SAMPLE

Building Tomorrow's Cloud Architects

The Boundary's Keeper in the Machine Era

Aiman JoeHanz

Free sample: Introduction & Chapter 1

putrabooks.github.io

Buy the full book:

Carousell carousell.sg/p/building-tomorrow-s-cloud-architects-ebook-pdf-1471383527

Ko-fi

Payhip

Contents

Introduction

Chapter 1 — The Boundary

Chapter 2 — The Territories

Chapter 3 — The Path

Chapter 4 — The Working Day

Chapter 5 — The Boundary

Chapter 6 — The Burnout

Chapter 7 — The Conversation

Chapter 8 — The Instruments

Chapter 9 — The Year Whole

Chapter 10 — The Long Path

Introduction

Two people will tell you what a cloud architect does, and both will be describing a cartoon.

The first is the diagram visionary. To them the job is drawing: boxes, arrows, lanes and zones, a design rendered in a tool that photographs well in a slide deck. The system exists as an image, the image is elegant, and the elegance is taken as evidence that the system will work. The second is the console jockey. To them the job is clicking: provisioning what is needed, connecting it to what already exists, and calling the accumulated result an architecture because it is in the cloud and the cloud is presumed to be someone else's problem.

The actual work is neither. A cloud architect decides where one thing ends and another begins, and the decision is a bet about failure. Put two services in the same failure domain and they will fall together. Put them in different ones and you have bought survivability at the price of latency, cost, complexity and a second set of secrets to rotate. Every architectural decision is some version of that trade, and the drawings and the consoles are only how it gets recorded and enacted.

I call the thing being kept the boundary, and the person keeping it the keeper. A boundary in this sense is not a network line or a security group, though it may be implemented with both. It is the answer to a question the diagram never asks: if this goes wrong, what else goes wrong with it? An architect is the person in an organisation who holds that answer, who knows which failures are shared and which are not, and who is willing to insist on the distinction when the schedule argues otherwise.

The trade has three commitments, and unlike most professional mantras these can be checked. Decide what fails together, and make those groups as small as you can afford, because a failure domain you did not name is a failure domain you do not control. Write down what the system will not do, because an architecture is defined by its refusals more truly than by its features, and a design with no stated refusals will grow in every direction until it can no longer be understood by anyone. And leave a path out, because every dependency you adopt is a lease whose renewal is decided by somebody else's roadmap, pricing page and corporate strategy.

Those commitments conflict, which is why the work requires judgment rather than a reference pattern. Small failure domains cost money and multiply the

number of things that must be operated. Written refusals cost the deal that the product team wanted. Exit paths cost engineering effort that produces no visible feature and is impossible to demonstrate in a demo. An architect who has not noticed the conflict is applying a template. An architect who has noticed it, and who can say which price is worth paying in this particular case, is doing the trade.

This book describes that trade. It covers where the work happens and who does it, how people arrive, what the day contains, and what the craft consists of once you stop describing products and start describing decisions. It spends a chapter on exhaustion, which this work produces in a specific and under-discussed form. It spends a chapter on conversation, because architecture is settled in arguments with finance, security, product and vendors rather than in documents. It describes the instruments, which are less exciting than the diagrams suggest and more decisive than the invoices admit. It follows one architect through a year of migration, cost pressure and a failure everyone had assumed away, invented and labelled as such. And it looks at the long arc, which in this trade is short enough to watch and long enough to shape a career.

What the book is not is a study guide. It will not prepare anyone for a vendor certification, and it names no products, because products change faster than books and a named service dates a page within a season. The judgment the certifications test for is real; the specific interfaces they test are perishable. What survives is the ability to look at a proposed system and say where it will fail, what will fail with it, and whether the organisation can afford the answer.

One thing belongs in an introduction because it is what most people get wrong before they start. This is not a senior engineering role in the sense of being the best coder in the room. It is a role about consequences: whose work stops when this dependency does, what the bill looks like at ten times the volume, what the team will be able to operate at three in the morning six months after the architect has moved to another project. The people who thrive are the ones who find those questions interesting rather than tiresome. If you find them tiresome now, you probably will later too, and it is cheaper to learn that here than in the second year of a migration.

Who This Book Is For

Three kinds of reader will find different chapters useful, and it is worth saying which.

The first is someone considering the trade: an engineer who has been building services and wonders whether the architecture role is the next step, or someone from operations wondering the same. For them the useful chapters are the territories, the path and the working day, because those describe the life rather than the discipline. The decision in front of them is about how to spend their attention for a decade, and the honest material for that decision is what the day contains.

