

The Cloud Architect Course

From zero to designing cloud systems — a B4LCILC course

August 13, 2026

The Cloud Architect Course

From zero knowledge to designing cloud systems a company would bet its business on.

How this course works

You are not training to *operate* cloud systems. You are training to **design** them: to take a messy business problem (“we need to sell to a million customers and never lose an order”), turn it into a drawn, costed, defensible technical plan, and then persuade both the engineers who will build it and the executives who will pay for it. That is the job of a **cloud architect** (also advertised as *solutions architect*, *cloud solutions architect*, or *cloud domain architect*), and it is a learnable craft with a clear syllabus — this one.

The course has five stages. **Stage 1 (Weeks 1–6): The building blocks, as decisions.** Every cloud component, re-taught not as a fact but as a *choice with trade-offs* — because an architect’s unit of work is the decision. **Stage 2 (Weeks 7–14): The design domains.** The six pillars of the Well-Architected Framework — reliability, security, performance, cost, operations, sustainability — each taught deeply with its standard patterns and a worked design. **Stage 3 (Weeks 15–20): The pattern catalog.** The dozen architectures that solve 95% of real problems, and — more importantly — when *not* to use each. **Stage 4 (Weeks 21–26): Migration and the real world.** Moving existing companies to the cloud, and the constraints (law, legacy, lock-in) that make real architecture harder than whiteboard architecture. **Stage 5 (Months 7–12): The craft.** The documents, diagrams, reviews, presentations, and certifications that make you *employable* as an architect, capped by three portfolio-grade design projects.

Who this course is for. It assumes zero prior knowledge: everything it depends on is re-taught briefly before it’s used. If you have already completed **The Cloud Engineer Course** (or you work hands-on in cloud today), Stage 1 will be familiar — *skim it, but do its exercises anyway*, because it re-frames things you know as facts into things you must weigh as decisions, and that re-framing is the whole shift from engineer to architect.

Rules for the whole course: study **one hour a day, six days a week** — consistency beats intensity. Every module ends with a **Vocabulary table** (the words you must own), **Exercises** (design work, not reading — architecture is learned by drawing and deciding), and a **Milestone** (proof you’re ready to move on; do not skip ahead past an unmet milestone). And from Week 1, keep a **Design Journal**: every architecture you draw, every trade-off you call, every prediction you make about cost or failure. Ten months from now it becomes your

interview portfolio; nothing impresses a hiring panel like a dated notebook of forty designs with honest notes on what you got wrong.

What a cloud architect actually does

Think of a building architect. She doesn't lay bricks — but she must know exactly what bricks can and cannot do, what they cost, and how buildings fail. She listens to a client (“a family home, sunny, under this budget, on this awkward plot”), and turns wants and constraints into *drawings and specifications* precise enough that builders can build from them and the client can sign off on them. Then she stays through construction, answering questions and adjusting the plan when the ground turns out to be softer than the survey said.

Swap bricks for servers and that is the job. A cloud architect spends their week doing some mix of: **listening** to business stakeholders and extracting real requirements from vague wishes; **designing** — choosing components, drawing diagrams, writing down decisions and why; **reviewing** other people's designs against the six pillars; **estimating** what a design will cost per month and defending that number to finance; **planning migrations** of old systems into the cloud; **presenting** — the same design explained one way to engineers and a completely different way to executives; and **mentoring** engineers so the standards survive contact with deadlines. On many teams there's also **pre-sales**: sitting beside a salesperson in front of a prospective customer, sketching the solution that wins the deal.

What an architect is *not*: the best coder in the room (usually isn't), the person who types the most (types the least), or a lone genius who hands down blueprints (the fastest way to be ignored). The architect's real output is **good decisions, written down, that others willingly build**.

The job description you are training to

In August 2026 we pulled real Cloud/Solutions Architect job postings and templates — a recruiter's cloud-architect template (KORE1), a staffing firm's job description (4 Corner Resources), Microsoft's own definition of the architect role in the Azure Well-Architected Framework, a pre-sales Solutions Architect posting (Arpio, AWS disaster-recovery), an enterprise Solution Architect posting (Intel, via Built In), and an enterprise Cloud Domain Architect posting (Halliburton, Azure-first). Strip away the company flavor and the same twelve requirements appear again and again. This course is built to that list — here is exactly where each one is taught:

What the job descriptions ask for (their words, paraphrased)	Where this course teaches it
“Design scalable and secure cloud architectures tailored to business and technical requirements” — end-to-end solution design	Stages 1–3, capstones in Stage 5
“Deep platform expertise” in AWS and/or Azure — compute, storage, networking, IAM, account structure	Stage 1 + certification track in Stage 5

What the job descriptions ask for (their words, paraphrased)	Where this course teaches it
“Run Well-Architected reviews” against the six pillars	Stage 2 (the pillars), Stage 5 Module 12 (running reviews)
“Lead migration/modernization initiatives — which workloads move as-is, get rearchitected, or retire”	Stage 4 Module 10 (the 7 Rs, wave planning)
“Design landing zones, account structure, guardrails, and reference patterns teams deploy within”	Stage 4 Module 10
“Integrate security and compliance into the design rather than bolting it on” — zero trust, identity	Stage 2 Module 5, Stage 4 Module 11
High availability and disaster recovery — “RTO/RPO gaps, downtime costs, ransomware exposure”	Stage 2 Module 4, Stage 3 Module 9 (multi-region)
“Own the cloud cost model — tagging, showback, reserved capacity, rightsizing”; estimate solution costs	Stage 2 Module 6, Stage 5 Module 13 (proposal costing)
Network architecture design — VPCs, hub-and-spoke, hybrid connectivity	Stage 1 Module 2, Stage 3 Module 9, Stage 4
“Create architectural documentation, diagrams, and standards”; “maintain Architecture Decision Records”	Stage 1 Module 3 (diagrams), Stage 5 Module 12 (ADRs, diagram sets)
Stakeholder communication — “present technical concepts to C-level and technical audiences”; “defend architectural decisions to security, finance, and engineering”	Stage 5 Module 13
“Mentor the cloud and platform engineers who build against your standards”; pre-sales support — demos, POCs, RFPs	Stage 5 Module 13

Certifications, verified against the same market: the standard ladder is **AWS Certified Solutions Architect – Associate (SAA-C03)** — the most-requested single credential, typically **2–3 months** of prep — followed by **AWS Certified Solutions Architect – Professional (SAP-C02)**, typically **4–8 months** more; Azure-heavy shops ask for **AZ-305 (Azure Solutions Architect Expert)**; large enterprises sometimes add **TOGAF** for enterprise-architecture method. The full plan is in Stage 5, Module 14.

STAGE 1 — THE BUILDING BLOCKS, AS DECISIONS (Weeks 1–6)

Cloud Engineer Course graduates: skim the explanations, but do every exercise. The facts are the same; the questions are new.

Module 1 (Weeks 1–2): What the cloud is, and how architects think

The cloud in one paragraph (the zero-knowledge refresher). The cloud is *someone else's computers, rented by the hour, managed by software*. AWS, Microsoft Azure, and Google Cloud run warehouses of servers (**data centers**) grouped into geographic **Regions** (e.g., “Asia Pacific (Bangkok)”); each region contains several isolated **Availability Zones (AZs)** — separate buildings with independent power, close enough for fast connections, far enough apart that one flood or fire can't take out two. You rent slices of all this per second: raw machines (**IaaS** — you manage everything on them), managed platforms (**PaaS** — you bring only your application), or finished software (**SaaS** — you just use it). That's the whole substrate. Everything an architect designs is arranged on top of it.

Now the architect's move: turn every fact into a question. An engineer learns “an AZ is an isolated data center.” An architect immediately asks: “*How many AZs does this workload deserve?*” — because two AZs cost more than one, three more than two, and a marketing brochure site does not deserve what a payments system deserves. This is the course's first and most important idea:

Architecture is the discipline of making trade-offs explicit. There are almost no wrong components — only components wrong *for this workload, this budget, this team, this deadline*.

The trade-off triangle. Every design negotiates between three corners: **fast** (performance — quick for users, quick to build), **cheap** (low monthly bill, low engineering effort), and **resilient** (survives failures, scales, stays secure). You can push toward any two corners; the third pays for it. A startup's prototype should be fast and cheap — resilience can wait. A bank's core ledger must be resilient and fast — it will not be cheap. When a stakeholder says “we want all three,” your job is to smile and ask which one they want *most*, because the design cannot start until they answer. Draw this triangle at the top of every design you do in this course, and mark where the workload sits. It will save you a thousand arguments.

Requirements: the raw material of design. Architects separate **functional requirements** (what the system does — “customers can order lunch”) from **non-functional requirements (NFRs)** (how well it must do it — “under 2 seconds, for 10,000 simultaneous students, 99.9% of the time, within PDPA”). Beginners obsess over the first list; architects earn their salary on the second, because NFRs are what actually determine the architecture. Two magic questions extract NFRs from stakeholders who don't know they have them: “*What happens to the business if this is down for an hour?*” and “*What does success look like at ten times today's size?*”

Vocabulary:

Term	Definition
Region / Availability Zone (AZ)	Region: a geographic cluster of data centers

Term	Definition
	you choose to run in. AZ: an isolated data center (or small group) within it — the unit of “one building can fail.”
IaaS / PaaS / SaaS	Renting raw machines / a managed platform / finished software. The slider from most control to least maintenance.
Workload	Any application or system, treated as a unit of design — “the payments workload.”
Functional requirement	What the system must do (“customers can order”).
Non-functional requirement (NFR)	How well it must do it — speed, scale, uptime, security, compliance, cost. NFRs drive architecture.
Trade-off	What you give up to get what you chose. Every design decision has one; an architect’s job is to name it out loud.
Constraint	A non-negotiable boundary: budget, deadline, law, existing systems, team skills. Constraints are not obstacles to the design — they <i>are</i> the design brief.
Stakeholder	Anyone with a stake in the system: users, engineers, finance, security, executives, regulators. Different stakeholders, different languages — you speak all of them.
Greenfield / brownfield	A brand-new system with no history (greenfield) vs. one entangled with existing systems (brownfield — most real work).
Managed service	A component the provider operates for you (backups, patching, failover included). The architect’s default, unless there’s a written reason otherwise.

Videos for this module (verified links):

Video	Channel	Length	Link
Top 50+ AWS Services Explained in 10 Minutes	Fireship	~10 min	https://www.youtube.com/watch?v=JlBYCM48to

Watch the Fireship tour twice: once now for the map, once at the end of Stage 1 — you’ll be surprised how many services you can now *place* in a design.

