

E-BOOK - FERNANDO F. AZEVEDO

Inside the Bank: The Architecture Elevator

From business to the ledger, from events to AI, financial systems architecture on AWS, for architects and developers who ride the elevator between strategy and code.

Fernando F. Azevedo

1st edition - 2026 - 16 chapters - ~119 pages

PART I

The Architecture Elevator

Gregor Hohpe's mental model applied to banks: why the architect rides between the executive floor and the engine room, and what's lost when they stay stuck on a single floor.

01 - Why the architect rides the elevator

The penthouse talks strategy, risk and revenue. The engine room talks idempotency, partitions and latency. Architecture is the elevator that connects them, and in a bank, whoever doesn't ride it decides in the dark.

In every large bank there is a silent chasm between those who set strategy and those who write the code that executes it, and that chasm is expensive, in money, in reputation, and sometimes in operating licenses. The modern architect is neither the best programmer in the room nor the most articulate executive on the committee: they are the person who rides between those two worlds without losing fluency in either. This book starts here because everything that follows, ledgers, events, security, AI, platform, only makes sense once you understand why the elevator exists and why, in banking, it jams with a frequency no other industry would tolerate.

The corporate building and the elevator nobody maintains

Gregor Hohpe describes the modern enterprise as a multi-story building. In the penthouse live the executives: they speak of competitive strategy, risk appetite, regulatory positioning, product revenue, and customer satisfaction. In the engine room, in the basement, live the engineers: they speak of message queues, database locks, network latency, asymmetric-key cryptography, and deployment windows. Between those two extremes lie dozens of intermediate floors, product managers, tech leads, business analysts, compliance teams, each with its own vocabulary, its own incentives, and its own partial view of the system.

The problem is not the vertical distance itself. The problem is that the elevator is broken. Decisions descend from the penthouse as PowerPoint slides full of intent and empty of technical constraint. Information rises from the engine room as incident tickets and capacity reports that nobody in the penthouse knows how to read. Along the way, each floor filters, mistranslates, and adds noise. The result is predictable: the strategy that arrived downstairs is no longer the one conceived upstairs, and the system built downstairs is no longer what the business needed.

The senior architect is, in Hohpe's definition, the person who spends their career in that elevator. Not because they enjoy meetings, nobody does, but because they understand that the only way to ensure a strategic decision produces the correct technical effect, and that a technical limitation is understood as business risk before it becomes an incident, is to be present on both floors with enough credibility to be heard on each. That is the job. It is not glamorous. It is essential.

Sixteen years translating between floors: Over more than sixteen years working on financial systems, from card processors to digital banks, from brokerages to instant payment infrastructures: I learned that the most dangerous moment in any project is not when the team does not know the technical answer. It is when the technical team and the business team sincerely believe they are talking about the same thing and they are not. An executive says 'we need high availability' and imagines the system never goes down. The engineer hears 'high availability' and thinks of a 99.9% SLO with automatic failover. Those two worlds are compatible,

but they are not identical, and the gap between them, when left unstated, becomes an incident at 2 a.m. on a Friday before a long holiday weekend. My job, the whole time, was to ride that elevator carrying context in both directions: taking engine-room constraints to the penthouse before they became surprises, and taking penthouse intentions to the engine room before they became wrong systems.

Why banks suffer more when the elevator jams

Every large organization has the jammed-elevator problem. But banks pay a disproportionate price when it fails, for three reasons that do not exist with the same intensity in any other industry.

First: real money has no rollback. When an e-commerce site has a pricing bug, it cancels the orders, issues a statement, and moves on. When a bank credits the wrong amount to thousands of accounts, the legal, accounting, and regulatory problem can last years. The irreversibility of financial transactions means that a misguided technical decision, about idempotency, event ordering, or eventual consistency, is not technical debt: it is a real financial liability. This will be the central theme of Chapter 6, but it must be established here: in banking, the engine room and the penthouse share the same balance sheet.

Second: the regulatory license is a fragile asset. A bank operates because the Banco Central do Brasil authorized it. That authorization can be suspended, restricted, or revoked. The BACEN does not accept 'we were in the middle of a migration' as justification for control failures. Architecture decisions about segregation of duties, operation traceability, encryption of data at rest and in transit, and operational continuity are not optional technical choices, they are conditions of the business's existence. When the architect does not ride up to the penthouse to explain that a particular design choice creates a compliance gap, they are not being modest: they are being negligent.

Third: trust is the real product. A bank does not sell checking accounts. It sells the belief that your money is safe, that the transaction will complete, that the statement is true. That belief is built over decades and destroyed in hours. A Pix that disappears, a credit limit that vanishes without explanation, an account blocked without notification, each of these events is, in the customer's perception, a betrayal. And each of them has a root cause that lives in the engine room: a race condition, a misconfigured timeout, a queue with no dead-letter. The architect who does not connect those two worlds leaves the bank vulnerable to damage that no hotfix repairs.

Two languages, one single system

The biggest obstacle to the elevator working is not lack of will, it is lack of a shared vocabulary. The penthouse and the engine room speak genuinely different languages, and the temptation to pretend otherwise produces the worst kind of misunderstanding: the kind nobody notices until it is too late.

The table that follows: Two floors, two languages: maps the central concepts of each floor side by side. It is not a curiosity table: it is a working tool. When an executive speaks of 'operational resilience,' the architect needs to know immediately that this translates to RPO, RTO, multi-region strategies on AWS, and data consistency decisions that have measurable cost and complexity. When an engineer speaks of 'P99 latency above 800ms in event processing,' the architect needs to know how to translate that to the penthouse as 'one in every hundred Pix transactions is taking longer than the regulatory limit allows, and that is a risk of fines and BACEN intervention.'

This translation is not simplification. Simplifying is losing information. Translating is preserving the consequence while changing the vocabulary. The executive does not need to know what a dead-letter queue is, but they need to know that without one, transaction messages can be silently lost and will never be reprocessed. The engineer does not need to know the exact cost of a BACEN

fine, but they need to know that the audit field they are thinking of omitting to simplify the schema is a non-negotiable regulatory requirement.

The architect who masters this bidirectional translation is not a passive intermediary. They are a decision multiplier: every conversation they facilitate between floors prevents weeks of rework and, in extreme cases, prevents incidents that cost more than the entire project. In the chapters that follow, every technical decision will be presented with its translation to the floor above, because that is the only form of architecture that actually works in a bank.

What this chapter establishes for the entire book

- The senior architect is defined by the ability to move between the strategic penthouse and the technical engine room with fluency and credibility on both floors.
- In banks, the jammed elevator is not merely organizational inefficiency, it is financial, regulatory, and reputational risk with irreversible consequences.
- The senior architect's central question is not 'which technology to use?' but 'what business risk does this reduce, what capability does it enable, and what commitment does it create for the future?'
- Translating between floors is not simplifying: it is preserving the consequence while adapting the vocabulary to the audience.
- Every chapter of this book will present technical decisions with their explicit translation to the business floor, because architecture without that bridge does not truly exist.