The second is a practitioner already in the role, often the only one in their organisation, who suspects that the work has become documentation and wants to know what it looks like when it is done properly. For them the craft chapter and the instruments chapter are the substance, and the conversation chapter is the one that will change their week. Many readers of this kind have no colleague to compare against, and a description of expert practice is the nearest available substitute.

The third is someone outside the trade who has to make decisions that involve it: a manager who has inherited an architecture function, a finance partner who funds one, an auditor who examines one, a technologist in a leadership role who needs to know whether the person across the table is answering the question. For them the territories and the conversation chapters are the useful ones, plus the short passage in the opening chapter on how to read a claim about resilience. That passage is written for readers who will never design anything.

A book that tried to serve all three on every page would serve none of them. The chapters are ordered for the first reader and left navigable for the others. The opening chapter and the craft chapter can be read alone. The path and day chapters do not require the craft chapter, and the craft chapter does not re-argue the definition. A reader who goes straight to what they need will not be missing an argument they require, which is the most a book of this kind can offer.

There is a fourth reader the book is written against, and naming them is fairer than pretending they do not exist. It is the reader who wants the trade to be a certification: a study path, a list of services, a credential that converts experience elsewhere into a title here. That reader exists because the compensation and the demand attract them, and this book will frustrate them, because it deliberately names no products and teaches no interfaces. The frustration is intended. A trade that can be entered without learning to hold a claim about failure is not this trade, and treating it as one produces

practitioners whose first serious incident is also their last.

Chapter 1 — The Boundary

The Two Cartoons

The diagram visionary and the console jockey are opposites in temperament and identical in error. Both treat the architecture as the thing that appears on the surface: in one case a drawing, in the other a running account. Neither asks the question that decides whether the system will survive its first bad day.

You can recognise the visionary by what they produce. The drawing is comprehensive, symmetrical and annotated in a style that suggests a standardised notation. Ask what happens when the message broker in the middle degrades, and the answer describes the broker rather than the consequences. Ask which services will stop and which will continue, and the visionary reaches for the diagram again, because the diagram is where they think. The design has never been tested against a failure, and its elegance is a property of the pen rather than of the system.

You can recognise the console jockey by a different tell. Their environment works, it was provisioned incrementally over years, and it contains several arrangements nobody would design on purpose. Ask why a particular database sits in a subnet of its own, and the answer is a story about a support case. Ask what would happen if a region were lost, and the answer is a shrug, because that question has never had an owner. The system is real and its structure is an accident.

The trade is the space between them. An architect needs the visionary's capacity to see structure and the jockey's respect for what is actually running, plus the one thing neither supplies: a habit of asking what fails together. That habit is teachable, it is not a personality trait, and it is the entire difference between an architecture and a picture.

What a Boundary Is

A boundary is a claim about co-failure. Two things on the same side of it will fail together, and two things on opposite sides can fail independently, at least with respect to the mechanism the boundary addresses. That is a precise definition and it is deliberately narrower than the word usually carries. A boundary does not mean isolation in every respect, and it does not mean two sides cannot affect each other. It means that for a specific class of failure, the boundary contains it.

Because boundaries are failure-class-specific, a system has many of them and

they do not nest neatly. A service may be isolated from a power failure in one building and completely exposed to a bad deployment of a shared library. It may survive a database outage and be destroyed by a pricing change. A network boundary does nothing about a correlated timeout, and a timeout budget does nothing about a shared identity provider. The architect's job is to enumerate the classes that matter and to know, for each, which boundaries exist and whether anyone tested them.

The most valuable boundary in most estates is the one that is hardest to see: the shared dependency that everyone has forgotten is shared. A single identity provider, a single DNS zone, a single container registry, a single build pipeline, a single NAT gateway, a single human who approves production access. These do not appear as boxes on most diagrams because they are not services the team wrote. They appear in the incident, at the moment when a minor problem in an unrelated system takes down everything at once.

Naming a boundary therefore has a second half, which is naming the shared things that cross it. An architect who can produce, for their estate, a short list of the things that would take down more than one service at once, is holding the trade's central artifact whether or not they call it that. The list is uncomfortable because it is usually longer than expected and because several entries are decisions nobody currently owns.

The Three Commitments

The first commitment is to decide what fails together and make those groups small. This is not a preference for redundancy. It is a demand that the grouping be chosen rather than inherited. Most systems acquire their failure domains by accident: a subnet here, a cluster there, a region chosen by whoever set up the account. An architect converts accident into decision, and then argues for the smallest groups the budget will bear, knowing that every split costs money, latency and operational attention.