Exercises: (1) Pick three apps you use daily (a bank app, a food-delivery app, a video app) and, for each, write its top three NFRs and mark it on the trade-off triangle. (2) Interview a friend about a business idea for ten minutes and extract five functional and five non-functional requirements — notice how the NFRs only come out when you ask the two magic questions. (3) Start your Design Journal with entry #1: the triangle, and one paragraph on why “which corner do we sacrifice?” is a business question, not a technical one.

Milestone: given any one-sentence system description, you can produce its likely NFRs and its position on the triangle in five minutes, out loud, without notes.

Module 2 (Weeks 3–4): Compute, storage, database, network — the four decisions

Every design you will ever draw is these four choices, made deliberately. Here is each one taught as a decision.

Decision 1 — Compute: VM, container, or serverless? A **virtual machine (VM)** is a rented slice of a server that acts like a whole computer — maximum control, maximum maintenance (you patch it, you scale it, you pay while it idles). A **container** packages one application with everything it needs so it runs identically anywhere; an orchestrator (**Kubernetes**) runs and heals fleets of them — great density and portability, but you’ve adopted a complex platform that needs skilled people. **Serverless** (AWS Lambda, Azure Functions) runs your code only when triggered and bills per invocation — near-zero maintenance and perfect for spiky traffic, but with limits (execution time caps, cold starts, and a deeper marriage to one provider). The decision table you should be able to reproduce from memory:

Choose	When	Watch out for
VM	Legacy software, special licensing, full OS control needed, steady predictable load	You own patching, scaling, and 3 a.m.; idle time bills fully
Containers + Kubernetes	Many services, team already has the skills, portability matters	Platform complexity — K8s is a full-time job; overkill for small teams
Serverless	Spiky or unpredictable traffic, event-driven glue, small teams, fast time-to-market	Runtime limits, cold starts, harder cost prediction at huge steady scale, lock-in

The senior insight: **this is a per-workload choice, not a company religion.** Real estates run all three side by side, correctly.

Decision 2 — Storage: object, block, or file — and how hot? **Object storage** (Amazon S3) is a bottomless bucket for files — cheap, absurdly durable (“eleven nines”), the default answer to “where do files go?” **Block storage** (EBS) is the virtual disk bolted to a VM. **File storage** (EFS) is a shared drive many machines mount at once. The architect’s extra dimension is **temperature**: hot data (accessed constantly, priced for speed) versus cold/archive tiers

(Glacier — pennies, but minutes-to-hours to retrieve). Designing lifecycle rules that drift old data to cold tiers is the cheapest cost win in cloud; forgetting to is the most common.

Decision 3 — Database: SQL or NoSQL (and which managed flavor)? Relational/SQL databases (PostgreSQL, MySQL; managed as RDS/Aurora) keep data in strict tables with guaranteed consistency — the default for anything where correctness is sacred: money, orders, inventory, users. **NoSQL** databases (DynamoDB, MongoDB) trade strict structure for flexibility and near-limitless horizontal scale — the default for sessions, catalogs, feeds, telemetry. The decision heuristic: *start with SQL unless you can name the specific reason it won't work* (extreme scale, flexible schema, single-digit-millisecond global reads). And in the cloud, “database” should almost always mean “**managed** database” — the provider handles backups, patches, and failover; a team running its own database on VMs should have a written reason. Add the specialists to your vocabulary: **cache** (Redis — hot data in memory, microsecond reads), **warehouse** (analytics at scale — Stage 3), **queue** (not a database, but often the missing piece — Stage 3).

Decision 4 — Network: the shape of the private world. A **VPC (Virtual Private Cloud)** is your fenced-off section of the provider's network. Inside it, **public subnets** hold the things the internet may reach (load balancers), and **private subnets** hold everything else — app servers and, always, databases. “*The database sits in a private subnet*” is the single most repeated sentence in architecture reviews; the reasoning — nothing can attack what has no path from the internet — is half of network security. Around the VPC: a **load balancer** spreads traffic across servers and routes around sick ones; **DNS** (Route 53) turns names into addresses; a **CDN** (CloudFront) caches content in hundreds of cities so it's fast everywhere; an **API gateway** is the managed front desk for your APIs (authentication, rate limits, logging). Connecting to the old world: **VPN** (encrypted tunnel over the internet) or **Direct Connect** (a private physical line) — the umbilical cords of every hybrid design in Stage 4.

Vocabulary:

Term	Definition
Instance / instance type	One rented VM / its size (CPU + RAM), which sets its hourly price.
Container / Docker / Kubernetes (K8s)	A portable package for one app / the tool that builds and runs them / the orchestrator that runs and heals fleets of them.
Serverless / Lambda	Code that runs only when triggered, billed per run, no servers to manage. Lambda is AWS's version.
Cold start	The extra delay when a serverless function runs after being idle — the classic serverless trade-off.
S3 / bucket / durability	AWS object storage / one named container of files / the chance data survives — S3's 99.999999999% means loss is essentially never.

Term	Definition
Storage tier / lifecycle policy	Price-speed class (hot – cold – archive) / the automatic rule that moves aging data to cheaper tiers.
RDS / Aurora / DynamoDB	AWS's managed SQL services / its cloud-native high-performance SQL / its flagship managed NoSQL.
Consistency	The guarantee that everyone reading data sees the same truth at the same time — SQL's superpower, and what NoSQL relaxes to gain scale.
Cache / Redis	A memory-speed copy of hot data placed in front of a database / the standard tool for it.
VPC / subnet (public, private)	Your private network section / its subdivisions — public faces the internet, private does not. Databases live private. Always.
Load balancer / health check	The traffic director across servers / the heartbeat test it uses to stop sending traffic to dead ones.
CDN / edge	City-level caches of your content worldwide / “the edge” = close to users.
API / API gateway	The controlled doorway one piece of software offers another / the managed front desk for those doorways.
VPN / Direct Connect	Encrypted tunnel over the internet / a private physical line to the cloud — the two ways on-prem meets cloud.

Videos for this module (verified links):

Video	Channel	Length	Link
Kubernetes explained in 15 mins	TechWorld with Nana	~16 min	https://www.youtube.com/watch?v=VnvRFRk_51k
Serverless Computing in 100 Seconds	Fireship	~2 min	https://www.youtube.com/watch?v=W_VV2Fx32_Y
AWS Networking Basics (VPC & Subnets)	KodeKloud	~30 min	https://www.youtube.com/watch?

Video	Channel	Length	Link
			v=QM63dyA_4Pc

Exercises: (1) For each of these five workloads, choose compute, database, and storage, and write one sentence of justification each: a school lunch-ordering site; a bank's transaction ledger; a photo-sharing app; a nightly report generator that runs 20 minutes; a chat app for 5 million users. (2) Take one of your choices and argue the *opposite* choice as persuasively as you can — architects who can't steel-man the alternative haven't understood the trade-off. (3) Journal: your compute decision table, from memory.

Milestone: given any workload in one sentence, you can name its four decisions with justifications in under three minutes — and for at least one decision, name what would make you change your mind.

Module 3 (Weeks 5–6): Reading and drawing architecture diagrams

An architect who cannot draw is a consultant who can only talk. Diagrams are your working language: this module makes you fluent in reading them and competent in drawing them.

The canonical diagram — learn this one first. The **three-tier architecture** is the “sentence structure” of cloud diagrams; most designs are variations of it:

```

Users → DNS → CDN → Load Balancer → [Web/App servers, ×N, multi-AZ,
auto-scaled]
                                   → [Cache]
                                   → [Database: primary +
standby, private subnets]

```

Tier 1 (**presentation**): what users touch — static content from the CDN, requests through the load balancer. Tier 2 (**application**): the fleet of interchangeable servers running your logic, in private subnets, scaled automatically. Tier 3 (**data**): the database, deepest and most protected, with a standby in a second AZ. Traffic flows one way: users never touch the app tier directly, the app tier alone talks to the data tier. Practice drawing this until your hand does it without your brain — it is the whiteboard-interview opener everywhere on earth.

Notation conventions that make you look professional (because they make you *think* professionally): boxes are components — label each with what it is *and* which service (“App servers — EC2, auto-scaling group”). Arrows show the direction a *request* flows, labeled with protocol when it matters. Dashed enclosures show boundaries — the VPC, each subnet, each AZ (draw AZ boundaries and multi-AZ becomes visible instead of claimed). A user/actor stands outside the system. Numbers on arrows (1, 2, 3...) let you narrate a request's journey. And every diagram carries a title, a date, and a legend. The deeper rule: **one diagram, one audience, one question**. A diagram showing everything shows nothing; you'll learn the standard set of zoom levels (context → container → deployment) in Stage 5.

Reading other people's diagrams — the architect's x-ray. When handed a diagram, run this scan out loud: Where does the internet touch this system (every touch point is attack surface)? Where's the data, and is it in a private subnet? What's duplicated (resilient) and what's a **single point of failure** — a box with no twin? Where would it hurt at 10× traffic?

What does each box cost monthly? Five questions, thirty seconds, and you've read the diagram the way a doctor reads an x-ray. Browse the AWS Architecture Center (<https://aws.amazon.com/architecture/>) and run the scan on three published reference architectures.

Vocabulary:

Term	Definition
Three-tier architecture	The classic separation: presentation (users' entry) – application (logic) – data (database). The default shape of web systems.
Tier / layer	A horizontal slice of the system with one responsibility, talking only to its neighbors.
Single point of failure (SPOF)	Any component whose lone death takes the system down. The first thing to hunt for in any diagram.
Multi-AZ	Running duplicates across at least two Availability Zones so one data center can fail invisibly.
Auto-scaling (group)	Machines added and removed automatically with demand — capacity that breathes.
Stateless / stateful	A server holding no unique data (any twin can replace it, so it scales freely) vs. one holding data that must not be lost. Design goal: stateless app tier, state pushed down to database and cache.
Reference architecture	A provider-published, blessed example design for a common problem — architects assemble from these before inventing.
Context diagram	The highest zoom level: your system as one box, plus the users and external systems it touches.
Attack surface	Every point where the outside world can touch the system. Smaller is safer.
North-south / east-west traffic	Traffic entering/leaving the system vs. traffic between components inside it.

Exercises: (1) Draw the three-tier diagram from memory, five days running, until it takes under four minutes with all boundaries (VPC, subnets, two AZs) drawn. (2) Take Module 2's five workloads and draw each one — twenty minutes per diagram, notation rules enforced. (3) Find any real architecture diagram online (the AWS Architecture Center has hundreds), and write its five-question x-ray scan in your journal. (4) The narration drill: diagram in front of you, narrate one user's click from browser to database and back, out loud, numbering the arrows as you go.

