ITI8801 · Cloud Architectures and DevOps

Networks and firewalls

Siim Rebane · TalTech · Autumn 2026

Today

1. Networking foundations: IP addresses, public and private, routing, NAT (network address translation), subnets
2. The same concepts in the cloud: your own network, subnets, the way out — and where Azure and AWS differ
3. Firewalls in the cloud
4. The load balancer
5. The two clouds side by side, and the exact products you set up this week
6. What sits in front in real deployments: the edge and the CDN
7. Costs and budget alarms
8. Assignment 2

What an IP address is

An IPv4 address is **32 bits** (4 bytes), written as four numbers of one byte each:

```
10.60.1.10   =   00001010 . 00111100 . 00000001 . 00001010
```

Every machine that talks over a network has at least one. The address is what goes into a packet's *to* and *from* fields. Everything in this lecture is, in the end, about who can send a packet to which address.

Public and private addresses

Public addresses must be unique on the whole internet, so they are handed out centrally: IANA (the internet's address authority) → regional registries → your internet provider → you. You cannot pick your own.

Three ranges are set aside for use inside local networks:

Range	Size
`10.0.0.0` – `10.255.255.255`	~16.8 million addresses
`172.16.0.0` – `172.31.255.255`	~1 million addresses
`192.168.0.0` – `192.168.255.255`	65 536 addresses

Internet routers are configured to drop packets addressed to these; they work only inside a network. Because they never travel the internet, the same range can be in use everywhere at once: your home network is `192.168.1.x`, so is your neighbor's, and nothing collides.

Networks, and the two parts of an address

Routers do not keep a list of every machine on the internet; that would be billions of entries. Delivery happens in two steps: a packet first travels to the destination's **network**, and that network delivers it to the **machine**.

For this to work, every address is read as two parts:

```
address:         10 . 60 .  1 . 10       (in a network whose split is after 16 bits)
network part:    10 . 60                ← identifies the network; same for every machine in it
host part:                  1 . 10      ← identifies one machine inside that network
```

Where the split sits is part of each network's definition, and differs from network to network.

CIDR notation

This way of writing networks is called **CIDR notation** (classless inter-domain routing). The split position is written after the address: `/` plus the number of network bits.

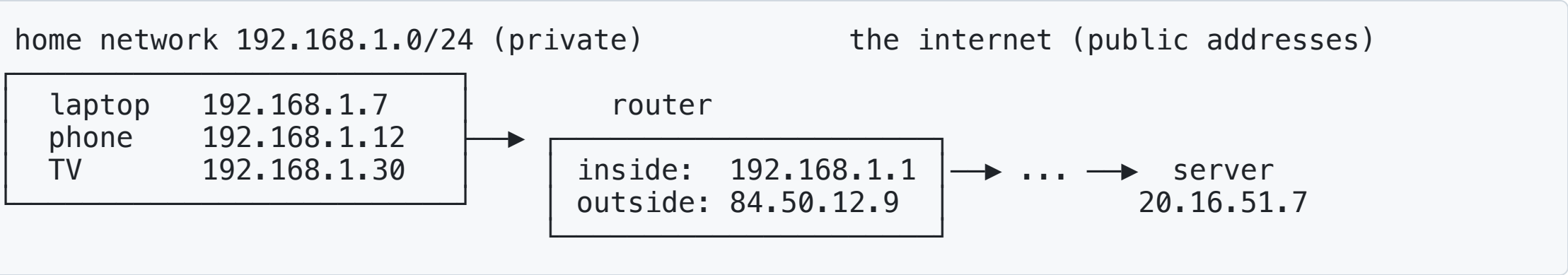
Notation	Network part	Host part	Addresses
`10.60.0.0/16`	`10.60` (16 bits)	16 bits	65 536
`10.60.1.0/24`	`10.60.1` (24 bits)	8 bits	256
`10.60.1.10/32`	all 32 bits	none	exactly 1 — this one machine
`0.0.0.0/0`	no bits fixed	all 32 bits	every address that exists

A larger number after the slash fixes more bits, so the network is smaller. Two edge cases are used constantly: `/32` in a firewall rule means this one machine only; `0.0.0.0/0` in a rule or route means anywhere. Every cloud form that asks for an address range expects this notation.