The second is to write down what the system will not do. This sounds like documentation and is actually a control. A system with stated refusals has a boundary the organisation can reason about; a system without them expands to meet every request and acquires, in the process, a shape nobody chose. The refusals are where an architect's judgment becomes visible and where most of the political difficulty lives, because a refusal is a decision to disappoint someone specific.

The third is to leave a path out. Every dependency is a lease, and the terms of

the lease are set by a party whose interests are not yours. Prices change, features are deprecated, acquisitions happen, and an architecture that cannot be moved is an architecture whose future is decided elsewhere. The exit does not have to be cheap and it does not have to be maintained. It has to exist, be documented, and be believed by the people who would have to execute it.

These three are in tension in every real design, and the tension is the subject of the craft chapter. What matters here is that they are commitments rather than techniques. A technique can be delegated. A commitment has to be held by someone who will be in the room when the shortcut is proposed, and that person is what the word architect is supposed to mean.

Why the Cloud Changed the Work

The architecture trade is older than cloud computing, and the cloud changed it in three ways that are worth separating from the marketing.

The first is that provisioning became a decision rather than a purchase. When hardware took months and cost capital, the architecture was largely fixed by the time anyone could use it, and the interesting work happened upstream. Now a team can create and destroy infrastructure in minutes, which means the architecture is continuously being decided by whoever is holding the credentials. That is a genuine transfer of power, and it is why so many estates are shaped by engineers who never thought of themselves as architects.

The second is that failure became somebody else's responsibility, nominally. A provider guarantees certain things and those guarantees are real and narrow. The narrowness is the trap. A provider can promise a service is available while your configuration makes it unusable, and it can promise durability while your access pattern makes recovery take days. The shared-responsibility idea is honest and is routinely read as a transfer of risk that has not occurred.

The third is that cost became architectural. In a hardware estate, capacity was a capital decision made by someone with a budget, and the architect designed within it. Now the bill is a function of design choices made daily, and an architect who cannot read the bill is designing without seeing one of the primary constraints. This is the change the trade has absorbed least well, and it is why the cost conversation appears repeatedly in the chapters that follow.

What the Trade Is Not

Four adjacent roles are routinely confused with this one, and the confusion does real damage to hiring and to design. Separating them is worth the effort

because each brings a different question to the same room.

The first neighbour is reliability engineering. A reliability engineer asks how the system behaves when it is degraded and how the response is organised. That is a large part of the boundary discipline and it is not the whole of it, because reliability work usually begins from an estate that already exists and has a shape. The architect's question is upstream: whether the shape should have been that. In practice the two roles overlap heavily, and the healthiest teams have both, but a team that hires only the first will keep inheriting shapes nobody chose.

The second neighbour is platform or infrastructure engineering, which builds the substrate. A platform engineer writes the module that everyone deploys through. The architect decides what the module should enforce and what it should permit, which is a smaller amount of code and a larger amount of argument. Conflating the two produces architects who build instead of deciding, and building is more comfortable, so the conflation is common and self-sustaining.

The third neighbour is the senior developer who has been given the title as a promotion. This is the most frequent confusion and the least discussed. A developer who is excellent at constructing a service has demonstrated one set of skills; deciding where that service's failure ends has not been tested. Some make the transition easily and some do not, and there is no reliable way to tell from the quality of their code.

The fourth neighbour is the enterprise architect who owns a standards catalogue. That role is real and it is described in the territories chapter. What distinguishes it is not seniority but instrument: the catalogue, the review board and the roadmap, rather than the estate's actual failure domains. An organisation that expects a standards function to hold the boundary will discover during an incident that nobody was ever assigned the job.

How to Read a Claim About Resilience

Outside this trade, and frequently inside it, resilience is asserted rather than demonstrated, and a reader needs a way to test a claim quickly. Four questions do most of the work and none of them requires technical knowledge to ask.

The first is what exactly is supposed to survive. Surviving a failure means different things: no data loss, no service interruption, degraded function, or a

stated recovery time. A claim that says the system is highly available has said nothing until one of those is named. Ask which, and ask what the stated number is.

The second is what the survivor depends on. Almost every resilience claim resolves to a second thing that must work, and the interesting question is what that second thing is and whether it shares a failure domain with the first. Three zones inside one account, one identity provider and one pipeline is a configuration that reads as resilient and behaves as a single unit.

