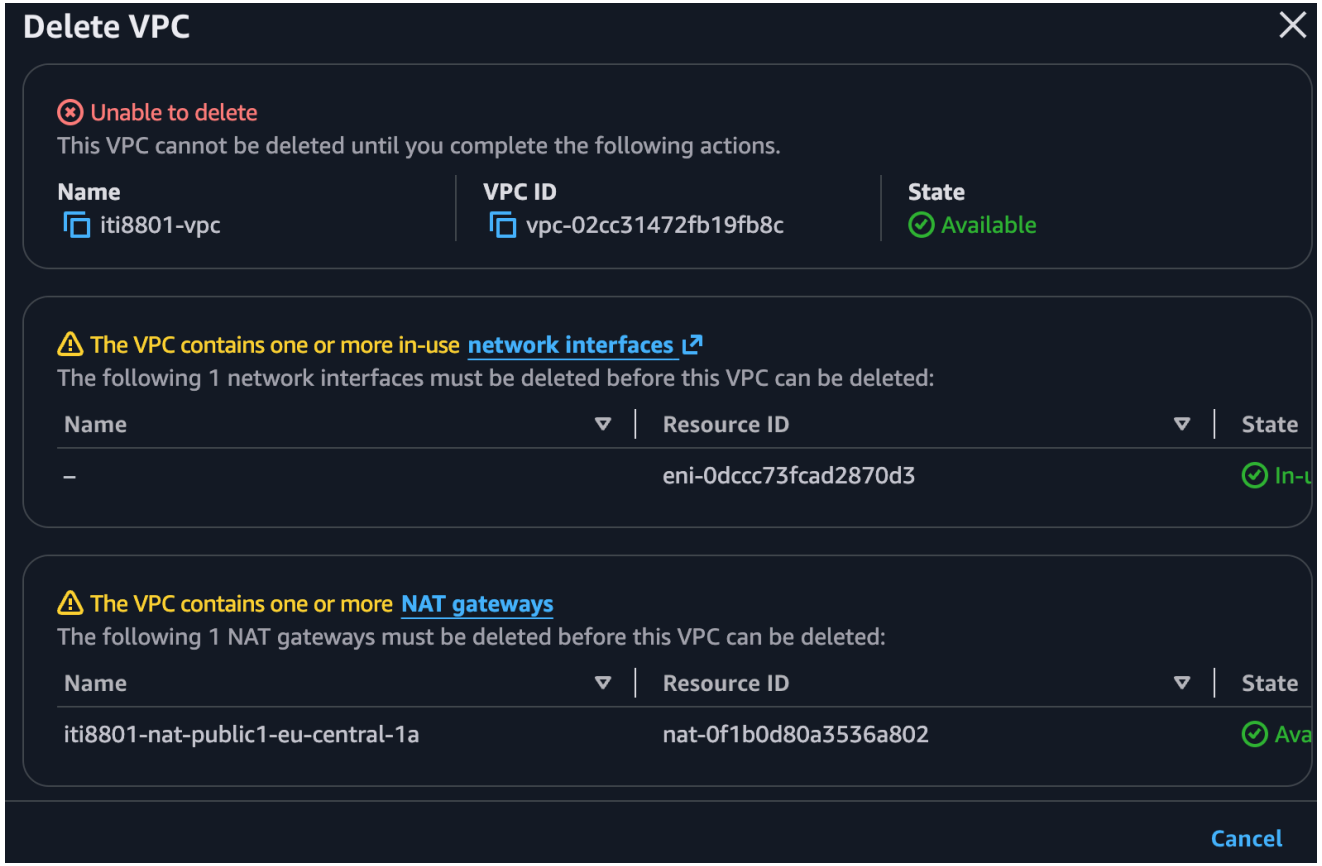
Everything you built bills by the hour, and your green check is already saved.

1. *Instances* → terminate both
2. *Load balancers* → delete the ALB, then *Target groups* → delete
3. *VPC* → `iti8801-vpc` → **Delete VPC**. This removes the subnets, route tables, internet gateway and security groups with it. If it refuses, something above is still there
4. *Elastic IPs*: must be empty. The ALB's two addresses disappear with the ALB; anything still listed is yours and bills until released

Check *Billing* → *Cost Explorer* tomorrow: the line should be flat again.

Phase 7 — when Delete VPC refuses

The dialog names what is still in the way. A NAT gateway must be deleted first and takes a minute or two to disappear; the network interface listed with it is the gateway's own and goes with it. Retry when the gateway's state reads *Deleted*.



Summary — what you built

By hand, in ~90 minutes:

VPC → subnets and routes → security groups → 2 instances that set themselves up → ALB over the two → lock-down

Keep today's notes. Every step you did by hand created one named object, and that list of objects is the whole setup.

Done when

- A **green check** on the validator (stored on the server; a screenshot is optional)
- **Everything deleted**, cost page flat, no Elastic IP left
- A2 submission in Moodle: **the check code** of a green check, nothing else

Deadline: **Sunday of week 3, 23:59.**