Milestone — end of Stage 1: on a whiteboard (or paper, photographed for your journal), you can draw a correct, properly-notated three-tier design for a novel one-sentence workload in under fifteen minutes, narrate a request through it, and answer “what fails if this box dies?” for every box. This drawing-plus-interrogation is precisely the first half of a real architect interview — from here on, everything is depth.

STAGE 2 — THE DESIGN DOMAINS (Weeks 7–14)

The map for this stage is the AWS Well-Architected Framework — the industry’s shared checklist of what “designed properly” means, organized into six pillars: **operational excellence, security, reliability, performance efficiency, cost optimization, sustainability**. (Azure has a near-identical framework; learn one deeply and you’ve learned both.) Job descriptions ask for architects who can “run Well-Architected reviews” by name, so we take the pillars one at a time, and for each you learn three things: the pillar’s **key questions**, its **standard patterns** (the boring, proven answers — architects assemble before they invent), and a **worked example** on a running scenario.

The running scenario for all of Stage 2: *ThaiTicket*, a fictional event-ticketing platform in Bangkok. Normal load: 2,000 visitors/hour. But when a famous artist’s tickets drop at 10:00 a.m., it takes 400,000 visitors in ten minutes, payments must not double-sell seats, and Thai customers’ personal data falls under PDPA. Fast, cheap, resilient — ThaiTicket needs all three and can’t have them, which is what makes it the perfect practice patient.

Watch before Module 4 (verified links):

Video	Channel	Length	Link
The Five Pillars of the AWS Well-Architected Framework	Amazon Web Services (official; a sixth pillar, Sustainability, was added later)	~4 min	https://www.youtube.com/watch?v=KvEDbPmha6o
What is the AWS Well-Architected Framework?	Tech With Lucy	~10 min	https://www.youtube.com/watch?v=MpDJ6TCWKjk

And bookmark the framework itself — <https://aws.amazon.com/architecture/well-architected/> and the full document at <https://docs.aws.amazon.com/wellarchitected/latest/framework/welcome.html> — you will live in it for eight weeks.

Module 4 (Weeks 7–8): Reliability — designing for the day things break

The pillar’s creed: everything fails, all the time. Disks die, AZs flood, deploys go wrong, and a certificate somewhere is always about to expire. Reliability is not the absence of failure — it is the *irrelevance* of failure: designing so that when (not if) a component dies, users never notice.

Key questions (ask these of every design, forever): What happens when each component fails — is there always a “then what”? How does the system handle 10× load? How do we *know* something failed before customers tell us? What are our **RTO** and **RPO** — and who in the business signed off on them? When did we last *test* a recovery?

RTO and RPO — the two numbers that are the disaster conversation. **Recovery Time Objective:** how long may we be down? **Recovery Point Objective:** how much data may we lose (i.e., how old may the last good backup be)? An RTO of 4 hours and RPO of 1 hour means: back within four hours, having lost at most one hour of data. These are *business* decisions with exponential price tags — RPO of 24 hours is a nightly backup (cheap); RPO of ~zero is continuous replication (expensive); RTO of minutes means warm infrastructure idling on standby (very expensive). The architect’s job is to make the business choose numbers *knowingly* — and to design exactly to them, not romantically beyond them.

The standard patterns: **redundancy** (two of everything that matters — N+1) · **multi-AZ** (duplicates across data centers; the load balancer and managed database failover make it automatic) · **auto-scaling** (capacity follows demand) · **health checks + self-healing** (dead instances detected and replaced by machinery, not humans) · **backups, tested** (an untested backup is a hope, not a plan — schedule restore drills) · **graceful degradation** (when overloaded, shed the least important features first: ThaiTicket can drop seat-map previews and keep checkout) · **queues as shock absorbers** (Stage 3) · **avoiding cascading failure** (timeouts, retries with backoff, circuit breakers — so one slow dependency doesn’t drown the fleet).

Worked example — ThaiTicket reliability design. Two AZs in the Bangkok region. Stateless app tier in an auto-scaling group behind a load balancer, pre-warmed by schedule before announced on-sale times (auto-scaling reacts in minutes; a 10:00 drop needs capacity at 09:45). Aurora database, primary in AZ-a, synchronous standby in AZ-b, automatic failover ≈ under a minute. A queue between “buy” clicks and payment processing so a payment-provider slowdown queues orders instead of crashing the site. Backups: continuous, point-in-time restore, monthly restore drill. Numbers agreed with the business: RTO 15 minutes, RPO ~zero for orders (it’s money), RPO 24 hours for analytics data (it’s not). Journal exercise: what did this cost us from the “cheap” corner of the triangle?

Vocabulary:

Term	Definition
RTO / RPO	Recovery Time Objective: how long you may be down. Recovery Point Objective: how much data you may lose. The two numbers that define every DR conversation — and they are business decisions.

Term	Definition
High availability (HA)	Designing so routine failures (an instance, an AZ) cause no user-visible outage.
Disaster recovery (DR)	The plan for <i>big</i> failures — a whole region, a ransomware event — with RTO/RPO targets and a tested runbook.
Redundancy / N+1	Spare capacity so any one component can die: N needed, N+1 running.
Failover	The automatic switch to a standby when the primary dies.
Health check	The automated heartbeat that detects dead components so machinery can route around them.
Graceful degradation	Shedding less-important features under stress so the core survives.
Timeout / retry with backoff	Giving up on a slow call after a limit / retrying with growing pauses — manners that prevent cascade failures.
Circuit breaker	A component that stops calling a failing dependency for a while so it can recover — like the fuse in a house.
Chaos engineering	Deliberately injecting failure (Netflix-style) to prove resilience claims before reality tests them for you.

Exercises: (1) Take your Stage 1 lunch-ordering design and upgrade it to survive: an instance death, an AZ loss, a database failure, and a 10× lunch-rush — draw the before and after. (2) Write the RTO/RPO conversation for three businesses (a blog, an e-commerce site, a hospital records system) as a short dialogue between architect and owner — practice making cost visible in the dialogue. (3) Journal: list every SPOF in ThaiTicket’s design above. There’s at least one left deliberately. (Hint: how many regions?)

Milestone: you can run the “what happens when this fails?” interrogation over any diagram for ten straight minutes without running dry, and you can explain RTO vs RPO to a non-technical owner using a shop-flood analogy in ninety seconds.

Module 5 (Weeks 9–10): Security — designing systems that are hard to hurt

The frame: the shared responsibility model. The provider secures the *cloud itself* — buildings, hardware, the hypervisor. You secure everything you *put in it* — data, identities, configurations, code. Nearly every famous cloud breach is a customer-side misconfiguration (a public bucket, a leaked key, an over-permissioned role), which is why security is an *architecture* problem before it is a tooling problem: the job descriptions say “integrate security into the design rather than bolting it on,” and this module is how.

Key questions: Who and what can access each component, and is every permission the *minimum* needed? Where is data encrypted — at rest, in transit, and who holds the keys? Where are the network boundaries, and what crosses them? How would we *detect* a breach — and could we tell the story afterward from logs? What law applies to this data, and where does the data physically live?

The standard patterns: **least privilege** (every human and program gets the minimum access its job needs — the golden rule against which every IAM policy is judged) · **MFA everywhere, root locked away** · **encryption at rest and in transit, always on** (it's a checkbox in cloud; there is no excuse) · **network segmentation** (public/private subnets, security groups as per-server firewalls; the blast radius of a breach is defined by the walls you drew in advance) · **secrets in a vault, never in code** · **zero trust** (verify every request explicitly — identity, device, context — trust nothing for merely being “inside the network”; the JDs name it, so must you) · **defense in depth** (layers, so one failed control isn't game over) · **audit logging** (CloudTrail-style records of who did what — immutable, monitored) · **security as guardrails, not gates** (encode the rules as automated policy that makes the secure path the easy path, rather than a review meeting that makes security the enemy of delivery).

Worked example — ThaiTicket security design. Customer data (names, emails, payment references) classified as personal data under PDPA → stored encrypted in Aurora in the Bangkok region (data residency), keys in KMS. Network: only the load balancer is public; app tier private; database subnet accepts connections *only* from the app tier's security group. Humans: SSO + MFA; engineers get read-only production access by default and time-boxed elevated access on request (least privilege with an audit trail). Payment card handling delegated to a certified payment provider so raw card numbers never touch our systems — a *design* decision that removes an entire compliance burden, which is security architecture at its best. CloudTrail on, alerts on unusual access. Breach question pre-answered: PDPA has a 72-hour notification duty — the logs and the runbook must let us tell the story in less.

Vocabulary:

Term	Definition
Shared responsibility model	Provider secures the cloud; you secure what's in it. Most breaches are on the customer's side of the line.
IAM / role / policy	Identity and Access Management — who may do what. A policy is a permission list; a role is a wearable bundle of them.
Least privilege	The golden rule: minimum access needed, nothing more, reviewed regularly.
MFA	A second proof of identity beyond the password. Non-negotiable for humans.
Encryption at rest / in transit / KMS	Data scrambled on disk / on the wire / the managed service holding the keys.
Security group	A per-server firewall rule list — “web traffic in from the load balancer only.”

Term	Definition
Network segmentation / blast radius	Dividing the network into walled zones / how far an attacker can reach after one breach. Walls drawn in advance define it.
Zero trust	Verify every request explicitly; trust nothing for being on the “inside.” The modern default posture.
Defense in depth	Multiple overlapping controls so one failure isn’t fatal.
Secrets management	Passwords, keys, and tokens live in a vault service — never in code, never in a spreadsheet.
Audit log / CloudTrail	The immutable record of who did what, when — how you detect trouble and reconstruct it after.
Data classification	Labeling data by sensitivity (public / internal / personal / regulated) so controls match the label, not a guess.
Data residency	Keeping data physically within a country’s borders — a legal requirement in many industries, an architecture input always.
PDPA / GDPR	Thailand’s and Europe’s personal-data laws: consent, breach notification (72 hours), cross-border transfer rules. Stage 4 goes deeper.

Video for this module (verified link): The AWS Shared Responsibility Model — Digital Cloud Training — ~4 min — <https://www.youtube.com/watch?v=ESPBBEK-cvo>

Exercises: (1) Draw ThaiTicket’s diagram and overlay its security: mark every trust boundary, every encryption point, every place credentials live. This overlay habit — same diagram, security lens — is exactly what a review asks of you. (2) Read one public post-mortem of a cloud misconfiguration breach (Capital One 2019 is the teaching classic) and write, in your journal, which *pattern above* would have prevented it. (3) Roleplay with an AI: it plays a startup founder who says “we’ll add security later” — talk them out of it with cost-of-breach arithmetic, kindly.