The third is whether it has been tested. A failover that has never been exercised is a hypothesis. Ask when it was last performed, by whom, and what happened. An organisation that cannot answer has a design belief rather than a capability, and the belief will be tested by an event rather than by a rehearsal.

The fourth is who is woken, and what they are expected to do. Every resilience mechanism has a response attached to it, whether or not anyone wrote it down. If the response requires a specific person, a specific credential or a specific console, then that requirement is where the claim fails, and it fails exactly when the estate is least able to absorb it.

These four questions also function as a design checklist, which is why they belong early in a book about the trade. A practitioner who can answer them about their own estate is doing the work. One who cannot has a diagram, and the diagram is not the thing that will be tested.

What Happens When the Boundary Fails

The abstract case for the discipline becomes concrete when you count what a crossed boundary actually costs, and the costs come in four kinds that a practitioner should be able to name separately.

The first is direct loss: work that stopped, revenue that did not arrive, transactions that had to be replayed by hand. It is the easiest to count and it is usually the smallest. Practitioners consistently overestimate the direct cost and underestimate the others, because the direct cost is the one that appears in an incident report.

The second is the cost of the recovery itself. Restoring an estate after a correlated failure consumes weeks of the team's attention that was budgeted for building, and the attention is spent by the people most able to improve the system. Projects slip, the drift that accumulates during the recovery is never

cleared, and the estate emerges weaker than it went in, which is the definition of a slow compounding loss.

The third is the loss of confidence, inside and outside the organisation. The finance partner who funded a redundancy argument and watched a shared dependency take everything down will fund fewer arguments next year. The operations team that was blamed for a design fault will be slower to escalate, because escalation is what got them blamed. These effects are real, they are durable, and they are the reason a single bad incident can set an estate's maturity back by years.

The fourth is the cost to the people, and it appears in no ledger at all. The engineer who approved the change, the architect who raised the concern and was overruled, the responder who could not fix it because a credential was in a safe nobody could open. The trade's casualties are made in these moments, and the way an organisation handles them determines whether it keeps the practitioners who would have prevented the next one.

A Note on Judging Your Own Estate

A reader who works in this space can assess their own situation against a small number of questions, and the answers are more informative than any maturity model.

Can you name, without looking, which components would fall together? Not the architecture's features: the shared items underneath them. If the answer requires research, the estate is not being held by anyone, whatever the org chart says.

Can you state what your system refuses to do? If the answer is a list of capabilities rather than a list of refusals, the design has no boundaries, only features. Systems defined by their capabilities expand until they cannot be understood, and the expansion is always justified case by case.

Do you know what a failure costs, per unit? Not the annual bill: the marginal cost of the thing you are protecting. An estate without unit economics has no way to argue for resilience that is proportionate rather than aspirational.

Has a recovery procedure been rehearsed in the last year by someone who did not write it? If not, the organisation has a plan rather than a capability, and the plan will be discovered to be a document at the moment it is most needed.

Who would be called if you were unavailable for a month? If the answer is

nobody, the estate's boundary knowledge is a concentration risk, and the risk is the same in kind as a single shared provider with the difference that it can be fixed in an afternoon.

None of these questions requires a tool or a budget. All of them can be answered this week, and the pattern of answers is a better account of an estate's real condition than any diagram, scorecard or certification it holds. The trade exists to keep those answers current, and a practitioner who cannot answer them about their own estate is not yet doing the work.

The Keeper

What the keeper holds is not a system and not a document. It is the truth about what fails together. That truth is inconvenient, it is expensive to act on, and it is the thing an organisation is buying when it hires for this role, whether or not it has articulated the purchase.

Holding it produces a particular kind of loneliness that this book will return to. The architect is often the person who knows that the promised resilience is not real, that the migration will not finish on schedule, that the shared dependency list has an entry nobody is fixing. Saying so costs credibility in the short term and is the job. The practitioners who endure are those who learn to say it early, in writing, with a number attached, rather than late and in alarm.

A reader deciding whether this is for them should test one thing. When you imagine a system, do you find yourself wondering what breaks first, and what breaks with it? If that question is the interesting part, this is the trade. If it feels like the tedious part before the real work, the trade will be a poor fit, and there are many adjacent roles where the building is the whole of the job.

ABOUT THE AUTHOR

The cloud architect's volume of the Building Tomorrow's People series is about boundaries: where one system ends, another begins, and who pays when the seam fails. Aiman JoeHanz follows the architecture through the migration, the bill that arrives a month after the design, the identity boundary nobody documented, the region that goes dark, and the abstraction that hides a machine someone still has to operate.