Two floors, two languages

Criterion Penthouse (business) Engine room (engineering)

--- --- ---

Vocabulary Margin, risk, NPS, churn, regulation Latency, idempotency, throughput, SLO

Horizon Quarter, year, positioning Sprint, release, incident

Unit of decision Business capability and investment Service, API contract, event

Main fear Losing market, a regulator fine Waking at 3am because of a deploy

What the architect delivers Trade-off in the language of risk and option An implementable decision with mechanisms

The question that defines the senior architect

There is a question that separates the senior architect from the architect who is still growing, and it has nothing to do with technical knowledge. Exceptional engineers ask: 'what is the best technology to solve this problem?' That is a legitimate and necessary question. But the senior architect asks a different, prior, and harder question: 'what business risk does this decision reduce, what capability does it enable for the bank, and what commitment does it create for the next three to five years?'

The difference is not semantic. When you ask which technology to use, you are looking inward at the system. When you ask what risk it reduces and what commitment it creates, you are looking at the system as an instrument in service of a regulated business that has real customers, legal obligations, and a strategy that will change. That second question forces the elevator to move: it requires you to ride up to the penthouse to understand the business context before descending to the engine room to choose the tool.

In practice, this means that when someone proposes migrating Pix processing to an event-driven architecture with Amazon EventBridge and Amazon SQS, the right question does not start with 'EventBridge or Kafka?' It starts with: 'What operational failure are we trying to eliminate? What is

the regulatory cost of a lost message in this flow? Does this change enable any new product capability, or is it purely defensive? And which teams will need to change how they work for this to function in production?' Only after answering those questions, which live in the penthouse, does it make sense to descend to the engine room and compare the technical properties of the options.

This book is organized around that question. Every chapter begins on the business floor, with the capability, the risk, or the regulatory requirement, and descends to the technical implementation on AWS with trade-offs made explicit. The elevator will go up and down the whole time. Prepare yourself for the ride.

The architect who never leaves the engine room: There is a failure pattern I have seen repeat in banking projects over the years: the technically brilliant architect who never rides up to the penthouse. They produce elegant designs, choose the right technologies, write impeccable ADRs, and deliver a system that the business cannot operate, that compliance cannot audit, and that the product cannot evolve without breaking everything. Not because the design is technically poor. Because it was conceived without the constraints and intentions that only exist on the floor above. In banking, this pattern is especially dangerous because the cost of rework is not just engineering time: it is accumulated regulatory risk, audit debt, and a product that did not reach the market while the competitor's did.

What this book is, and what it is not

This is not an AWS recipe book for banks. It is not a service catalog with financial use cases pasted in. It is a book about how to think like a senior architect in an environment where technical decisions have real regulatory, financial, and reputational consequences, and where the only way to make good decisions is to keep the elevator moving between strategy and implementation. Every chapter will go up and down. Every technical decision will be presented with its business context. And every trade-off will be named explicitly, because in banking, unnamed trade-offs are unmanaged risks.

02 - The anatomy of a bank's floors

Between the penthouse and the engine room there are intermediate floors, product, journey, domain, data, platform. Mapping them is what keeps every conversation from collapsing into 'generic integration'.

Every bank has a penthouse where people talk about risk, margin and regulation, and an engine room where threads, queues and bytes run, but between those two extremes there are at least five floors that most architects never name properly, and that is precisely where projects get stuck. Mapping these intermediate floors is not an academic exercise: it is what separates an architecture conversation from a generic integration meeting that ends without a decision. In this chapter I descend each floor with you, name the three distinctions that get confused all the time, and show what happens when the elevator gets stuck.

My read after 16 years in financial systems: I have participated in hundreds of architecture sessions at banks, from board meetings to incident war rooms. The failure pattern that repeats most is not lack of technology: it is lack of shared vocabulary across floors. The product team talks about 'credit journey', the engineering team talks about 'proposal service', the data team talks about 'table TB_PROPOSTA', and nobody notices that all three are describing different facets of the same business phenomenon. When this happens, each floor builds its own representation of the world and integration becomes the accidental, and most expensive, product

of the project. The diagram I present in this chapter is my number-one alignment tool in any banking engagement.

The building has more floors than you think

Gregor Hohpe's elevator metaphor places strategy at the top and implementation at the base, and that is correct, but insufficient for a bank. A bank is one of the most stratified organizations that exist: Central Bank regulation, board-level risk appetite, C-level product targets, UX-designed customer journeys, business domains managed by stable teams, events carrying auditable facts, data requiring traceable lineage, platforms abstracting infrastructure, and operations guaranteeing SLAs under scrutiny from BACEN and the European Central Bank when overseas branches are involved.

As the diagram below shows, I organize these floors into seven layers: Strategy (risk appetite, competitive positioning, regulatory obligations), Business Capabilities (what the bank knows how to do in a repeatable and measurable way), Product and Journey (how those capabilities are packaged and delivered to the customer), Domains and Events (where rules, decisions and business facts live), Data (lineage, quality, governance and analytical product), Platform and Runtime (what abstracts infrastructure from product teams) and Operations and Security (continuous evidence of reliability and compliance).

Each floor has its own vocabulary, its own artifacts and its own stakeholders. The architect who knows how to move between them, rising to translate a technical decision into risk language, descending to translate a regulatory directive into a design requirement, is the one who delivers real value. The one stuck on a single floor, whether the strategy penthouse or the Kubernetes engine room, loses the ability to influence what matters.

The three distinctions that get confused all the time

Without properly naming the intermediate floors, three confusions take hold and transform any banking initiative into expensive and fragile distributed CRUD.

Capability Screen. A business capability is a function the bank executes in a repeatable way with a measurable outcome: 'Personal Credit Origination', 'TED Settlement', 'Collateral Management'. A screen is an interface that accesses that function. Confusing the two leads to roadmaps that describe UI features and never question whether the underlying capability is healthy. I have seen banks redesign their credit application three times in two years without touching the credit decision engine, which kept producing default rates above expectations. The screen changed; the capability did not evolve.

Domain Microservice. A domain is a boundary of language, decision and responsibility, a space where a set of concepts has precise meaning and where a team has authority to make decisions without asking another team for permission. A microservice is a deployment unit. You can have a domain implemented in a well-structured monolith or fragmented into thirty incoherent microservices. What matters first is the responsibility boundary; deployment granularity is a derived technical decision. Banks that jump straight to microservices without defining domains create distributed coupling, the worst of both worlds.

Event Technical Message. A business event is a fact that happened in the real world with auditable relevance: 'LoanProposalApproved', 'CreditLimitRevoked', 'SuspiciousTransactionIdentified'. It carries enough context for any consumer to understand what occurred without querying the source system. A technical message is a transport envelope. Treating events as technical messages produces systems where the consumer needs to make eight REST calls to reconstruct the context of a fact the producer already knew entirely. This antipattern is costly in latency, coupling and

traceability, three critical dimensions under BACEN scrutiny.