Milestone: you can take any of your Stage 1 diagrams and produce its security overlay in twenty minutes, and explain least privilege, zero trust, and the shared responsibility model to a non-technical stakeholder without jargon.

Module 6 (Weeks 11–12): Performance and cost — speed and money, the coupled pillars

These two pillars are taught together because they are the same lever pushed in opposite directions, and the architect is the person with a hand on the lever.

Performance efficiency — key questions: Where will users feel slowness first? What's computed repeatedly that could be computed once and cached? Is each component the *right tool* (SQL doing a search engine's job is a performance bug of the architecture, not the code)? How does performance change at 10× — and where's the first bottleneck?

Performance patterns: caching layers — the single highest-leverage performance pattern: browser cache — **CDN** at the edge (static content served from a city near the user; can absorb the vast majority of read traffic) — application cache (**Redis** in front of the database for hot reads: seat maps, sessions, product pages) — database query cache. Each layer answers requests before they reach the expensive core; the design questions are always *what may be cached, for how long, and how is it invalidated when truth changes*. Then: **read replicas** (copies of the database that serve reads so the primary keeps writing) · **asynchronous processing** (don't make users wait for work that can happen after — email receipts, thumbnails, reports) · **right tool per job** (search — a search engine; analytics — a warehouse; hot lookups — a key-value store) · and **measure first** (performance work without measurement is superstition).

Cost optimization — key questions: What does this design cost per month at today's load — and per *unit* (per order, per customer)? What's running at 3 a.m. that doesn't need to be? What steady usage could be committed at a discount? What would finance say the trend is?

Cost patterns: rightsizing (most fleets are quietly over-provisioned; shrinking to measured need is free money) · **reservation strategy** — the pricing menu: **on-demand** (full price, full freedom) for spiky/unknown load, **reserved instances** / **savings plans** (1–3 year commitment, 30–70% off) for the steady baseline, **spot** (up to 90% off, reclaimable at short notice) for interruptible batch work; the architect's move is layering them: reserve the floor, auto-scale the middle on demand, spot the batch · **storage lifecycle** (Module 2's tiering, automated) · **turn things off** (non-production environments sleeping nights and weekends can cut their cost by two-thirds) · **watch egress** (data *leaving* the cloud is billed; chatty cross-region designs and big public downloads surprise everyone once) · **tagging and showback** (every resource labeled with owner/project so every baht is attributable) · and the crown metric, **unit economics**: not “the bill is ฿800k/month” but “cost per ticket sold is ฿1.90 and falling.” Growing bills are fine; growing *unit costs* are an architecture smell. This discipline has a name — **FinOps** — and architects sit in the middle of it.

Worked example — ThaiTicket, both lenses. Performance: CloudFront serves the artist pages and seat-map images (the 400,000 gawkers mostly never touch the servers); Redis caches seat availability with a 2-second TTL — stale enough to be cheap, fresh enough that checkout (which double-checks against Aurora, the source of truth) prevents double-selling; receipts and tickets are generated asynchronously after payment. Cost: the steady 2,000-visitors/hour baseline runs on a savings plan ($\approx 40\%$ off); on-sale surges run on-demand for the hour they exist; analytics jobs run on spot at night; staging sleeps outside business hours; every resource tagged project:thaiticket. The proposal to the CFO reads: “฿62,000/month baseline,

≈\$9,000 per major on-sale event, cost per ticket ≈\$1.90 declining with volume” — and *that sentence* is what a cost-fluent architect sounds like.

Vocabulary:

Term	Definition
Latency / throughput	How long one request takes / how many requests per second the system sustains. Related, not the same.
Bottleneck	The narrowest point that sets the whole system's pace. Optimizing anywhere else is decoration.
Cache / TTL / invalidation	A fast copy of expensive-to-fetch data / how long it may be trusted / the hard problem of refreshing it when truth changes.
CDN	The outermost cache — your content in hundreds of cities. First answer to “make it fast globally.”
Read replica	A copy of the database serving reads, so the primary keeps its strength for writes.
Asynchronous processing	Doing non-urgent work after replying to the user, usually via a queue.
On-demand / reserved / savings plan / spot	Full-price flexible / 1–3 year commitment at 30–70% off / same idea, more flexible / up to 90% off but reclaimable. The architect layers all four.
Rightsizing	Shrinking over-provisioned resources to measured need. Free money in almost every account.
Egress	Data leaving the cloud — billed per GB. The famous bill surprise; design data flows with it in mind.
Tagging / showback	Labeling every resource with owner and project / showing each team its own bill. Accountability changes behavior.
Unit economics	Cost per business unit (per order, per user). The metric that makes a bill meaningful — and the architect's favorite sentence in front of a CFO.
TCO	Total cost of ownership — sticker price plus people, licenses, migration, and exit costs over the whole life of a choice.
FinOps	The discipline of making cloud spend visible,

Term	Definition
	allocated, and continuously optimized.

Video for this module (verified link): What is FinOps? — FinOps Foundation (official) — ~2 min — https://www.youtube.com/watch?v=Y-c_xw9bHFw

Exercises: (1) Price ThaiTicket’s baseline in the AWS pricing calculator (search “AWS pricing calculator” — the exercise of itemizing a real design once demystifies every future cost conversation); compare your total to the \$62,000 above and explain any gap. (2) Add the caching overlay to two of your Stage 1 diagrams: mark every cache, its TTL, and its invalidation story. (3) An AI plays an engineer who wants everything on-demand “to keep it simple” — negotiate the reservation strategy, with numbers.

Milestone: for any design you’ve drawn, you can state its three biggest cost lines and its likely first bottleneck — and propose one change that improves both at once (there almost always is one; caching is usually it).

Module 7 (Weeks 13–14): Operational excellence, sustainability — and running all six pillars together

Operational excellence is the pillar that asks: *can humans actually run this thing, calmly?* Key questions: How does a change reach production — through an automated, tested pipeline, or through a heroic human? How do we know the system is healthy *right now*? When it breaks at 3 a.m., what does the on-call engineer actually do? Patterns: **infrastructure as code** (the environment defined in reviewed, version-controlled text — Terraform/CloudFormation — so it’s reproducible and auditable; an architecture that exists only as console clicks is folklore, not engineering) · **CI/CD pipelines** with safe deployment strategies (**blue-green**: stand up the new version beside the old and switch; **canary**: give the new version 5% of traffic and watch) · **observability** (logs, metrics, traces, and alerts tied to *user-visible* symptoms, not machine trivia) · **runbooks** (the written what-to-do for each known failure) · **blameless post-mortems** (after incidents: what failed, why, what prevents a repeat — no villains, or people stop telling the truth). The architect’s part: *design* for operability — a system twice as clever and half as observable is a worse system.

Sustainability, the sixth pillar, asks you to waste less of the physical world: rightsize (idle CPUs burn real electricity), scale to demand instead of peak-provisioning, use managed and serverless services (shared infrastructure is fuller infrastructure), tier storage, and delete what’s dead. Conveniently, the sustainable choice and the cheap choice are usually the same choice — say both in reviews and you’ll carry the room twice.

Running the pillars together — your first Well-Architected review. In real reviews the pillars *conflict*: multi-region reliability fights cost; strict security review fights operational velocity; performance caching fights data-freshness correctness. The framework doesn’t resolve conflicts for you — it forces them into the open, where the business can choose. That is its entire genius, and yours to wield. The review method itself (question sets, prioritizing findings, the report) is taught fully in Stage 5, Module 12; this fortnight you rehearse the raw skill: interrogating one design from six directions in one sitting.

Vocabulary:

Term	Definition
Infrastructure as Code (IaC) / Terraform	Infrastructure declared in version-controlled text files that tools turn into reality / the most popular such tool.
CI/CD pipeline	The automated conveyor that builds, tests, and deploys every change.
Blue-green / canary deployment	Two safe-release patterns: switch traffic between old and new copies / trickle traffic to the new version and watch.
Observability (logs, metrics, traces)	The system's ability to explain itself: event records, numbers over time, and per-request journey maps.
Alert / on-call / runbook	The automated page when thresholds break / the human rotation that answers it / the written script they follow.
Blameless post-mortem	The honest, no-villains review after an incident that produces prevention, not punishment.
Drift	Reality diverging from the IaC definition because someone clicked something. The enemy of reproducibility.
Toil	Repetitive manual operational work that automation should have absorbed. Architects design it out.
Well-Architected review	A structured interrogation of a workload against all six pillars, producing prioritized findings. Stage 5 teaches you to run one.

Exercises: (1) The six-lens drill: take your best diagram and spend ten minutes per pillar writing findings — sixty minutes, one design, six lenses; this drill is Stage 2 in miniature, repeat it weekly from now on. (2) Read two AWS post-incident reports (published on their status pages after major outages) and identify which pillar's pattern failed. (3) Journal: for ThaiTicket, write the three pillar-conflicts you'd surface to the business, each as a one-sentence choice ("We can have X or Y at this budget — which?").

Milestone — end of Stage 2: the mock review: an AI generates a flawed architecture for a Thai online-lending startup; you must produce written findings across all six pillars, including at least: a reliability gap (their single-AZ database), a security gap (over-broad IAM), a cost gap (everything on-demand), an operational gap (no IaC), and the RTO/RPO conversation they never had — each finding phrased as a question a colleague could hear without flinching. When your findings list reads like a helpful senior colleague rather than an auditor, Stage 2 is done.

STAGE 3 — THE PATTERN CATALOG (Weeks 15–20)

Architects do not invent; they **select**. This stage is your catalog of the standard architectures — for each: what it is, when to use it, when *not* to, and its cost/complexity profile. The “when not” columns are the stage’s real payload: any course can tell you what microservices are; knowing when they’ll wreck a company is what you’re paid for. As you study, keep browsing real reference architectures at <https://aws.amazon.com/architecture/> — pattern-spotting in the wild is the exercise that makes the catalog stick.

Module 8 (Weeks 15–17): Application patterns

Monolith vs. microservices — the industry’s noisiest argument, settled calmly. A **monolith** is one deployable application containing all the features; **microservices** split the system into many small, independently deployed services, each owning its data, talking over APIs and events. The honest table:

Pattern	What it is	Use when	Do NOT use when	Cost/complexity
Monolith	One app, one deployment, one database	Small team, young product, unclear domain boundaries — i.e., most new systems	Multiple teams blocked on each other; parts need to scale very differently	Low complexity, low cost; scales further than fashion admits — “boring” is a feature
Microservices	Many small services, independently built, deployed, and scaled	Many teams needing independent release cadence; wildly different scaling per part; proven, stable domain boundaries	Small team (“a distributed monolith is a monolith with network failures added”); domain still shifting	High complexity: every function call becomes a network call that can fail; needs mature CI/CD, observability, on-call

The architect’s ruling: **start monolithic, modular inside; extract services when — and only when — a team-scaling or load-scaling pain actually arrives.** Saying this in interviews, with reasons, marks you as senior; ideology in either direction marks you as junior.