The private ranges from before, written in CIDR form: `10.0.0.0/8`, `172.16.0.0/12`, `192.168.0.0/16`.

The setup we will follow a packet through

A home network and a server on the internet. Every device at home has a private address; the router is the only machine with a public one:



The router stands with one foot in each world: a private address on the home side, a public address on the internet side. The whole home shares that single public address.

Following a packet, step 1: leaving the laptop

Your laptop (`192.168.1.7`) sends a request to the server `20.16.51.7` :

1. The laptop checks: is the destination inside my own network, `192.168.1.0/24` ? It is not, so it cannot deliver directly.
2. Everything non-local is handed to the **router**, which the laptop knows as its *default gateway*.

The route table

Every router holds a **route table**: a list of lines, each saying *packets for this destination range go that way*. A packet is matched against the lines and forwarded accordingly. Our home router's table:

```
192.168.1.0/24  → deliver locally
0.0.0.0/0       → internet provider
```

`0.0.0.0/0` covers every address that exists: it is the line that matches when no other line does. Our packet matches it and leaves toward the internet.

A route table decides only where packets *can go*. If no line matches, the packet is dropped; there is nowhere to send it.

Following a packet, step 2: NAT at the router

The packet has a problem: its **sender** address is `192.168.1.7`, a private address. A reply addressed to it could never travel back across the internet.

The router solves this with **NAT** (network address translation): it rewrites the sender.

```
before:   from 192.168.1.7    to 20.16.51.7
after:    from 84.50.12.9     to 20.16.51.7      (84.50.12.9 = the router's public address)
```

The router remembers the substitution. When the reply arrives for `84.50.12.9`, it looks the conversation up and hands the reply to `192.168.1.7`.

The asymmetry matters: machines behind NAT can start conversations **out**, but nothing on the internet can start a conversation **in**. An unknown incoming packet matches no remembered conversation, so it is dropped. Clouds sell this exact function as a **NAT gateway**.

Following a packet, step 3: across the internet

Between your router and the server, the packet crosses a chain of internet routers. Every one of them does the same single operation: compare the destination against its route table, forward the packet one step closer to the network `20.16.0.0/16`.

```
laptop → home router → provider → ... internet routers ... → server's network → server
        (NAT here)                  (each: route table lookup)  (delivers to host .51.7)
```

The server's reply makes the same trip in reverse. No router on the path knows the whole route; each one knows only its own next step.

Subnets

A network's owner can divide it into **subnets**: smaller networks inside it. The router that connects them can apply **a different route table to each subnet** — the rules for packets leaving that subnet's machines. This is the control mechanism the whole lecture is built on:

Each subnet's route table either contains a `0.0.0.0/0` line or it does not:

- **With the line**: machines in that subnet can send packets to the internet
- **Without it**: a packet addressed to the internet matches no line and is dropped. Replies would need the same path back, so the internet cannot reach the subnet either. A missing route cuts the connection in both directions

These are all the networking facts this lecture needs. The rest of the lecture applies them, using the cloud vendors' names for them.

Your own network in the cloud

Azure calls it a **VNet** (virtual network), AWS calls it a **VPC** (virtual private cloud). Both mean the same thing: a private network that belongs to you, with an address space you pick, divided into subnets you define.

```
VNet 10.60.0.0/16
├── subnet "app"  10.60.1.0/24    ← things the load balancer (LB) talks to
└── subnet "db"   10.60.2.0/24    ← things only the app talks to
```

- Machines inside reach each other over private IPs, with no internet involved
- There is no router machine to see: **the cloud itself is the router** between subnets, and it applies a route table to each subnet
- Nothing in your VNet is reachable from outside unless *you* create a path

The **load balancer** is a service that receives requests on a public address and forwards them to machines inside. It is the one piece of this setup that will face the internet.