Why these three distinctions matter so much in a bank

- Business capabilities are the penthouse vocabulary, without them, the technical roadmap has no strategic anchor.
- Well-defined domains are the precondition for autonomous teams, autonomy without boundary is chaos with latency.
- Business events are the raw material for the audit trail and traceability required by BACEN and CMN Resolution 4.893.
- Confusing the three turns integration into an accidental product, the most expensive and hardest to evolve.
- Correctly naming each floor reduces the onboarding cost of new architects and engineers in high-turnover projects.
- The event technical message distinction is what enables event sourcing and CQRS as audit patterns, not just performance patterns.

The elevator's floors in a bank

The architecture conversation rides up and down these levels. Each ascent turns detail into risk and capability; each descent turns intent into an implementable decision.

Penthouse: Estratégia

- Conselho e estratégia crescimento - risco - eficiência (external)

Negócio e Produto

- Capacidades de negócio conta - crédito - pagamentos (frontend)
- Produto e jornada onboarding - Pix - crédito (frontend)

Domínio e Dados

- Domínios e eventos cliente - conta - contrato - limite (compute)
- Dados e linhagem produto - governança - prova (data)

Sala de máquinas: Plataforma

- Plataforma e runtime APIs - eventos - EKS - Lambda (compute)
- Operação e segurança SLO - IAM - auditoria - FinOps (security)

Flows

- estratégia -> capacidade: down: intent -> capability
- capacidade -> jornada: expressed as
- jornada -> domínio: realized by
- domínio -> dados: produces/consumes
- domínio -> plataforma: runs on
- plataforma -> operacao: operated by
- operacao -> estrategia: up: risk, cost, capability

Antipatterns: when the elevator gets stuck between floors

Recognizing a stuck elevator is as important as knowing how to operate it. Throughout my career in financial systems, I have learned to identify specific symptoms that indicate an organization has lost

the ability to move context between floors, and that any architectural decision made in that state will be expensive to undo.

Committee that only approves technology. When the only architecture governance forum discusses framework choices and library versions, but never questions whether the business capability being built solves a real pain point, the elevator is stuck in the technical basement. The typical consequence is a portfolio of technically correct services that nobody uses or that duplicate functionality without anyone noticing.

Roadmap without business pain. A roadmap that lists initiatives like 'Migration to Kubernetes', 'Credit Module Refactoring' or 'GraphQL Adoption' without linking each item to a business capability with an outcome metric is an engine room roadmap disguised as strategy. I have seen this pattern consume eight-figure budgets without moving a single business indicator.

Diagram with 40 boxes and zero owners. When an architecture diagram has dozens of components and none of them has an identified responsible team or person, what looks like a map is actually a photograph of technical debt. Without an owner, there is no decision; without a decision, there is no evolution.

Decisions only in one person's memory. In banks with high architect turnover, and this is more common than is admitted, knowledge about why a system was designed a certain way exists only in the head of whoever built it. When that person leaves, the team starts undoing correct decisions because they do not understand the context that motivated them. Architecture Decision Records (ADRs) are not bureaucracy; they are the mechanism that keeps the elevator operating even with operator changes. I will return to this topic in Chapter 14.

Skipping floors means building the right thing in the wrong place: The most expensive mistake I have seen in banking projects was not choosing the wrong technology, it was building the technically correct solution on the wrong floor. A critical business rule implemented directly in a data pipeline instead of in a business domain with a clear owner. A business event modeled as a database field instead of an immutable auditable fact. An entire business capability living inside a single microservice without a domain boundary. Each of these cases creates a debt that is not technical, it is architectural, and architectural is harder to pay because it requires organizational realignment, not just code refactoring.

How to use the floor map day to day

The elevator floor diagram is not an artifact to present once and file away. I use it as an active diagnostic tool in three recurring situations in banking projects.

At the start of a new engagement, I use the diagram to ask each stakeholder a simple question: 'On which floor do you spend most of your time, and on which floor do you feel most misunderstood?' The answers reveal where the communication gaps are before any technical analysis. A CTO who answers 'I spend time on the platform floor but struggle with the strategy floor' tells me there is a translation problem between what technology delivers and what the business expects, and that my priority work is to build that bridge.

In design reviews, I use the diagram to verify that each decision was made on the correct floor. A decision about event granularity belongs on the Domains and Events floor, not the Platform floor. A decision about data retention belongs on the Data floor with input from the Strategy floor (regulation), not the Operations floor. When a decision is being made on the wrong floor, it usually optimizes for the wrong criterion.

In incident discussions, the diagram helps separate technical root cause from architectural root

cause. A latency incident may have a technical cause (connection pool configuration) or an architectural cause (a domain that became a bottleneck because it absorbed responsibilities from three other domains). Treating the second as the first is what guarantees recurrence.

The ultimate goal of this chapter is simple: before writing a line of code or drawing a box in a diagram, you need to know which floor you are on. That location awareness, that ability to say 'I am making a domain decision, not a platform decision', is what distinguishes an architect who builds durable systems from one who builds systems that work in the demo and fail in production. In the next chapter, I will explore how to maintain that context while moving up and down, because the elevator in motion is where the real work happens.

Frequently asked questions about the bank's floors

Do I need all seven floors formalized before I start building?

No, but I need at least the floors adjacent to what I am building. If I am designing a domain, I need to understand the business capability above and the platform below. Formalizing everything at once is waterfall by another name.

How do I convince an engineering team to think in business capabilities if they only want to talk about services?

I start with their pain: I show how the lack of capability boundary is the reason the same bug appears in three different services. When the connection between technical confusion and the absence of business vocabulary becomes visible, resistance drops.

Does the floor model apply to small fintechs or only to large banks?

It applies to any organization that processes third-party money under regulation. In a small fintech, one person may inhabit several floors simultaneously, but the floors exist and the confusions between them cause the same damage, just faster because there is less margin for error.

What this chapter changes in your practice

After this chapter, you should be able to walk into any banking architecture meeting and identify which floor the conversation is happening on, and whether it should be happening on a different one. You should be able to name the difference between capability, domain and event without hesitation, and recognize the four stuck-elevator symptoms before they become production incidents or project failures. Most importantly, you should understand that skipping floors is not agility, it is building the right thing in the wrong place, and in financial systems that cost is always higher than it appears at the moment of the decision.

03 - Riding up and down without losing context

Riding the elevator is a trainable skill: going up turns technical detail into risk, cost and option; going down turns strategic intent into an implementable decision. This chapter gives the method.

Moving between strategy and the engine room is not innate talent, it is a learnable method with deliberate practice. The architect who masters this traversal does not merely design better systems: they become the only professional in the room capable of translating consequence in both directions, protecting the bank from invisible technical decisions that become regulatory risk, and from grand strategic visions that never find an implementation floor.