Event-driven architecture — queues and topics, the shock absorbers. Instead of components calling each other and *waiting* (synchronous), components publish **events** (“OrderPlaced”) to a **queue** (SQS — one consumer takes each message, at its own pace) or a **topic** (SNS — every subscriber gets a copy; fan-out). The producer doesn’t know or care who’s listening. Use when: components should survive each other’s outages (the queue holds

messages while a consumer is down — resilience *and* decoupling in one purchase); load is spiky (the queue absorbs the spike; workers drain it steadily); one occurrence triggers many reactions (order placed → charge, email, stock, analytics — four subscribers, zero coupling). Do NOT use when: the user needs the answer *now* (checkout can't “eventually” confirm); or the team is small and a simple synchronous call would do — every queue adds delivery semantics (at-least-once means consumers must be **idempotent** — safe to run twice), dead-letter handling, and a monitoring burden. Cost/complexity: components cheap, debugging dearer — a request's story is now scattered across services and queues, which is why observability (Module 7) stops being optional here.

Three-tier: already yours (Module 3). Use for: the broad middle of web applications — it is the pattern the others are measured against. Not for: very spiky workloads with idle troughs (serverless is cheaper) or genuinely huge multi-team products (see above). Profile: well-understood everywhere, hires easily, medium cost.

Serverless-first: compose managed pieces — API Gateway → Lambda functions → DynamoDB, S3, and queues — owning no servers at all. Use when: spiky or low traffic (scale-to-zero means idle costs ~ nothing), small teams, fast time-to-market, event-driven glue. Do NOT use when: long-running or specialized compute (runtime limits), ultra-latency-sensitive paths (cold starts), massive *steady* load (per-invocation pricing can exceed reserved servers — do the arithmetic at scale), or when portability between clouds is a genuine requirement (this is the deepest lock-in of any pattern — often worth it, but say so out loud). Profile: lowest operations burden in the catalog; cost superb at low/spiky scale, needs checking at high steady scale.

Vocabulary:

Term	Definition
Monolith / modular monolith	One deployable app with everything in it / the disciplined version: one deployable, clean internal module boundaries — the best default for new systems.
Microservices	Many small, independently deployed services, each owning its own data. A team-scaling tool that charges a distributed-systems tax.
Coupling / decoupling	How much components depend on each other's availability and details. Architecture is largely the art of buying the right amount of decoupling.
Synchronous / asynchronous	Call-and-wait vs. send-and-continue. The fundamental choice on every arrow you draw.
Event / event-driven	A fact announced to whoever listens (“OrderPlaced”) / an architecture built from such announcements.
Queue (SQS)	A line of messages; each taken by one

Term	Definition
	consumer at its own pace. Shock absorber and decoupler.
Topic / pub-sub (SNS)	A broadcast channel; every subscriber gets each message. The fan-out tool.
Dead-letter queue	Where messages that repeatedly fail processing are set aside for humans — the pattern’s safety net.
Idempotent	Safe to process twice with the same result — required of consumers, because queues may deliver a message more than once.
Serverless-first	Composing managed, scale-to-zero pieces (functions, managed DBs, queues) instead of running servers.
Lock-in	Dependence on one provider’s proprietary services, priced as switching cost. Not a sin — an economic term to be weighed in the open (Stage 4).

Exercises: (1) Design ThaiTicket’s order flow twice — synchronous three-tier, then event-driven with SQS/SNS — and write one paragraph on what each version does during a payment-provider outage; that paragraph is the whole argument for events. (2) For four companies (3-person startup, 50-engineer scale-up, a bank, a TV-voting app used 4 nights/year), pick a pattern and defend it — then name the trigger that would make each company change patterns. (3) Find one real “we migrated to microservices and regretted it” engineering-blog story and one success story; journal the difference (it’s almost always team size and domain maturity).

Milestone: given a company description, you can recommend a pattern *with its exit ramp* — “start here; when X happens, evolve to Y” — in five minutes. Evolution paths, not verdicts, are how architects actually talk.

Module 9 (Weeks 18–20): Data, global, and hybrid patterns

Data architectures — lake vs. warehouse. Transactional databases (Stage 1) run the business; analytics wants to *ask questions* of it without slowing it down. A **data warehouse** (Redshift, Snowflake, BigQuery) stores *structured, cleaned* data optimized for fast SQL analytics — use when the questions are known and dashboards must be quick; costs more per TB, needs modeling discipline up front. A **data lake** (S3 + catalog + query engines like Athena) stores *everything, raw, cheaply* — logs, clickstreams, images — and applies structure at read time; use when you want to keep everything now and decide questions later; but ungoverned it degrades into the industry joke, the **data swamp**. The modern consensus is both, layered (“lakehouse”): raw truth in the lake, curated marts in the warehouse, fed by **ETL/ELT pipelines**. The architect’s rules: analytics never queries the production database (replicas or pipelines feed it), and every dataset has an owner, a catalog entry, and a retention policy — that governance line is one sentence in your design and a year of pain if you omit it.

Multi-region: active-passive vs. active-active. When one region isn't enough — because the business demands DR from region-scale disasters, or users span continents — you choose:

Pattern	What it is	Use when	Do NOT use when	Cost/complexity
Active-passive	One region serves; a standby region holds replicated data, from “backups only” (cold) to “scaled-down copy running” (warm), promoted on disaster	Business-stated RTO/RPO justify it; compliance demands a DR story	Nobody has signed the RTO/RPO that justifies the spend (most companies' honest need is good multi-AZ + cross-region backups)	1.1×–1.7× cost by standby warmth; moderate complexity — the killer requirement is <i>failover you actually drill</i> , else the standby is theater
Active-active	Two+ regions serve simultaneously; users routed to the nearest; data replicated both ways	Global user base wanting local latency; near-zero RTO genuinely required (payments networks, trading, big SaaS)	Almost everyone else — cross-region <i>write conflicts</i> are one of the genuinely hard problems in computing	2×+ cost, highest complexity in this course; needs conflict-resolving data stores and senior teams

The interview-grade sentence: “*Multi-AZ is table stakes; multi-region is a business case.*” Make the business state the RTO/RPO, price the options, and let the numbers choose.

Hybrid cloud. Part on-prem, part cloud, joined by VPN or Direct Connect — for most established enterprises not a pattern but a *decade-long reality*: mainframes that can't move, latency-bound factory systems, regulated data pinned on-premises, and a migration (Stage 4) passing through. Use as: a deliberate bridge with a direction of travel. Do NOT accept: “hybrid” as a euphemism for “we never decided.” Design notes: identity must be unified first (one login across both worlds — the thing enterprise JDs mean by “hybrid identity integration”); name which systems are the source of truth; watch egress-across-the-wire costs; and expect the network link to *be* a SPOF unless doubled. Complexity: two estates' worth of everything — the honest reason architects push to shrink the on-prem side steadily.

Vocabulary:

Term	Definition
OLTP / OLAP	Transaction processing (many small fast reads/writes — runs the business) vs. analytics processing (huge scans —

Term	Definition
Data warehouse	studies the business). Separate them. Structured, curated storage optimized for fast SQL analytics (Redshift, Snowflake, BigQuery).
Data lake / data swamp	Cheap storage of everything, raw, structured at read time (S3 + Athena) / the ungoverned version, where data goes to be lost.
ETL / ELT	The pipelines that move data from source systems into lake/warehouse (Extract, Transform, Load — order varies).
Data governance / catalog / retention	Ownership, documentation, and lifespan rules for every dataset — one design sentence that saves a year of pain.
Replication (sync / async)	Copying data continuously to another database or region — instantly-consistent-but-distance-limited vs. slightly-behind-but-anywhere. Async lag is where RPO comes from.
Active-passive / failover drill	Standby-region DR / the scheduled rehearsal that proves it works. An undrilled failover is a hope.
Active-active	Multiple regions serving at once with two-way replication. Powerful; genuinely hard; usually unnecessary.
Write conflict	Two regions changing the same data at once — the technical reason active-active is expert territory.
Hybrid cloud / hybrid identity	On-prem + cloud joined as one estate / one sign-on across both — the first thing to unify.
Direct Connect	The private physical line joining data center to cloud — the hybrid umbilical, doubled if it matters.

Exercises: (1) ThaiTicket goes regional: design the Singapore expansion twice — active-passive (Bangkok primary) and active-active — with cost multiples and RTO/RPO for each; write the one-page recommendation and *make a call*. (2) A retailer has 15 years of sales in a production SQL database and wants “AI-ready analytics”; sketch the lake + warehouse design and write the three governance sentences. (3) Pattern-spot: pick three reference architectures from <https://aws.amazon.com/architecture/> and name every catalog pattern present in each.

Milestone — end of Stage 3: the catalog gauntlet: an AI gives you six rapid-fire scenarios; for each you name the pattern, the reason, the when-NOT warning, and the cost/complexity profile

— under five minutes apiece. This exact exchange, at this exact speed, is the middle third of a real architect interview.

STAGE 4 — MIGRATION AND THE REAL WORLD (Weeks 21–26)

Greenfield design is the minority of the job. Most architecture happens in companies that already exist — with server rooms, ancient software the business depends on, contracts, and laws. This stage is that world, and it’s where “lead migration and modernization initiatives” — a bullet in nearly every JD we studied — gets taught.