Routing in the cloud: AWS

The subnet's route table is the control mechanism. On AWS you hold it directly:

- Every subnet has a route table you attach. A new VPC's table has one line, `10.61.0.0/16 → local`: packets for the VPC's own addresses are delivered inside it. No other line exists, so a packet for any other address matches nothing and is dropped
- Internet exists only when you create an **internet gateway** (the VPC's router connection to the internet) and add `0.0.0.0/0 → internet gateway` to a subnet's table
- A private subnet is one where that line is missing

A new VPC starts cut off. Every way out is a line you added.

Internet gateway and NAT gateway: two different doors

Both are ways out of a VPC, and every AWS setup names both. They do different things:

- **Internet gateway**: the VPC's connection to the internet, one per VPC. It works in **both directions**, but only for a machine that has a **public address of its own**: it maps that public address to the machine's private one and back. A machine without a public address gets nothing from it
- **NAT gateway**: one public address **shared** by a whole subnet, **out only**, the home router's job sold as a service. It translates the sender of every packet from the private subnet to its own address and remembers the conversation; nothing on the internet can start a connection in. It lives in a public subnet and uses the internet gateway itself

In a route table: a public subnet's line reads `0.0.0.0/0 → internet gateway`, a private subnet's `0.0.0.0/0 → NAT gateway`. Delete the NAT gateway and the private subnet has no way out; the public subnet, and the load balancer in it, keep theirs.

Azure has no internet gateway object: that role is built into every VNet. Its NAT gateway is the same product as AWS's.

Routing in the cloud: Azure

Same mechanism, the opposite default:

- Every subnet gets a built-in table (*system routes*) with the same local line, `10.60.0.0/16 → VNet`, and **already** a second one: `0.0.0.0/0 → Internet`
- That line cannot be removed. You attach a table of your own only to override lines
- There is nothing to create: the route out already exists. Whether a VM can actually reach the internet depends on whether it has a **public address to leave with**

In AWS a subnet starts cut off and you add the way out; in Azure the way out is there from the start and what you control is the address.

The way out on Azure: a public address is required

A packet from a private address can only leave if something on the way translates its sender to a public address. On Azure the route out exists, but the translation happens only if the VM has one of:

- a **public IP** attached to its own network card
- a **NAT gateway** attached to its subnet
- a **load balancer** that is allowed to translate its outbound traffic

A VM with none of these sends the packet along the route, reaches the edge of the VNet, and the packet is dropped: there is no public address to send it as. Nothing can reach the VM either, since it has no public address to be reached at.

One catch: unless a subnet is marked **private** (a checkbox on the subnet), Azure lends every VM in it a hidden shared public address for outbound traffic, called *default outbound access*. The portal ticks that box for new subnets; check that it is ticked, and the three paths above are the only ones.

Public subnet, private subnet

The words sound like a setting. They are not; they describe a consequence:

	AWS	Azure
Public subnet	route table has `0.0.0.0/0 → internet gateway`	no such thing; a *machine* is public when it has a public IP
Private subnet	no line out; or out-only via a NAT gateway	a subnet checkbox: no default outbound access. Its machines get out only via a public IP, a NAT gateway or LB outbound
Who lives there	public: the few pieces whose job is to face the internet · private: your app, your database: everything that does not need to face it	

- A VM in a private subnet with no NAT has **no way to reach the internet, and the internet has no way to reach it**. In routing terms it does not exist. It costs nothing: it is the absence of a route and of an address

What a cut-off VM can still do

A VM with no route out and no public address can still talk to everything **inside** its own network, and to the cloud's own services that live there:

- other VMs in the VNet or VPC, over private addresses
- the platform's **DNS resolver**: Azure's is `168.63.129.16`, AWS's is the VPC's second address (`10.61.0.2` in a `10.61.0.0/16` VPC). Resolving a name works; reaching the address it resolves to does not
- the **metadata service** at `169.254.169.254`, where a VM reads its own identity, and the platform's health checks

Resolving a name is not reaching it. This week's validator tests reachability, not resolution.