What it means to go up, and why most people stop at the mezzanine

Going up is not simplifying. Going up is reframing: taking a technical detail and expressing it in the currency that circulates on the upper floor, risk, cost, business capability, strategic option. Most architects stop at the mezzanine: they go up far enough to talk to product managers, but not far enough to sit with the Chief Risk Officer or the compliance director and be taken seriously.

Take a concrete example that runs through this entire book: end-to-end idempotency in Pix. On the technical floor, idempotency is a design attribute, each operation can be retried without additional side effects. This involves idempotency keys in the SPI, deduplication in the ledger, EndToEndId traceability, and state control in asynchronous events. It is real, it is complex, and most engineers can describe it precisely.

But the executive in the penthouse does not buy 'idempotency'. They buy the sentence only the architect can construct: 'this design decision eliminates an entire class of duplicate payments, meaning direct financial loss, formal complaints to BACEN, and reputational risk at social-media scale, and simultaneously creates the option to multiply transaction volume by five without rewriting the payments core.' Now you are on the right floor. You have transformed a technical attribute into avoided risk, avoided cost, and a real option: in the financial sense: the ability to act in the future without paying the cost today.

The heuristic I use is direct: if you cannot write one sentence that makes sense to the CRO and another that makes sense to the platform engineer, you do not yet fully understand the problem. It is not about having two speeches, it is about having a deep enough understanding that the same truth fits in different registers.

My view: the double sentence as a comprehension test: After sixteen years working on financial systems, from brokerages to digital banks, from mainframe migrations to live Pix platforms: I have learned that the architect who cannot write both sentences does not have a communication problem. They have a comprehension problem. Communication is consequence; comprehension is cause. When I enforce this discipline in architecture reviews, I invariably discover decisions that appeared technical but were actually unrecognized business choices, and vice versa. The double sentence is not rhetoric: it is a diagnostic instrument.

Heuristic: always write both sentences: For any architecture decision, write: (1) one sentence on the upper floor, what this means in terms of risk, cost, or option for the business; (2) one sentence on the lower floor, which design constraint, pattern, or mechanism implements that intention. If you cannot write both with precision, you do not yet understand the problem. Do not advance to the solution.

What it means to go down, and the rule of never going deeper than necessary

Going down is the inverse movement and equally critical: taking a strategic pressure and translating it into implementable design constraints. The key word is constraint, not solution. The architect who goes down with a ready-made solution has stolen from the engineer the space for creativity and accountability. The architect who goes down with clear constraints and explicit criteria has enabled the team to find the best solution within the correct space.

Consider the strategic pressure: 'reduce credit approval time from days to minutes'. In the penthouse, this is a competitive positioning decision, the bank wants to capture the customer's moment of intent, which has a half-life of minutes in digital channels. Descending this intention naively produces a vague requirement: 'the system must be fast'. That is not architecture, it is

wishful thinking.

Descending with method produces a set of chained constraints and decisions:

- Decision engine with versioned policy: credit logic must be auditable, testable, and modifiable without redeploying the core. This implies separating the engine (rules, models, variables) from the execution runtime, a design constraint, not a technology choice.
- Synchronous simulation with timeout and fallback: the customer journey requires a response in seconds. The design must anticipate what happens when bureau data arrives with latency above the SLO, the fallback is not an error, it is encoded business policy.
- Asynchronous formalization via events: approval can be communicated in real time, but contract formation, ledger update, and regulatory notification happen asynchronously, guaranteeing eventual consistency without blocking the journey.
- Journey SLO, not service SLO: the performance commitment is not 'the scoring microservice responds in 200ms', it is 'the customer receives a decision in under 90 seconds in 99% of cases'. This changes what you monitor, what you alert on, and what you report.

The rule I carry with me is: never go deeper than necessary. Every floor you descend without need increases the risk of over-specification, of turning a legitimate constraint into a premature solution that ties the team's hands and creates technical debt before the first commit.

The elevator itinerary: five steps to avoid losing context

1. Stop 1: Declare the capability and the outcome: Before any diagram or technical decision, articulate the business capability being built or protected, and the expected measurable outcome. 'Process Pix payments with guaranteed idempotency' is a capability. 'Eliminate duplicate-payment complaints at BACEN and enable volume growth without rewriting the core' is the outcome. If you cannot declare both precisely, return to the penthouse.
2. Stop 2: Model domains, events, and data: Descend to the domain floor: identify the relevant bounded contexts, the business events that carry state and intent, and the data that requires auditable lineage. At this stop, you have not yet chosen technology, you are mapping the problem space in the language of the business. In a bank, this means identifying which events are immutable facts (transactions, approvals, rejections) and which are derived projections.
3. Stop 3: Compare options with explicit criteria: Never present a single solution. Generate at least three architectural options and evaluate them against explicit criteria derived from the constraints identified in previous stops: estimated operational cost, operational complexity, regulatory adherence, reversibility, time to production. The criteria must be visible and traceable, not implicit in the final choice. This protects the decision from future revision and creates shared accountability.
4. Stop 4: Record the decision and create mechanisms: An unrecorded decision does not exist, it becomes folklore. Use Architecture Decision Records (ADRs) with context, options considered, criteria, decision, and anticipated consequences. More importantly: create mechanisms that make the decision hard to inadvertently violate: IaC guardrails, AWS SCP policies, contract tests, drift alerts. The decision must live in code and infrastructure, not only in the document.
5. Stop 5: Review in production with real data: Architecture only truly exists in production: Chapter 13 deepens this. At this stop, the architect returns to the elevator with observed data: real journey latency versus declared SLO, compensating event rate (an indicator of idempotency failures), real infrastructure cost versus estimate. This data rises to the penthouse as evidence to revise strategic premises, or descends to the engine room as corrected constraints. The cycle never closes, it iterates.

Maintaining context while the elevator moves

The greatest risk of the traversal is not going up or down, it is losing the thread of context midway. This happens in predictable ways: the strategy meeting ends and the architect goes directly to a technical refinement session without recording the constraints just heard; or the engineer brings a performance problem and the architect responds with a technical solution without checking whether the problem has relevance on the upper floor.

The mechanism I use to preserve context is deliberately simple: a context paragraph at the top of every ADR and every design document, written before any technical decision, that answers three questions, which business capability is at stake, what is the risk if that capability fails, and what time pressure conditions the decision. This paragraph is the umbilical cord between the penthouse and the engine room. When the team discusses technical options, it is always visible.

In the Brazilian banking context, maintaining this context has an additional regulatory dimension. BACEN does not ask which technology you used, it asks which risk you managed and how you can prove it. When the architect ascends with technical evidence translated into risk language, and descends with regulatory constraints translated into design decisions, they are building the bridge that makes the bank auditable by design, not by retroactive effort.

This also changes the nature of architecture reviews. Instead of sessions where engineers present diagrams and executives approve without understanding, you have conversations where every decision has a sentence on the upper floor and one on the lower floor, and anyone in the room can verify the coherence between the two. That is the environment where quality architecture happens: not in technical isolation, but in continuous dialogue between floors.