Module 10 (Weeks 21–23): Migration — moving a company to the cloud

The 7 Rs — the shared vocabulary of every migration conversation. For each workload in the estate, you choose one:

R	Meaning	When	Effort / payoff
Retire	Turn it off — nobody actually uses it	Every estate has 10–20% of these; find them first	Trivial / instant savings — the best R
Retain	Leave on-prem, for now	Latency-bound, compliance-pinned, or end-of-life-soon systems	None / defers cost — honest “not yet”
Rehost (“lift and shift”)	Move to cloud VMs as-is	Speed matters, apps are stable, skills are thin	Low / quick exit from the data center, but little cloud benefit yet — a first step, not a destination
Relocate	Move at the hypervisor level (e.g., VMware fleet to VMware-on-cloud)	Big virtualized estates on a deadline	Low / fastest bulk move; a waypoint
Repurchase (“drop and shop”)	Replace with SaaS	Undifferentiated apps — email, HR, CRM	Low-medium / whole categories leave your estate
Replatform (“lift, tinker, shift”)	Small upgrades in flight — self-managed DB → RDS, app → containers	The pragmatic middle: real benefit, bounded risk	Medium / the workhorse R of most migrations
Refactor / re-architect	Rewrite for cloud-native (Stage 3 patterns)	Core differentiating systems whose limits hurt the business	High / highest payoff — spend it on the few workloads that

R	Meaning	When	Effort / payoff deserve it
---	---------	------	-------------------------------

The migration method: assess – mobilize – migrate. **Assess:** inventory everything (discovery tools plus the archaeology of asking people), and score each workload on business value and migration difficulty — outputs: an application inventory with an R per row, and a **business case** (TCO of staying vs. moving; be honest that the bill *rises* during the overlap period when both estates run — leaders who aren’t warned about the migration bubble become leaders who cancel migrations halfway). **Mobilize:** build the **landing zone** — the pre-built, governed cloud foundation *before* the first workload lands: multi-account structure (separate accounts per environment and team, so blast radii stay small), centralized identity and logging, the network hub (hub-and-spoke VPCs, the Direct Connect back to on-prem), and **guardrails** — automated policies making the secure, tagged, compliant path the default (AWS Control Tower is the managed starter kit). JDs say “design the landing zone every workload inherits” — this is that. Skipping it to “just start migrating” recreates the untidy data center in the cloud at higher cost; it is the classic failed-migration signature. **Migrate in waves:** group workloads into waves of a few, ordered easy-first: Wave 1 is deliberately low-stakes (learn the machinery where mistakes are cheap), later waves take the crown jewels with rehearsed cutovers, each with a rollback plan and a **hypercare** period of heightened watch. For each database move, the two questions that matter are Module 4’s numbers in disguise: how much downtime may the cutover take (RTO) — and continuous replication with a short switchover exists when the answer is “almost none.”

Vocabulary:

Term	Definition
7 Rs	Retire, retain, rehost, relocate, repurchase, replatform, refactor — the per-workload migration menu. (You’ll also hear “6 Rs” — the older list without relocate.)
Discovery / application inventory	Finding out what actually runs (tools + interviews) / the resulting list, one row per workload, with owner, dependencies, and its R.
Dependency mapping	Charting what talks to what — the reason workloads migrate in groups, and migrations without it fail on day one.
Business case / migration bubble	The stay-vs-move TCO argument / the temporary cost hump while both estates run. Warn about it or be ambushed by it.
Landing zone	The governed, pre-built cloud foundation — accounts, identity, network, logging, guardrails — that every workload inherits. Built <i>before</i> migration.
Multi-account strategy	Separate cloud accounts per

Term	Definition
	environment/team, so billing is attributable and blast radius is contained.
Guardrail	An automated policy that prevents or flags non-compliant actions — governance as machinery, not memos.
Control Tower	AWS's managed service for standing up a multi-account landing zone with guardrails.
Wave plan	The migration schedule in small groups, easy-first, dependencies together.
Cutover / rollback / hypercare	The moment of switching to the cloud copy / the rehearsed undo / the heightened-support window after.

Exercises: (1) The paper migration: an AI generates a fictional 40-server company inventory (you'll meet it again as Capstone 2); assign an R to every row and defend the ten hardest calls. (2) Sketch a landing zone: account diagram, identity flow, network hub-and-spoke, five guardrails you'd enforce from day one. (3) Write the two-paragraph "migration bubble" warning to a CFO — practicing bad-news-early is architect cardio.

Milestone: given an inventory of ~20 workloads with descriptions, you produce a defensible R-per-workload table, a three-wave plan with reasoning, and a landing-zone sketch — in one sitting.

Module 11 (Weeks 24–26): Constraints — law, legacy, and lock-in

Data residency and privacy law as architecture inputs. Personal-data laws — Thailand's PDPA, Europe's GDPR, and their cousins worldwide — share a shape an architect must know cold: personal data needs a lawful basis (often consent); individuals have rights (access, correction, erasure — your design must be able to *find and delete one person's data*, which is hard if you've scattered it across ungoverned copies); breaches carry a **72-hour notification** duty (your logging must let you tell the story in less); and **cross-border transfer rules** constrain where data may physically live — which is a *region-selection* constraint, a *replication-design* constraint (that analytics copy in Singapore may be a legal event), and a reason in-country regions exist. The architect's method, in order: classify the data (what's personal?), map its journeys (every store, every copy, every border crossed — the flows nobody drew are where violations live), then design controls: residency-compliant regions, encryption, retention schedules, and a deletion path that actually reaches backups' expiry. Say "data protection by design" — it's the phrase both laws use, and it is literally your job title in a sentence.

Legacy integration. The mainframe billing system is not moving this year, and the shiny cloud app must talk to it. The patterns: an **anti-corruption layer** — a translation service between new and old, so the legacy system's odd data shapes don't leak into and corrupt your new design; the **strangler fig** — route traffic through a façade, then peel functions off the legacy system one by one until, years later, it can be switched off (named for the fig that slowly

envelops a host tree; it beats big-bang rewrites, which fail at a legendary rate); batch bridges and event taps for data that must flow between worlds; and respect for the latency physics of chatty cloud-to-on-prem calls (Direct Connect helps; better yet, design the chatter out). The rule: **contain the legacy, don't catch it** — every new component should be built as if the legacy system were already gone.

Vendor lock-in economics. Every convenient managed service deepens your marriage to one provider; portability (multi-cloud abstractions, self-managed everything) is a real option that costs real money in complexity and forgone velocity. Neither is a sin — the failure mode is not *choosing* lock-in, it's not *noticing*. The architect's tool is an exit-cost estimate written into the design: "Using DynamoDB saves ≈ 2 engineer-years now; switching later $\approx$ a 6-month rewrite of the data layer — we accept this, and here's the interface boundary that would shrink the rewrite." Three sentences, honestly priced, decision recorded (Stage 5's ADRs) — that's mature lock-in management. Multi-cloud *as a strategy* (running the same workload portably on two clouds) is usually the most expensive possible answer and is bought by exactly the organizations whose scale or regulator demands it; multi-cloud *as a fact* (different workloads on different clouds by history or fit) is normal life.

Vocabulary:

Term	Definition
PDPA / GDPR	Thailand's and Europe's personal-data protection laws — the template pair for privacy-law constraints worldwide.
Data protection by design	Building privacy controls into the architecture from the first sketch — the legal phrase and the architect's duty.
Lawful basis / consent	The legal justification required for processing personal data.
Right to erasure	An individual's right to have their data deleted — which your design must make <i>possible</i> .
Data flow mapping	Charting every store, copy, and border-crossing of a category of data. Where the undrawn flows are, violations live.
Cross-border transfer	Personal data leaving the country — regulated; makes replication design a legal question.
Anti-corruption layer	A translation component that keeps a legacy system's quirks from leaking into a new design.
Strangler fig	Modernizing by routing through a façade and replacing the legacy system piece by piece until it can be switched off.
Big-bang rewrite	Replacing a system all at once, cut over on

Term	Definition
	one day. Legendary failure rate; the strangler fig exists because of it.
Exit cost / switching cost	The realistically-priced cost of leaving a provider or service — the number that turns lock-in from fear into economics.
Multi-cloud (strategy vs. fact)	Deliberately running portably on multiple clouds (expensive, rarely justified) vs. simply having workloads on several (normal).

Exercises: (1) Data-flow-map ThaiTicket’s customer data: every store, every copy (don’t forget logs, backups, the analytics pipeline, the support team’s exports), every border; then write the erasure path. Feel how the map finds problems the diagram hid. (2) Design the strangler-fig plan for a 20-year-old on-prem inventory system, first three peels named. (3) Write the DynamoDB-style lock-in paragraph (benefit now, exit cost later, accepted or mitigated) for three services you’d actually use.

Milestone — end of Stage 4: the constraints gauntlet: one design brief laced with all three (“Thai insurer, customer data, mainframe policy system, board nervous about AWS dependence”) — you produce a one-page approach touching residency, integration, and lock-in, each with a named pattern and an honest cost. When the constraints excite you more than the greenfield would — because constraints are where architects out-earn diagram-drawers — Stage 4 is done.

STAGE 5 — THE CRAFT OF THE ARCHITECT (Months 7–12)

Everything so far made you able to *design*. This stage makes you able to *work as an architect* — the documents, reviews, presentations, and proofs that the role is actually made of, plus the certifications that get you past HR, and three capstones that become your portfolio.

Module 12 (Months 7–8): The deliverables — ADRs, diagram sets, and running reviews

Architecture Decision Records (ADRs). An ADR is a one-page document capturing one significant decision: what we chose, what we didn’t, and why — written *when the decision is made*, numbered, and kept in the project’s repository forever. Why the JDs name them: in two years, someone will ask “why on earth is this DynamoDB?” and the ADR answers in thirty seconds — with the context, the constraints, and the alternatives that were honestly considered. Teams with ADRs relitigate nothing; teams without them argue in circles annually. The template — memorize it:

```
# ADR-014: Use DynamoDB for the session store
Status: Accepted           Date: 2026-08-13
Context:   What situation forced a decision? (Load, constraints,
```

deadlines – the facts.)

Decision: What we chose, in one sentence, active voice: "We will..."

Options considered: 2–3 real alternatives, each with honest pros/cons – including the one you rejected reluctantly.

Consequences: What becomes easier; what becomes harder; the risks we accept; the exit cost.

Write one ADR per significant decision in every capstone. An interview portfolio containing real ADRs is rare and devastatingly effective.

The diagram set — one system, three zoom levels. Real architecture documentation is a *set* (the C4 model popularized this discipline): the **context diagram** — the system as one box, with its users and the external systems it touches; the executive/new-joiner view, and the one you'll present to leadership. The **container diagram** — zoom in: the major running pieces (web app, API, database, queue, cache) and how they talk; the everyday engineering view, roughly your Stage 1–3 drawings. The **deployment diagram** — where it all physically runs: region, AZs, VPC, subnets, scaling groups; the view for reviews, security, and operations. One system, three audiences, three diagrams — kept current, dated, in version control next to the ADRs. A diagram that can't be found or trusted is a diagram that doesn't exist.