How far to split

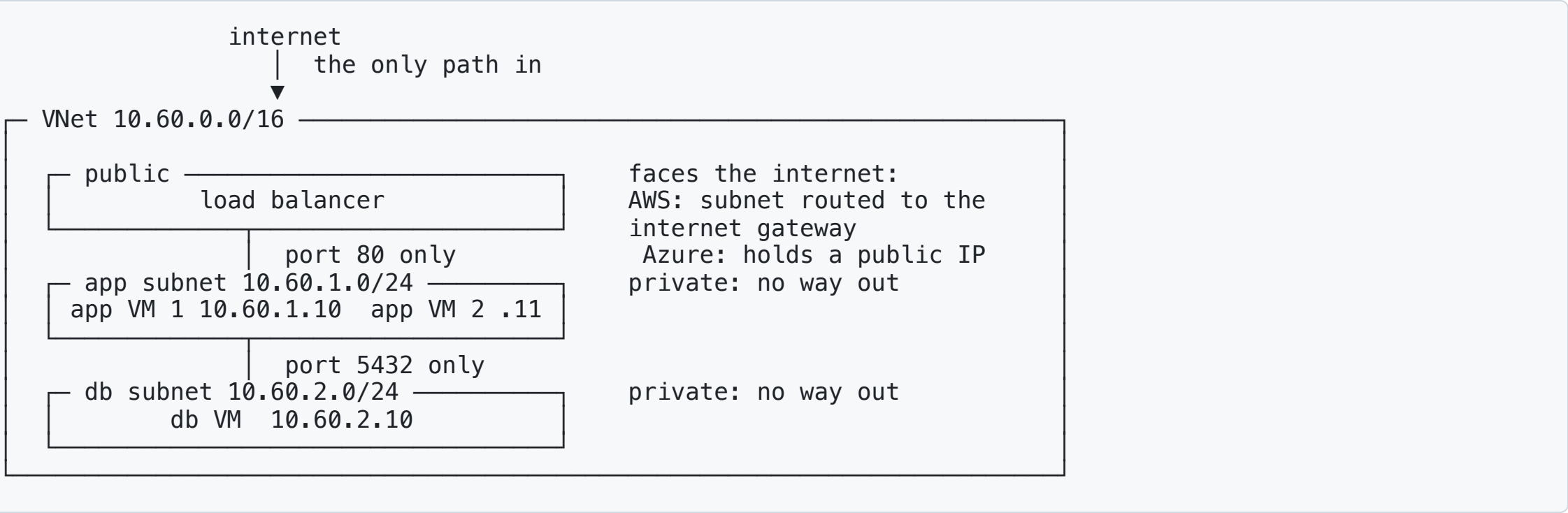
The public / private split is in nearly every real setup: the load balancer in front, everything else behind. How far the private side is split beyond that is a design choice:

- **One subnet per tier** (app subnet, database subnet): how both clouds' reference architectures draw it. The rule "database only from the app subnet" becomes an address range in a table that anyone can read and test, and a subnet boundary is something route tables and auditors can see
- **One private subnet, a firewall per machine**: equally common, and the norm on AWS, where security groups per role do the work and subnets are mostly about availability zones

This course splits by tier, for the readable rule above and for one practical reason: managed database services (Azure's PostgreSQL Flexible Server, for one) demand a subnet of their own.

The two-tier setup

Three VMs: two app VMs doing the same job, one database VM, in subnets by exposure:



The VMs have no way out: on AWS no route out, on Azure no public address. The database sits two firewalled steps from the internet: the LB, then an app VM.

Firewalls

A **firewall** is a filter standing at a boundary — in front of a machine or a subnet. It inspects every packet that tries to cross and either lets it through or drops it, according to a list of rules. A rule names a source (an address range or a group), a destination port, and a verdict.

The difference from routing, stated plainly:

- The **route table** answers: is there a path for this packet, and where does it lead?
- The **firewall** answers: this packet has a path — is it allowed to use it?

A packet arrives only when both answers are yes. The private subnets in our setup use both: no route out, and rules on what may come in.

Firewalls: an example

The app VM in our setup:

- **Routing**: a path exists from the load balancer to `10.60.1.10` — packets can be delivered
- **Firewall**: of everything that path could carry, only **port 80** is allowed in
- A connection attempt to port 22 or 5432 travels the same path and is dropped at the wall

The working principle is to **minimize exposure**: every open port is a possible way in, so only the ports the application needs are open, and only from the sources that need them. Everything else stays closed even where a path exists.