The elevator as institutional practice, not individual skill

Everything I have described so far may sound like a personal skill of the architect, and in part it is. But the ultimate goal is not to have an architect who rides the elevator: it is to have an organization that knows how to ride the elevator. This means product teams that understand the technical constraints conditioning their roadmap decisions. It means engineers who know the business risk behind the SLOs they are implementing. It means executives who can read an ADR and understand why a decision was made, even without understanding the implementation details.

This maturity does not happen by decree. It happens when the architect practices the method consistently and visibly, when every important decision has both sentences, when every architecture review starts with business context, when every production incident is analyzed on both the technical and the risk floor. Over time, the language of the elevator becomes the language of the organization.

In the following chapters, this method will materialize in specific domains: Chapter 06 will descend to the ledger and idempotency with the precision that Pix demands; Chapter 08 will show how events are the nervous tissue connecting floors in real time; Chapter 12 will demonstrate that security is evidence, and evidence only exists when the architect knows how to ascend with it. In all these cases, the elevator is the same. What changes is the destination floor and the translation currency.

The ability to go up and down without losing context is, ultimately, what distinguishes the architect who designs systems from those who merely describe them. It is what makes architecture consequential, not as an artifact, but as a living practice inside a bank that needs to be reliable, auditable, and capable of evolving.

Chapter key points

- Going up means reframing technical detail in the currency of the upper floor: avoided risk, avoided cost, created option. It is not simplifying, it is translating with precision.
- Going down means translating strategic pressure into implementable design constraints, never into ready-made solutions. Constraints enable; premature solutions bind.
- The two-sentence heuristic is a comprehension test: if you cannot write one sentence for the CRO and one for the platform engineer, you do not yet understand the problem.
- The elevator itinerary has five stops: declare capability and outcome; model domains and events; compare options with explicit criteria; record the decision and create mechanisms; review in production with real data.
- In the Brazilian banking context, BACEN requires that risks be managed and proven, the architect who translates technical decisions into risk language builds auditability by design.
- The ultimate goal is not an architect who rides the elevator: it is an entire organization that has learned the language of traversal between strategy and implementation.

PART II

Inside the Bank

Business before technology: intermediation, spread, the capability map, the payment rails, BACEN regulation, and the double-entry ledger as the accounting heart.

04 - What a bank does, capabilities, not screens

Before any system diagram: financial intermediation, spread, and the capability map that describes what the bank knows how to do, regardless of how it's implemented today.

Before drawing any system diagram, the architect needs to understand what a bank actually does, not which screens it displays, not which APIs it exposes, but which business functions it executes with real, irreversible financial consequences. This chapter opens Part II with a deliberately simple question: what is a bank, seen from the inside? The answer changes everything that follows.

The bank as a distributed system with strong guarantees

When I explain banks to experienced software engineers, I use an analogy that provokes productive discomfort: a bank is a distributed cache with exceptionally strong guarantees. A deposit is a write: the customer delivers money and the bank records an obligation. A withdrawal is a read with side-effect: the bank returns value and decrements the obligation. So far, familiar. What breaks the comfortable analogy is credit: when a bank lends R\$ 100,000, it is making a speculative read on money that does not physically exist at that instant in that account. The bank is betting that it will capture more than it will lose to defaults, that the spread will cover operating costs and default risk, and that regulation will keep the rules of the game stable enough for the model to work.

What makes this radically different from any conventional software system is the absence of tolerance for inconsistency. In modern distributed systems we accept eventual consistency as a reasonable trade-off: a lost message is reprocessed, a divergent state is reconciled in seconds. In a bank, every inconsistency is real money leaving or entering incorrectly. There is no 'we lost a message, that's fine.' A customer's balance cannot be eventually consistent, it must be exact, auditable and traceable to every cent, at any moment, including during a BACEN audit that can

happen without notice.

This does not mean banks don't use asynchronous architectures or distributed systems, they do, increasingly so. It means that consistency guarantees must be explicitly designed, not assumed as a framework default. The architect who rides from the engine room to the penthouse carries this awareness: what looks like a technical decision about idempotency or event ordering is, in fact, a decision about financial risk and regulatory compliance. We will return to this in depth in Chapter 06, when we address the ledger and idempotency as central mechanisms.

A minimal glossary of the business floor

- Financial intermediation: The bank takes money from those with a surplus and lends it to those who need it, taking on the risk and the term in between.
- Banking spread: The difference between the rate the bank charges borrowers and the rate it pays depositors. It is the margin of the core business.
- Ledger: The central accounting record, immutable and auditable, where every money movement is booked by double entry.
- Settlement: The moment money actually changes hands and the debit/credit becomes final and irreversible.
- BACEN (Central Bank of Brazil): The Central Bank of Brazil, regulator and supervisor; runs the core infrastructure (SPI, STR) and defines the system's design constraints.
- AML/CFT: Anti-Money-Laundering and Counter-Financing of Terrorism, mandatory, non-optional controls built into product and architecture.

Why the cache analogy matters for the architect: I use this analogy not to oversimplify banking, but to create an honest entry point for engineers who arrive with web-systems patterns in their heads. The real danger is not the engineer who doesn't know banking, that person asks questions. The danger is the engineer who thinks they know because they once integrated a payment gateway. Financial intermediation, spread, credit risk and regulatory obligation are concepts that change the nature of technical decisions. When the architect doesn't internalize them, they design a system that works in demo and breaks in production on the first accounting close.

Financial intermediation: the core business before any feature

A bank exists, in its economic essence, to do one thing: capture money from those who have it and lend it to those who need it, charging more than it pays. That difference, the spread, is the gross revenue of the intermediation business. Against it fall the cost of operations (personnel, technology, branches, compliance), the cost of risk (default provisions, regulatory capital required by BACEN under Basel III/IV) and, finally, profit.

Why does this matter to the architect? Because every technical capability we build directly serves one side of that equation. The funding system (checking accounts, savings, CDBs, LCIs) serves the liability side, the bank owes the depositor. The credit system (personal loans, financing, revolving credit card) serves the asset side, the bank is the creditor. The payments system (Pix, TED, boleto, cards) is the movement infrastructure connecting both sides and, increasingly, is also a revenue source through fees and transactional data.

When an architect doesn't understand this structure, they treat all systems as equivalent in criticality and fault tolerance. But they are not. A failure in the credit system during a proposal analysis window can be recovered in minutes without direct financial consequence. A failure in the Pix settlement system during SPI operating hours can generate regulatory fines, reputational damage and, in

extreme cases, BACEN intervention. Fault tolerance is not a technical decision, it is a function of business risk and applicable regulation.

The minimum business-floor glossary, presented next, formalizes these concepts with the precision needed for engineers and architects to speak the same language as risk directors and auditors. Without that shared vocabulary, the elevator doesn't work, the architect rides to the penthouse and can't communicate, or descends to the engine room and loses regulatory context.