Running a Well-Architected review. You know the six pillars (Stage 2); here is the meeting itself. **Before:** pick the workload and scope, get the diagram set current, invite the people who *operate* the thing (not just its designers), and set the tone — this is a health check for the team's benefit, not an audit for anyone's file. **During (half a day):** walk the pillars with the framework's question sets (AWS publishes them, with a free Well-Architected Tool in the console that structures the whole exercise —

<https://docs.aws.amazon.com/wellarchitected/latest/framework/welcome.html>); for each answer, note the finding, and *prioritize as you go* — a handful of high-risk issues, not a hundred nitpicks. Your questioning tone is the review: "help me understand what happens when the payment provider times out" opens doors that "you don't handle timeouts?" slams.

After: a short report — top findings, each with risk, effort, and a recommendation — and the improvement items land in the team's real backlog with owners, or the review was theater.

Practice: run a full review against your own Capstone 1 design; finding your own flaws in a structured way is the fastest-compounding skill in this course.

Exercises: (1) Back-fill ADRs for five big decisions in your Stage 2–3 journal designs — you'll find at least one you can no longer justify, which is the lesson. (2) Produce the full three-diagram set for ThaiTicket. (3) Run the mock review above, produce the one-page findings report.

Milestone: a stranger (or an AI playing one) can pick up your ThaiTicket pack — three diagrams + five ADRs — and correctly answer "what is this, how does it run, and why is it built this way?" without you in the room. That pack *is* the deliverable of the architect's job.

Module 13 (Months 8–9): The human craft — presenting, costing, pushback, and mentoring

Presenting to executives vs. engineers — one design, two languages. Executives buy outcomes; engineers buy mechanisms. To executives: lead with the decision and the number

(“This design supports the million-customer target at \$1.90 per order, and here are the two decisions I need from you”); one slide, the *context* diagram, risks framed as business risks (revenue, compliance, reputation), and never say “Kubernetes” when “the platform” will do. Prepare for the three questions executives always ask: *what does it cost, what could go wrong, why not the cheaper option?* To engineers: lead with the problem and constraints *before* the solution (engineers who feel the problem accept the solution; engineers handed a verdict hunt for flaws on principle), show the container and deployment diagrams, name the trade-offs and rejected options yourself (credibility comes from what you admit), and leave real room to change the design — the questions in the room are free review. The worst architect habit is one deck for both audiences; the best habit is writing the executive summary *first*, because if the design can’t survive compression to five sentences, it isn’t finished.

Estimating costs for a proposal. The method: decompose the design into its ~10 cost-bearing components; price each at expected load in the pricing calculator; **state your assumptions in writing** (requests/day, data growth, egress — the assumptions are the estimate; when they change, the number moves with a clear conscience); apply the reservation strategy to the steady parts; add growth scenarios (today, 2×, 10× — executives remember the 10× number); and present as a *range with drivers* (“\$55–70k/month, driven mainly by egress — here’s the lever”), never a false-precision single number. Include the migration bubble if there is one. Under-estimating to win approval is the classic junior-architect sin; the project remembers.

Handling pushback. You will be pushed on cost (“too expensive” → return to the triangle: “we can cut \$30k by accepting single-AZ — here’s the outage math; your call, and I’ll write it down” — making the trade explicit and *recorded* converts most pushback into either agreement or an informed accepted risk, both fine); on taste (“engineer prefers a different stack” → steel-man it aloud, then take it to criteria, not preference: fit to NFRs, team skills, ecosystem, exit cost — and when it’s genuinely close, let the implementing engineer’s preference win, cheaply buying commitment); and on authority (“the CTO’s friend says use X” → never fight opinion with opinion; ask for the criteria, run X through the same public evaluation as everything else, let the matrix answer politely). And when you’re wrong — you’ll design something that fails; every architect does — the move is the blameless post-mortem applied to yourself, in public, with the ADR updated. Nothing builds decade-long credibility faster; nothing destroys it faster than defending a corpse.

Mentoring engineers. The JD phrase is “mentor the engineers who build against your standards,” and the working forms are: **design review office hours** (a standing door, so guidance happens before code, not after); **reviewing their designs kindly** — questions before verdicts, and always the *why* behind the standard (a guardrail explained recruits an ally; a guardrail imposed recruits a workaround); **delegating real decisions** with a safety net (“you own the cache design; here are the constraints; I’ll review, and I’ll back your call”); and **teaching in public** — every ADR, every review writeup, every brown-bag session scales you beyond your own hours. The blunt career truth: a solo genius architect caps out; an architect who has grown five engineers into design-capable colleagues runs the practice.

Exercises: (1) Present ThaiTicket twice — a 5-minute executive version and a 20-minute engineering version — record both, and listen for jargon leaking into the first. (2) Produce the full written cost proposal (assumptions, range, drivers, 2×/10× scenarios). (3) Pushback theater with an AI: three rounds — cost attack from a CFO, stack preference from a senior engineer,

CTO's-friend authority play — journal what worked. (4) Review a junior's design (an AI can generate a flawed one) in writing, questions-first, and have the AI grade your tone.

Milestone: the double-header: deliver the executive pitch and survive twenty minutes of hostile-but-fair mixed Q&A (AI panel: one CFO, one skeptical engineer) without jargon-crossing, defensiveness, or a single unwritten trade-off.

Module 14 (Months 9–12): Certifications and the three capstones

The certification path, with honest timelines. Certifications don't make you an architect — the preceding thirteen modules do — but they get you interviews, and preparing for them wires the provider's services into your fingers. The ladder, at one-hour-a-day pace: **AWS Certified Solutions Architect – Associate (SAA-C03)** — 2–3 months; the most-requested architect credential in job ads, and after Stages 1–3 much of it will feel like revision with service names attached; take it first. **AWS Certified Solutions Architect – Professional (SAP-C02)** — 4–8 months after the Associate; scenario-based, genuinely hard, and the single strongest resume line in this field; Stage 4's migration and multi-account material is half its syllabus. **Azure AZ-305 (Azure Solutions Architect Expert)** — add if your market is Microsoft-heavy (most enterprise markets are at least bilingual); expect 2–3 months with AWS knowledge transferring at a steep discount (start at

<https://learn.microsoft.com/en-us/credentials/certifications/azure-solutions-architect/>).

TOGAF Foundation — optional, 2–4 weeks; some large enterprises ask for it; it certifies enterprise-architecture *method* and vocabulary rather than cloud skill. Sequence for this course: SAA study alongside Months 7–8, exam at ~Month 8; then either SA Pro (deep-technical path) or AZ-305 (breadth path) through Month 12, with SA Pro completed by Month 14–16 at the honest pace.

The three capstones. Each is a complete, portfolio-grade pack: three-level diagram set, 5+ ADRs, cost estimate with assumptions, and a self-run Well-Architected review with findings. Spend 3–4 weeks each. These three artifacts, plus your journal, *are* your interview evidence of “end-to-end solution design.”

Capstone 1 — the greenfield: a Thai e-commerce platform for 1M users. Brief: marketplace, 1M registered users, 50k concurrent at flash-sale peak, mobile-first, PromptPay-style payment integration, PDPA-compliant, 99.9% availability, seed-stage budget-conscious. Must include: pattern choice with evolution path, caching strategy, RTO/RPO conversation written as dialogue, unit-economics estimate, data-flow map for personal data.

Capstone 2 — the brownfield: migrating a 40-server on-prem company. Brief: a Thai logistics firm; 40 servers (ERP on Oracle, 12-year-old warehouse-management app, file shares, AD, various department apps — generate the full inventory with an AI and freeze it); data center lease expires in 14 months; board wants out. Must include: full 7-R inventory table, three-wave plan with dependency reasoning, landing-zone design, migration-bubble cost timeline, and the CFO memo.

Capstone 3 — the hard one: multi-region DR for a fintech. Brief: a payments company regulated to RTO ≤ 15 minutes / RPO ≈ 0 for the ledger, with data-residency constraints on customer data and an annual regulator-witnessed failover test. Must include: active-passive

vs. active-active analysis with costs, replication design with the residency map, the failover runbook, and the drill plan.

The self-assessment rubric (apply to each capstone, honestly, in your journal):

Dimension	1 — Not yet	3 — Solid	5 — Hire this person
Requirements	Jumped to solution	NFRs stated and traced to design choices	Trade-off triangle positioned, conflicts surfaced to “the business,” decisions extracted
Design quality	SPOFs remain; pattern misfit	Correct patterns, multi-AZ, failures handled	When-NOT reasoning shown; evolution path; simplest design that meets requirements
Six pillars	Pillars ignored	Each pillar visibly addressed	Self-run review found real flaws — and the design was revised in response
Cost	No numbers	Itemized estimate, stated assumptions	Range with drivers, unit economics, 10× scenario, reservation strategy
Documentation	Diagram only	Three-level set + ADRs, correct notation	A stranger could answer “what/how/why” from the pack alone
Communication	One artifact for all audiences	Exec and engineer versions exist	Five-sentence summary that survives; pushback pre-answered in writing

Score every dimension 4+ on all three capstones — revising until you do — and you have completed the course.

Ten architect interview questions (with strong answers)

Practice these aloud; the strength is in the *structure* of each answer, which you now own.

1. “Design a URL shortener / ticketing site / photo app for a million users.” Strong shape: requirements first, out loud (“Reads dominate writes? Availability target? Budget?”) — position on the triangle → three-tier or serverless-first skeleton with multi-AZ, cache, CDN — name the

trade-offs unprompted → finish with cost order-of-magnitude and the 10× evolution path. Interviewers pass the *questions you ask*, not the boxes you draw.

2. “When would you choose NoSQL over a relational database?” “I default to SQL — correctness guarantees and universal skills — until a named reason overrides: extreme horizontal scale, flexible schema, or single-digit-millisecond key-value access. Then I pick the NoSQL flavor to fit the access pattern, and I write the ADR including what we give up: joins, and some consistency semantics.”

3. “Explain RTO and RPO, and how they drive design.” Define both crisply — “they’re business decisions with exponential price tags” — the ladder: nightly backups → continuous replication → warm standby → active-active, with cost rising at each rung → “my job is making the business choose knowingly, then designing exactly to the number — and drilling it, because an untested failover is a hope.”

4. “Monolith or microservices?” The Module 8 ruling, verbatim: start modular-monolith; extract when team-scaling or load-scaling pain actually arrives; microservices are a team-scaling tool that charges a distributed-systems tax. Bonus points for “I’d rather run a good monolith than a bad distributed system.”

5. “How do you handle a large cloud bill / cost optimization?” “Visibility first — tagging and showback; then the harvest in order of effort: kill dead resources, rightsize from measurements, tier storage, schedule non-prod sleep, then reserve the steady baseline. And I report unit economics, not totals — a growing bill with falling cost-per-order is success, not a problem.”