NSGs and security groups

Azure calls its firewall a **Network Security Group (NSG)**, AWS calls it a **Security Group**. Both are **stateful**: if the request was allowed in, the *response* is allowed back out automatically. You write rules for who may *start* a conversation, never for the replies.

	Azure NSG	AWS Security Group
Attached to	a subnet, or one VM's network card	one instance's network card
Rules can point at	address ranges, or named groups like "Internet"	address ranges, or **another security group**
Default for incoming	allow from inside the VNet, deny from outside	deny

Note the AWS model: a rule can allow traffic *from another security group*, so membership replaces addresses and the rule stays correct as machines change.

Azure's built-in NSG rules

Every NSG is created with three inbound rules (and three outbound ones) that cannot be deleted. Rules are checked in **priority** order, lowest number first, and the built-in ones sit at the very end:

Priority	Built-in rule	Effect
65000	AllowVnetInBound	anything inside the VNet may reach anything, on any port
65001	AllowAzureLoadBalancerInBound	the platform's health checks may come in
65500	DenyAllInBound	everything else is dropped

The consequence for a rule like "allow 5432 from the app subnet": on its own it changes nothing. The VNet rule already allows that traffic, and it also allows every other subnet on every other port. To make *only from the app subnet* true, you add your own **deny** for the `VirtualNetwork` source at a priority above 65000 (for example 4000): your allow at 100 wins for the app subnet, your deny at 4000 catches the rest.

AWS security groups have no built-in allow. There, deny really is the default.

The firewall rules of this setup, complete

The complete list of rules for this setup:

Wall	Rule	Why
app subnet	allow 80 in, from anywhere	the LB must deliver requests
db subnet	allow 5432 (the Postgres port) in, from `10.60.1.0/24` only	the app talks to the database
db subnet, Azure only	deny everything else from `VirtualNetwork`, priority 4000	otherwise the built-in rule lets any VM in the VNet in
—	*(nothing else)*	there is nothing else

Note the db rule's *source*: the app subnet's range. A machine in any other subnet of the same VNet, for example a compromised VM, cannot open a connection to the database. Firewall rules apply between subnets, not only against the internet. On Azure that sentence is only true with the deny row in place.

Default deny

Start from **nothing gets in**, then add openings one by one, each with a reason:

- Port 80 to the app tier, because the LB delivers user requests there
- Port 5432 from the app subnet to the database, because the app queries it
- Nothing else: no other traffic is part of how this application works

The most common and most dangerous mistake: SSH open to `0.0.0.0/0`. Automated scanners find a new public IP with port 22 open within minutes; this is measured behavior, not folklore. Administrative access needs its own controlled path.

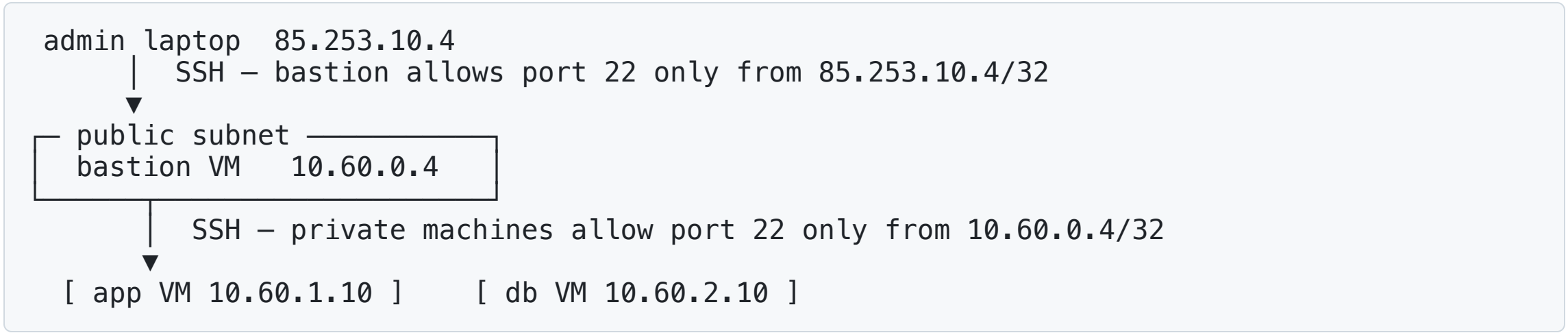
Administrative access

The setup so far leaves an open question: if nothing can reach the machines, how do **you** reach them when something needs attention?