What the spread finances, and why the architect needs to know

- Funding cost: interest paid to depositors and investors; defines the floor of the lending rate.
- Allowance for doubtful accounts (PDD): mandatory accounting reserve proportional to the credit portfolio risk.
- Regulatory capital (Basel): portion of equity the bank cannot use operationally, a direct opportunity cost.
- IT operational cost: every system we build is a line in the P&L; inefficient architecture erodes margin.
- Liquidity risk: the bank must honor withdrawals even when the credit portfolio is illiquid, the cash system is critical.
- Service revenue (fees, FX, custody): complements the spread and depends directly on the quality of payment and custody systems.

A bank's business capability map

What the bank knows how to do, grouped by capability family. No box here is a system, they are business functions that survive any technical rewrite.

Captação e Conta

- Abrir e manter conta (frontend)
- Captar depósitos (frontend)

Crédito e Cartões

- Conceder crédito política - motor - formalização (compute)
- Emitir e processar cartão (compute)

Pagamentos

- Pix - TED - boleto (messaging)
- Liquidar e conciliar (messaging)

Controle e Risco

- Conhecer o cliente (KYC) (security)
- Prevenir fraude e PLD (security)

Núcleo Contábil

- Ledger / escrituração (data)

Flows

- conta -> ledger: books
- credito -> ledger: books
- pix -> liq: triggers

- liq -> ledger: debits/credits
- kyc -> conta: enables
- fraude -> pix: monitors

Capability is not a screen: the map that precedes any solution

The most common mistake I see in banking modernization projects is starting with the system. The team receives a mandate: 'modernize payroll credit' or 'implement open finance', and immediately moves to technology choice, microservice definition and API design. The capability map is never drawn. The invariable result is a system that implements the current digitized flow, without questioning whether that flow makes sense, and that becomes impossible to evolve because no one knows where one capability ends and another begins.

A business capability is a function the bank knows how to execute, with a measurable outcome, regardless of how it is implemented today. 'Credit origination' is a capability. 'Credit analysis screen in legacy system X' is not, it is a specific, possibly poor, implementation of part of that capability. This distinction seems obvious written this way, but disappears completely under deadline pressure and when stakeholders describe the business in terms of screens and reports.

The capability map diagram presented next organizes a bank's functions into cohesive domains: account and relationship, credit, cards, payments, KYC and onboarding, fraud prevention and AML, and ledger/accounting. Each domain has its own language, distinct data model, specific regulatory controls and, crucially, different fault tolerance. Pix requires millisecond response latency and near-100% availability during SPI hours: BACEN monitors and penalizes deviations. Credit requires auditable decisions, reproducible simulations and legally valid formalization, latency of minutes is acceptable, loss of audit trail is not. KYC requires persistent, immutable evidence accessible for audit years later, throughput is low, but durability is absolute.

When the architect lacks this map, everything becomes generic integration. Each domain receives the same architectural pattern, the same SLA, the same data model. The result is a system that is simultaneously too expensive where it could be simple and too fragile where it needs to be robust. The capability map is the instrument that allows the architect to calibrate each technical decision against the correct business risk: and that is why it comes before any solution diagram.

Why the capability map comes before the solution: Without the map, the architect cannot answer the most important question they will receive at the penthouse: 'if this system fails, what does the bank lose?' With the map, the answer is precise: 'we lose credit origination capability for N hours, impacting X contracts per day and an estimated Y in revenue, plus SLA risk with banking correspondents.' Without the map, the answer is 'the credit system goes down', which means nothing to a risk director or to BACEN.

Each capability has its own language, and the architect must speak all of them

One of the most underestimated skills of the senior architect in banking is the ability to change vocabulary as they change domains, without losing technical precision. This is what the elevator demands in practice: ride up to the credit floor and speak in 'bureau score', 'credit policy', 'risk band' and 'CCB' (Cédula de Crédito Bancário); descend to the engine room and speak in 'versioned decision model', 'feature store', 'immutable contract with hash' and 'event sourcing for decision audit'. They are the same phenomenon, seen from different floors.

The payments domain speaks in settlement, clearing, D+0 window, SPI, ISPB and transaction

purpose. The KYC domain speaks in due diligence, PEP (Politically Exposed Person), OFAC list, CNPJ parent/branch and liveness proof. The fraud and AML domain speaks in typology, COAF reporting, R\$ 50,000 threshold, watchlist and behavioral analysis. The ledger domain speaks in double-entry, COSIF chart of accounts, accrual versus cash basis and position reconciliation.

Each of these domains has data that cannot be freely shared between them: not due to technical limitation, but by regulatory requirement and the principle of data minimization (LGPD applied to the financial context). The architect who designs a centralized data lake where all domains read and write freely is creating a governance problem that will surface at the first BACEN audit or the first AML investigation.

The table accompanying the capability map, presented next, details, for each domain, the primary regulatory controls, the type of sensitive data involved, latency tolerance and the consequence of failure. It is the calibration instrument that transforms the capability map from a strategic artifact into an operational guide for architecture decisions. The architect who masters this table can, in any technical or executive meeting, instantly connect a design decision to a business consequence, and that is precisely what distinguishes an elevator architect from an engine-room architect.

Frequently asked questions about business capabilities in banks

Can I use the same microservice for two different capabilities if the logic is similar?

Technically yes, but the cost tends to be high in the medium term. Capabilities with different regulations evolve at different rates and for different reasons, a credit policy change should not force a deployment in the payments system. Superficial logic similarity hides deep divergences in control, audit and fault tolerance. The separation criterion should be the regulatory domain and data model, not code similarity.

Does the capability map replace the domain model (DDD)?

No, they operate at different levels. The capability map is strategic: it says what the bank does and with what consequences. The domain model (DDD) is tactical: it says how each capability is structured internally in aggregates, entities and services. The map comes first and informs the DDD bounded contexts. Without the map, bounded contexts tend to reflect the current organizational structure, not the real business capabilities.

How does BACEN use this concept of capabilities in regulation?

BACEN does not use the term 'capability' explicitly, but its regulations (CMN Resolution 4.893 on PSTI, Circular 3.909 on Pix, BCB Resolution 85 on open finance) are written in terms of business functions and their controls, not in terms of systems or technologies. This means the capability map is the correct abstraction level for mapping regulatory obligations: each capability can be traced to the regulations that govern it, regardless of how it is implemented.

The capability map as a precondition for any banking project

No modernization project, cloud migration or new regulatory implementation in a bank should begin without a capability map validated with business, risk and compliance areas. Not because it is an architectural formality, but because without it the architect cannot calibrate criticality, cannot define system boundaries with foundation, and cannot translate technical consequences into risk language for the penthouse. The map is not the destination: it is the navigation instrument that makes the elevator functional.

05 - The rails and the rules

How money moves between institutions in Brazil: Pix, TED, card, boleto, what BACEN requires to operate, and what really changes when you're a fintech instead of a full bank.