6. “How would you migrate a legacy on-prem application?” “Assess before moving — inventory, dependencies, and an R per workload from the 7 Rs; build the landing zone before wave one; waves easy-first with rehearsed cutovers and rollback plans; and for the crown-jewel database, replication-based cutover sized to the downtime the business signed. Also: the honest migration-bubble warning up front.”

7. “How do you secure a cloud architecture?” Walk the layers: identity (least privilege, MFA, no daily root) → network (private subnets, segmentation, minimal attack surface) → data (encryption at rest/in transit, classification, residency) → detection (audit logs, alerts) → “and by design, not bolted on — the cheapest security control is the architecture decision that removes the risk entirely, like never touching raw card data.”

8. “Tell me about a design decision you got wrong.” They’re testing ego, not history. Shape: real (capstone) example → what you believed → what reality said → the post-mortem, the updated ADR, the pattern you now check for. An architect who can’t produce this answer is an architect who hasn’t been reviewed.

9. “How do you explain a complex technical decision to a non-technical executive?” “Decision and business number first, mechanism only on request; context diagram, not container; risks in revenue-compliance-reputation terms; and I bring the two decisions I need from them, written as choices with price tags — executives decide between options; they don’t approve mysteries.”

10. “An engineer strongly disagrees with your design. What do you do?” “First I steel-man them out loud — they may be right, and the review that changes my design is the review working. If it’s genuinely close, criteria decide (NFR fit, team skills, exit cost), not seniority — and I let the implementer’s preference win ties, because commitment is cheap at that price. Either way the decision and the rejected option go in the ADR, so we never argue it twice.”

The one-page syllabus

When	Module	Focus	External proof
Weeks 1–2	1	Cloud refresher; trade-off triangle; NFRs	Design Journal begun
Weeks 3–4	2	Compute/storage/database/network as decisions	—
Weeks 5–6	3	Reading & drawing diagrams; 3-tier fluency	15-minute whiteboard milestone
Weeks 7–8	4	Reliability: multi-AZ, auto-scaling, RTO/RPO	—
Weeks 9–10	5	Security: least privilege, zero trust, segmentation	—
Weeks 11–12	6	Performance & cost: caching, CDN, reservations, unit economics	Priced design in calculator
Weeks 13–14	7	Ops excellence & sustainability; six-pillar drill	Mock six-pillar review
Weeks 15–17	8	Patterns: monolith/microservices, event-driven, serverless	—
Weeks 18–20	9	Patterns: data lake/warehouse, multi-region, hybrid	Catalog gauntlet
Weeks 21–23	10	Migration: 7 Rs, waves, landing zones	Paper migration
Weeks 24–26	11	Constraints: PDPA/GDPR, legacy,	Constraints gauntlet

When	Module	Focus	External proof
Months 7–8	12	lock-in economics ADRs, diagram sets, running Well- Architected reviews	ThaiTicket pack; SAA-C03 exam (2–3 mo prep)
Months 8–9	13	Presenting, proposal costing, pushback, mentoring	Recorded double- header
Months 9–12	14	Capstones 1–3	Portfolio complete; SA Pro (4–8 mo) or AZ-305 underway

A closing word from your teacher. Twenty-six weeks in, you could design; twelve months in, you can *practice* — there is a difference, and it is the difference this course was built to close. The habits are now the career: draw before you argue, write the ADR the day you decide, price what you propose, name the trade-off before anyone asks, and grow the engineers around you until your standards outlive your presence in the room. Architecture is the rare technical job that gets *better* as you age into it, because its raw material is judgment, and judgment compounds. Keep the journal going. Forty designs from now, you won't need this course — you'll be correcting it.

Sources

Job descriptions and role definitions used for the requirements mapping (retrieved August 2026): the [KORE1 Cloud Architect job description template](#) · the [4 Corner Resources Cloud Architect job description](#) · Microsoft's [Solution Architect responsibilities in the Azure Well-Architected Framework](#) · Arpio's [Pre-Sales Solutions Architect – AWS posting](#) · Intel's [Solution Architect – Enterprise posting \(via Built In\)](#) · Halliburton's [Cloud Domain Architect posting](#). Certification prep-time estimates: [CBT Nuggets' SAA-C03 study-time survey](#) and [Whizlabs' SAP-C02 preparation guidance](#); certification definitions from [Microsoft Learn \(AZ-305\)](#) and the [AWS Well-Architected Framework](#). A companion volume to *The Cloud Leader Course* in the B4LCILC series.

NEW LESSONS — AUGUST 2026 EDITION

In August 2026, new lessons were added to the three interactive courses at b4lcilc.org — the Cloud Leader, Cloud Engineer, and Cloud Architect courses. This addendum brings the reading edition up to date with faithful condensations of that new material, in the same voice as the rest of this book.

A word on what you are holding. These pages are the reading edition: the ideas, the metaphors, and the sentences worth saying out loud. The full interactive versions online carry considerably

more — each new lesson ships with quizzes that check your understanding before letting you move on, hands-on labs (including a practice terminal that behaves like the real one but cannot cost money), videos, vocabulary drills, and — for the capstone material — guided design exercises with requirement checklists and worked model answers. Read these condensations to get the shape of each idea into your head; then, when you can, sit the interactive versions to make the knowledge stick.

The additions fall into three groups. For the **Cloud Leader** course: a gentle introduction to the terminal, Thailand's own privacy law, a leader's guide to AI proposals and AI bills, and a capstone exercise that assembles the entire course into one page. For the **Cloud Engineer** course: serverless computing — the pattern that turns small operational scripts into functions that cost nothing to run. For the **Cloud Architect** course: the arithmetic of availability promises, the Retrieval-Augmented Generation pattern that customers now ask about weekly, and a full costing exercise — pricing a design before anyone builds it.

For the Cloud Architect course

The arithmetic of nines — SLAs, SLOs and error budgets

Availability is quoted in nines, and the first thing an architect does with a nine is convert it into minutes — because minutes are what the business can feel. A 30-day month contains 43,200 minutes:

Availability	Downtime allowed per month
99.9% (“three nines”)	about 43 minutes
99.99% (“four nines”)	about 4.3 minutes
99.999% (“five nines”)	about 26 seconds

Five nines — the number executives like to say aloud — is beyond any human-operated recovery; failover must be automatic or the promise is fiction. Each additional nine costs about ten times more to engineer; converting into minutes is how you make a stakeholder choose knowingly.

Three acronyms carry the conversation. A **Service-Level Indicator (SLI)** is a measurement. A **Service-Level Objective (SLO)** is the internal target on that measurement. A **Service-Level Agreement (SLA)** is the external contract, with money attached to a miss. The discipline: the SLO is always stricter than the SLA — you aim inside the line you are paid to hold — and the SLI measures what users experience: a server that is up while returning errors is down in every sense that matters.

The error budget makes SLOs operational: the gap between perfection and the objective, treated as a resource to spend. A 99.9% SLO grants 43 minutes of failure a month; deployments and experiments spend from it; when it is exhausted, the team stops shipping and works on reliability — ending the oldest argument in software, ship versus stability, by replacing opinion with arithmetic.

Now the trap: availability multiplies along a chain. Three services in series, each honestly at 99.9%, compose to $0.999 \times 0.999 \times 0.999 \approx 99.7\%$ — roughly 130 minutes a month, three times what any single service admits to. Redundancy works the other way: two independent 99.9% instances in parallel fail together only $0.1\% \times 0.1\%$ of the time — why multi-zone is such good value. The interview-grade sentence: *you cannot promise more availability than the product of your synchronous dependencies, so either shorten the chain or lower the promise.*

Architecture in the age of AI — RAG and its relatives

By 2026 the pattern catalog has a new entry that customers ask about weekly: what it is, when it earns its keep, and when it will embarrass you. A **Large Language Model (LLM)** API — Amazon Bedrock, Azure OpenAI Service — is, architecturally, *an expensive, slow, stateless dependency*. It is metered per **token** — a word fragment of roughly four characters. It answers in seconds, not milliseconds, remembers nothing between calls, and is occasionally confidently wrong — a component with unusual failure modes, deserving the same timeouts, retries, circuit breakers, and budget alarms as any dependency, because a runaway loop of calls burns real money by the hour.

The model does not know your documents; the standard answer is **Retrieval-Augmented Generation (RAG)**: instead of teaching the model your data, you *look up* relevant facts at question time. Documents are cut into chunks, each converted to an **embedding** — a list of numbers positioning its meaning in space — stored in a **vector database** (pgvector on AWS, AI Search on Azure), which returns the nearest chunks. Those chunks are pasted into the prompt: the model is not consulting your data; it is summarising the excerpts you fetched for it.

On the whiteboard, RAG is two pipelines: an **ingestion path** running ahead of time (documents chunked, embedded, stored) and a **query path** running per request — two existing catalog patterns, which is why architects pick this up in an afternoon.

The trade-offs are the standard three, sharpened: cost (every query costs real money — cache repeats, route easy questions to a cheaper model), latency (seconds, not milliseconds — stream the answer), and privacy (your documents leave your boundary; if the vector database ignores document-level access control, the chatbot will cheerfully summarise the salary spreadsheet for anyone who asks). Know when *not* to build it: many “we need a chatbot” requests are a search box wearing fashion, and exact, auditable answers — sums, balances, ledgers — are the wrong job for a system right 97% of the time. The interview-grade sentence: *RAG is a search problem stapled to a summarisation API.*

Costing a design before anyone builds it

The course’s Pillars of Performance and Cost module now closes with a full design exercise: estimation under fire. The brief: leadership wants a monthly cost estimate for a document-management product before approving the build. The workload — 200,000 registered users, 40,000 active daily, each making about 25 API requests a day and uploading on average 2 megabytes of documents a month, kept for seven years. The design: a load-balanced API tier across two zones, a managed multi-zone PostgreSQL database, object storage, a nightly 50-gigabyte export to a partner, and separate development and staging environments. The task: a one-page monthly estimate in round numbers, with assumptions stated in writing before any

prices appear, the capacity arithmetic shown (users $\rightarrow$ requests per second $\rightarrow$ instance count), and today / 2 $\times$ / 10 $\times$ scenarios.

The exercise's checklist teaches the discipline: price compute from peak load, not average; count the database standby; model storage as growth over seven years with lifecycle tiers; calculate egress rather than guess it — the line most estimates forget and the one that most often dominates; remember NAT (Network Address Translation) gateways, non-production environments, backups, monitoring, and the support plan; apply commitment discounts only to the steady baseline; and end with a range with named drivers plus a unit-economics sanity check — cost per active user per month. A single precise number is wrong the day it is spoken.