The tension is real: every path made for the administrator is also a path an attacker can try. There are two standard answers, the bastion host and the managed bastion service. This week's lab uses a third: no standing access at all.

The bastion host

A **bastion host** (also called a jump host) is one small VM whose only job is to be the administrators' entry point. Nothing else runs on it: fewer programs, fewer weaknesses.



The firewall rules chain: the bastion accepts SSH from one known address; the private machines accept SSH only from the bastion. Every administrative session passes through this one machine, which makes it one place to watch, log, and defend.

The bastion as a service

Both clouds sell the same idea as a managed service, so the entry-point machine is not yours to run and patch:

- **Azure Bastion**: you open a shell to your VMs through the browser, inside the portal. The VMs need no public IPs and no SSH open to the internet
- **AWS Systems Manager (SSM) Session Manager**: an agent on each VM connects *outward* to AWS; you get a shell through the AWS console. No inbound SSH exists at all. The console's *Connect* button offers it as one tab; the other tab, *EC2 Instance Connect*, is ordinary SSH in a browser and needs a public address and port 22 open, so it is not a bastion

The trade in both cases: a running cost, against not operating your own bastion.

Access in this week's lab

This week's lab deliberately ends with **no administrative access at all**: the machines are set up, verified, and then every access path is removed.

That is workable because of how the machines are treated:

- Setup happens once, at creation, by hand or from a script the machine runs at boot
- A machine that misbehaves is not repaired but **replaced**: destroyed and recreated from the same steps
- When access is genuinely needed, it is added back the same way it was removed, used, and removed again

This approach has a name: **immutable infrastructure** — machines are never changed after creation, only replaced. Machines that live long need a bastion or a managed access service.

The load balancer

Everything so far closes doors. The load balancer is the **one** you deliberately open.

- Owns the **public IP**. Backends have none, so the LB is the only way in
- Distributes requests across backends: here two VMs, the same with any number. Each request goes to one of them; over many requests the split is even
- **Health checks**: the LB sends each backend a periodic request of its own; a backend that stops answering stops receiving traffic, automatically

```
20 requests ──▶ LB ──▶ backend A (10 of them)
                  └──▶ backend B (the other 10)
```

How a backend is picked: Azure's Load Balancer hashes the packet's addresses and ports and maps the hash to a backend. Every new connection can land on either machine; consecutive connections from one client, whose source ports are neighbours, often land on the same one. Over many clients the split is even.

Health checks

The load balancer has a list of backends and no idea which of them work. So it asks, all the time: every few seconds it sends each backend one request and looks at the answer.

- **TCP check**: opens a connection to the port. Proves that something listens, nothing more. A crashed application whose port is still held open passes
- **HTTP check**: sends `GET <path>` and requires status **200**. Proves the application itself answers. This week's path is `/hello`, the course beacon's endpoint

A backend that fails two checks in a row is marked unhealthy: it gets no new connections until it answers again. Nothing else happens; the machine keeps running, and the check keeps asking. With a 5-second interval, a dead backend is out of rotation in about 15 seconds and back in as soon as it answers.

Azure calls this a **health probe**, AWS a **health check**; both are set up together with the load balancer.

The load balancer as a way out (Azure)

Three things can lend a private VM a public address: its own public IP, a NAT gateway, and the load balancer. The load balancer is the one that catches people.

The LB holds a public IP. Replies to requests it forwarded are always allowed back through it; the firewall is stateful. But an Azure load-balancing rule can additionally translate connections that the **backend starts** to the LB's public address: **SNAT** (source network address translation, the same sender rewrite the home router did). With that on, your "private" VM has full internet access through the LB, and this week's validator will see it.

When you create the rule, the portal asks about outbound SNAT. Choose **"(Recommended) Use outbound rules"**, which means: this rule does not translate outbound. There is no outbound rule in our setup, so nothing gets out.