Before designing any financial service on AWS, you need to understand the rails on which money will flow, and the rules that determine who is allowed to operate those rails. Ignoring this layer is the most expensive mistake an architect can make: you can build a technically flawless system that the BACEN shuts down the week of go-live. This chapter descends into the engine room of Brazilian payments and rises to the floor of regulatory strategy, because both perspectives are inseparable.

The distinction that matters most: authorization is not settlement

There is a conceptual confusion that runs through entire product and engineering teams, and it costs dearly when it reaches production: authorization and settlement are distinct events, separated in time, with completely different guarantees.

Authorization is a promise. When you swipe a card at a terminal, in milliseconds the issuer responds "approved", but not a single cent has moved yet. The cardholder received a credit reservation, the merchant received a conditional guarantee, and the entire system will live in that intermediate state for hours or days until settlement occurs. In the four-party card model, cardholder, issuer, acquirer, and brand, this window between authorization and settlement is a deliberate feature: it allows for compensation, reversal, chargeback, and batch reconciliation. The cost is complexity and financial latency.

Pix collapsed that window. When a Pix transaction is confirmed, authorization and settlement happen in the same event, in seconds, with irrevocable finality in the BACEN reserve account via SPI. This is technically harder, not easier. There is no correction window. A routing error, an idempotency problem, a reconciliation failure, all of this needs to be handled before confirmation, because afterward there is no way to undo it without a new transaction in the opposite direction, with all the regulatory implications that entails.

This distinction is not a product detail. It defines the resilience architecture, the error compensation model, the idempotency design (which Chapter 06 explores in depth), and even the risk profile you need to communicate to the board. The architect who does not internalize this difference will design Pix systems with a card mindset, and will discover the problem in production.

My read on the real difficulty of Pix: After working on high-criticality payment systems, the statement that irritates me most is 'Pix is simple because it's just an API'. Pix is one of the most demanding integrations a financial engineering team can face, precisely because the simplicity of the user experience hides a brutal requirement for zero eventual consistency: you cannot err and correct afterward. All the investment in idempotency, circuit breakers, proactive reconciliation, immutable audit trails, it exists to compensate for the absence of that correction window that cards offer for free. When I evaluate client architectures, the first sign of maturity I look for is whether the team understands this asymmetry.

The rails: each payment method has its own physics

Payments in Brazil are not a homogeneous market. They are four distinct ecosystems, each with its own infrastructure, operator, settlement model, and consequently distinct architectural requirements. The table that follows: The payment rails: what each one is for: organizes this view comparatively. Here, I want to build the intuition that makes the table useful.

Pix operates on two BACEN systems: DICT, which resolves the key (CPF, email, phone, random key) to the destination account, and SPI, which executes real-time gross settlement in reserve accounts. It operates 24 hours a day, 7 days a week, 365 days a year, without exception. This means your availability architecture cannot have conventional maintenance windows. Any PSP participating in SPI must guarantee contractual availability with BACEN; the cost of unavailability is not just reputational, it is regulatory.

TED operates on the STR (Reserve Transfer System), also from BACEN, but with defined hours, typically until 5 PM on business days. Settlement is gross and final, but the hourly model creates completely different physics: there is a queue, there is a cutoff, and transactions outside business hours need to be queued for the next business day. Architecturally, this requires pending transaction state management and retry logic with date semantics.

Boleto operates in a batch settlement model, with a D+1 or D+2 cycle. It is the most latency-tolerant instrument, but also the one requiring the greatest reconciliation robustness, because the volume of boletos paid at bank branches, lottery outlets, or internet banking generates a return flow that needs to be processed and reconciled against the internal ledger.

Card: credit and debit, operates in the four-party model already mentioned. Debit settles at D+1 via the card brand; credit can take 28 to 30 days for the merchant to receive, depending on the acquirer contract. Each of these windows is a design decision: who carries the credit risk during the interval, how float is managed, and what is the associated cost of capital.

The payment rails, what each is for

Criterion Speed Availability Primary use case

Pix Seconds (up to ~10s) 24/7/365 Instant P2P/P2B transfer and payment

TED (wire) Minutes, within hours Business days, STR window High-value interbank transfers

Boleto (bank slip) Hours to 1 business day Batch clearing Billing, bills, receivables

Card Authorizes in ms, settles in days 24/7 (authorization) Point-of-sale / e-commerce purchase

The end-to-end Pix flow: where each technical decision lives

To make concrete what I just described, the diagram that follows: End-to-end Pix outgoing flow: traces each step from the user's payment intent to the settlement confirmation in SPI. I want to draw attention to three critical points the diagram makes visible.

The first is key resolution via DICT. Before any payment instruction is sent to SPI, the paying PSP needs to query DICT to translate the Pix key into branch, account, and destination bank ISPB. This query is synchronous and is on the critical path of user experience. A failure here is not just a timeout, it is a decision: do you return an error to the user or try a cache? DICT caching has security implications (a key may have been transferred to another account). BACEN defines maximum TTL for DICT cache, and ignoring this rule is a real regulatory risk.

The second point is ISO 20022 messaging between PSPs via SPI. The communication protocol between SPI participants is based on ISO 20022 messages, with ICP-Brasil certificates for mutual authentication. This is not a conventional REST API, it is a messaging system with guaranteed delivery semantics, where each message has an end-to-end identifier (EndToEndId) that must be preserved immutably throughout the entire processing chain. This EndToEndId is the idempotency key of the Pix universe, and any architecture that does not treat it as a first-class citizen will have duplication problems.

The third point is settlement confirmation as a business event. When SPI confirms settlement, this

event needs to propagate to the internal ledger, to the notification system, to the limits engine, to the AML/CFT platform, all of this consistently. This is where event-driven architecture (Chapter 08) becomes not an aesthetic choice, but an operational necessity: the settlement event is the immutable fact from which all downstream systems derive their state.

End-to-end flow of an outbound Pix

A Pix crosses four trust boundaries in seconds. Every arrow is a point where idempotency, timeout and reconciliation must be solved, there is no 'undo'.

Banco Remetente / Sending bank (PSP)

- App / BFF (frontend)
- Motor Pix / Pix engine valida - reserva saldo (compute)
- Ledger remetente / sender (data)

BACEN: SPI / DICT

- DICT chave -> conta / key -> account (external)
- SPI liquidação / settlement (external)

Banco Destinatário / Receiving bank

- Motor Pix destino / receiver (compute)
- Ledger destinatário / receiver (data)

Flows

- pagador -> app: initiates
- app -> motorpix: idempotency key
- motorpix -> dict: resolve key
- motorpix -> spi: payment order
- spi -> motorD: settles
- motorpix -> ledgerR: debit
- motorD -> ledgerD: credit

AML/CFT is not a form, it runs through the entire architecture: Anti-Money Laundering and Counter-Terrorism Financing (AML/CFT) is frequently treated as a compliance module that the legal team handles. This understanding is dangerous and wrong. AML/CFT is an architectural constraint that runs through the entire stack: you need an immutable audit trail of every transaction (minimum 5-year retention per BCB Resolution No. 44), real-time monitoring of suspicious patterns (which requires event streaming, not batch), the ability to immediately freeze accounts under investigation (which requires your data model to support freeze states without corrupting history), and complete traceability of fund origin and destination. On AWS, this translates to concrete choices: CloudTrail with Object Lock for immutability, Kinesis or MSK for transaction event streaming, and a data model that separates operational state from regulatory state. Any architecture that does not design AML/CFT from the start will need to be rewritten, and rewriting a ledger in production is one of the riskiest operations that exists.