AWS's load balancer never does this: it forwards inbound requests and nothing else. Outbound on AWS is only ever a route to a NAT gateway.

The OSI layers

Network traffic is described with a 7-layer reference model, **OSI** (Open Systems Interconnection, a standard from 1984). Each layer is carried inside the one below it:

#	Layer	What it does	Real examples
7	Application	what the data means	HTTP, DNS, SMTP
6	Presentation	encoding, encryption	TLS
5	Session	keeping a dialogue together	in practice: inside the application
4	Transport	end-to-end delivery, ports	TCP, UDP
3	Network	addressing and routing between networks	IP (this lecture's first half)
2	Data link	delivery inside one local network	Ethernet, Wi-Fi, MAC (hardware) addresses
1	Physical	bits on a wire or radio	cable, fiber, antenna

Layers 5 and 6 are not separate pieces in the modern internet; their jobs live inside applications and TLS. The numbering survives as vocabulary: a device is named after the highest layer it understands — an "L2 switch", an "L3 router", an "L4 or L7 load balancer".

L4 vs L7 — what the LB understands

	L4 (transport)	L7 (application)
Sees	TCP connections	HTTP requests
Can route by	port	path, host, headers, cookies
Azure	Load Balancer	Application Gateway
AWS	NLB (network load balancer)	ALB (application load balancer)

- L4 is a pipe: fast, dumb, forwards bytes
- L7 reads HTTP: can send `/api` one way and `/static` another, retry a failed request, unwrap HTTPS
- **For this week either works**; the requirement is that the LB is the only entry point, not which layer it works at

The two clouds side by side

Everything from the first half of the lecture, in both vocabularies:

Concept	Azure	AWS
Your own network	VNet	VPC
Subnet	subnet	subnet, tied to one availability zone
Route to the internet	built in; cannot be removed	internet gateway + a route table line you add
Way out for a private VM	public IP, NAT gateway, or LB outbound (subnet marked private)	a route to a NAT gateway
Firewall	Network Security Group, on a subnet or a network card	Security Group, on an instance
Firewall default	allow inside the VNet, deny from outside	deny
Load balancer, L4 / L7	Load Balancer / Application Gateway	NLB / ALB
Temporary admin access this week	public IP + SSH rule, removed at the end	none: the VM sets itself up at boot

What you set up this week: Azure

The lab builds the front half of this lecture's setup: the two app VMs behind the load balancer. The database tier stays on the slides. The products, in the order the lab creates them:

1. **Resource group**: the container everything below lives in; deleting it deletes all of it
2. **Virtual network** `10.60.0.0/16` with one subnet `app` `10.60.1.0/24`, marked private
3. Two **VMs**, the free-tier size in your region (B2ats_v2 or B1s), Ubuntu 24.04, private addresses assigned by Azure. Temporarily: a public IP each, so that software can be installed over SSH
4. One **Network Security Group** on the subnet: 80 from the internet, and a temporary SSH rule from your address only
5. **Load Balancer** (Standard, public): a public IP, a backend pool with both VMs, a health probe on `/hello`, one rule 80 → 80 with outbound SNAT off
6. Lock-down: remove the VMs' public IPs and the SSH rule. The VMs have no way out and no way in except through the LB

Azure for Students allows **three public IPs** per subscription: two temporary ones on the VMs, one on the load balancer. The A1 VM holds one, so it must be gone first: delete it once A1 is graded, or do A2 first and A1 afterwards. Nothing needs to survive after grading.

What you set up this week: AWS

1. **VPC** `10.61.0.0/16` from the wizard, which also makes: an **internet gateway**, two **public subnets** in two availability zones (the ALB requires two), two **private subnets**, the route tables, a **NAT gateway** in one public subnet
2. Two **security groups**: `alb-sg` allows 80 from anywhere; `app-sg` allows 80 from `alb-sg`
3. Two **EC2 instances**, t3.micro, Ubuntu Server LTS, one per private subnet, no public IP. Software is installed at first boot by a **user-data script** (a shell script the cloud runs once at creation) through the NAT gateway
4. **Application Load Balancer** in the two public subnets, a **target group** with both instances, health check `/hello`, listener 80 → target group
5. Lock-down: delete the NAT gateway and release its public address. The private route tables have no way out

No SSH exists on the AWS path: everything the instance needs is in its boot script.

The problem the edge solves

Our setup lives in one datacenter, in one region. Three problems arrive with real users:

- Users are everywhere. For someone far away, every request crosses half the planet, and that distance is latency on every single click
- Every request reaches your machines: static files, repeated pages, junk traffic — all of it consumes your capacity
- Attacks arrive from everywhere at once, and two VMs cannot absorb what a network of thousands of machines can send

The shape of the solution: machines **close to the users**, in hundreds of locations worldwide, that handle what they can and pass on only what they must.

The edge

That worldwide layer of machines in front of servers is what the industry calls **the edge**: it sits at the edge of the network, near the users. Not a standard's term, but every vendor's: AWS runs CloudFront in *edge locations*, Cloudflare calls itself an *edge network*. Your own server behind it is called the **origin**.

The edge layer does four jobs:

- **Content caching**: static content answered near the user
- **HTTP filtering**: requests inspected, attack patterns dropped
- **Attack absorption**: floods soaked up by shared capacity
- **TLS termination**: the encrypted session ends at the edge

```
user ⟶ DNS ⟶ edge: cache, filter, absorb, decrypt ⟶ your LB (the origin)
```

CDN: content delivery network

The observation a CDN is built on: most of what a web application serves is **identical for every user** — images, scripts, styles, fonts, pages that rarely change. This is called **static content**, and serving it needs no logic, only delivery.

- A CDN keeps **copies** of that content on its edge machines (a *cache*); a user's request is answered by the nearest machine that holds a copy
- Only what the cache cannot answer — dynamic, per-user responses — travels on to the origin
- Cached copies carry a lifetime and are refreshed when it expires, the same caching idea as DNS TTL from week 1

The effect is double: users get content from nearby (latency drops), and the origin never sees most requests (load drops). Your two VMs serve only what genuinely needs them.

Costs: this week bills by the hour

New this week: resources that cost money **while existing**, not while used.

Resource	Order of magnitude
Load balancer / ALB	~20 €/month *just for existing*
NAT gateway	~30 €/month + per-GB
2 small VMs	~15 €/month

- The habit from week 1 stands: **every exercise ends with delete**. The cost page the next day shows whether it did
- This week's validation is designed for it: once your check is green, the result is **saved**; tear everything down the same evening

Spending limits: what the clouds actually offer

On a normal pay-as-you-go account, neither Azure nor AWS lets you set a hard limit — there is no "stop everything at 20 €". The reasons are structural:

- Billing data lags hours behind usage; by the time a limit fired, it would already be passed
- Enforcing a cap would mean deleting running resources, and destroying a customer's data over a few euros is a worse failure for the cloud than sending a bigger bill
- So the default is: everything keeps running, and everything keeps billing

What you can get is **alerting**: a budget with thresholds that email you at, for example, 50% and 80% of a monthly amount. The alarm does not save money — it tells you that something is still running, days before the bill would.

Your student accounts are the exception: there the **credit itself is the cap** — Azure for Students suspends when it runs out, the AWS free plan closes. Running out is not a fine, but it is the hassle of a new account.

Setting an alert up takes four minutes and is part of the pre-semester setup guide. If you skipped it, do it today.

Assignment 2: what you build next

The front half of this lecture's setup, on your own account. The database tier is not part of A2:

- Two VMs in a private subnet, **no public IPs**
- A load balancer as the only way in; SSH open to no one
- On each VM runs the **course beacon**, a small container we provide. It reports what the machine can see from inside: its own ID, its private IP, and that the internet is not reachable. Two different IDs behind one address is how the validator sees that the load balancer really has two machines behind it
- Checking is automatic: you enter your student code and the load balancer's address on the course validation server, it verifies the setup through your LB, and every check is stored permanently under a random **check code**. That code is your Moodle submission, so you tear everything down the same day

We build this together in the practice session, step by step. Deadline: **Sunday of week 3, 23:59.** 2 points.



Questions