Regulation as design constraint: licenses, capital, and the SCR

When rising to the executive floor, the architect needs to translate regulation into business risk language. When descending to the engine room, they need to translate that same regulation into concrete design constraints. The elevator between these two floors is where most financial projects

fail, either by architects who never went up to the regulatory floor, or by executives who never came down to the technical floor.

BACEN operates a graduated license system. A multiple bank can take deposits, lend, issue cards, operate foreign exchange, but requires minimum regulatory capital that can be estimated in the tens to hundreds of millions of reais depending on active portfolios, plus governance structure, independent audit, and continuous reporting to BACEN. A Payment Institution (PI), regulated by BCB Resolution No. 80, can issue prepaid payment instruments, payment accounts, or credential merchants, with proportionally smaller capital and governance requirements. An SCD (Direct Credit Company) can grant credit with its own resources, without taking public deposits.

Each license type defines the perimeter of what you can do and, consequently, what needs to be in your system. A PI that cannot take deposits needs a client resource segregation model (a payment account is not a bank checking account, resources must be held in safe assets). An SCD that grants credit needs to report to the SCR (BACEN Credit Information System): and the SCR is bidirectional: you report the credit operations you grant, and can query a borrower's credit history. This has privacy implications (LGPD), latency implications (SCR query is on the credit approval path), and data integrity implications (divergence between your ledger and SCR is a serious regulatory problem).

The point I want to fix: regulation is not a list of documents to deliver to BACEN. It is a set of invariants that your system needs to maintain in production, continuously, under any load or failure condition. Designing for this from the start is incomparably cheaper than remediating afterward.

Bank versus fintech on BaaS: the autonomy ceiling

One of the most consequential strategic decisions for a company wanting to operate financial services is: build on its own license or operate on Banking as a Service (BaaS) from a partner? The table that follows: Bank × fintech: what changes in practice: maps the dimensions of this choice. Here I want to deepen the central trade-off the table captures.

Operating on BaaS is the path of least initial friction. You outsource the license, regulatory capital, SPI/STR connection infrastructure, and part of the regulatory responsibility to the partner bank. In exchange, you gain go-to-market speed that can be measured in months versus years. For a fintech in product validation phase, this trade-off often makes sense.

But there is an autonomy ceiling that needs to be understood before committing to this architecture: the real ledger belongs to the partner. When your fintech processes a transaction via BaaS, the definitive record of that transaction lives in the partner bank's system. You have a derived view, a mirror, a reconciliation, but not the primary ledger. This has direct consequences: your ability to innovate in credit products is limited by what the partner exposes via API; your ability to respond to regulatory audits depends on partner cooperation; and if the partner changes its commercial policy or discontinues the BaaS product, you have a business continuity problem not under your control.

The architectural maturity of a fintech can be measured, in part, by how consciously it manages this ceiling. The most sophisticated ones build a shadow ledger: an internal representation of all financial positions, continuously reconciled against the BaaS partner, which serves both for operational autonomy and for the day they decide to migrate to their own license. This design decision, made early, is what separates fintechs that can scale from fintechs that remain trapped in partner dependency.

The architect who understands this spectrum, from fintech on BaaS to multiple bank with its own infrastructure, can have a much more honest conversation with the board about what the company

is buying with each choice. Speed now versus autonomy later is not an obvious choice; it is a calculated bet that needs to be recorded as an explicit architectural decision.

Bank vs. fintech: what changes in practice

Criterion	Full bank	Fintech (payment institution)	Fintech on BaaS
-----------	-----------	-------------------------------	-----------------

BACEN license	Full (takes deposits and lends)	Payment Institution	None of its own, uses the partner's
---------------	---------------------------------	---------------------	-------------------------------------

Access to the payment system	Direct participant	Direct or indirect	Indirect, via a settling bank
------------------------------	--------------------	--------------------	-------------------------------

Can keep its own ledger?	Yes, it is the core	Yes, for the payment account	Limited, the real ledger is the partner's
--------------------------	---------------------	------------------------------	---

Speed to launch	Slow, heavy control	Medium	Fast, but with an autonomy ceiling
-----------------	---------------------	--------	------------------------------------

Where the architecture stalls	Regulatory weight and legacy limits	Capital and compliance	BaaS dependency and limits
-------------------------------	-------------------------------------	------------------------	----------------------------

What this chapter established

- Authorization is a promise in milliseconds; settlement is fulfillment, and Pix collapses both, eliminating the correction window that other instruments offer.
- Each payment rail (Pix/SPI, TED/STR, boleto, card) has its own physics: availability, latency, settlement model, and distinct architectural requirements.
- The Pix EndToEndId is the system's idempotency key, treating it as a first-class citizen throughout the entire processing chain is non-negotiable.
- AML/CFT is an architectural constraint that runs through the entire stack, immutable trail, real-time monitoring, years of retention, not an isolated compliance module.
- The license type (multiple bank, PI, SCD) defines the perimeter of what the system needs to maintain as an invariant in production, not just what can be sold as a product.
- Fintech on BaaS gains speed, but the primary ledger belongs to the partner, the autonomy ceiling needs to be managed consciously, ideally with a shadow ledger from the start.

What the architect carries from this chapter

The rails and the rules are not background context, they are the ground on which every technical decision rests. An architect who understands the difference between authorization and settlement, who knows where DICT ends and SPI begins, who reads a Payment Institution license as a non-functional requirements specification, and who can explain the BaaS autonomy ceiling to a CEO in three minutes, that architect is operating in the elevator, between the penthouse and the engine room, which is exactly where value is created. The next chapters will deepen each of these layers: the ledger and idempotency in Chapter 06, the complete reference architecture in Chapter 07, and events as the nervous system in Chapter 08. But all of that only makes sense on the foundation we just built here.

06 - The ledger is the heart, and idempotency is the blood

The double-entry ledger is the Git of money: immutable, auditable, append-only. And idempotency isn't an implementation detail, it's the central functional requirement of any system that moves money.

Every financial system, however sophisticated its product layer may appear, rests on a

five-century-old primitive: the double-entry ledger. When that foundation is poorly implemented, balances updated in place, idempotency treated as an engineering detail, reconciliation relegated to month-end, the bank does not have a technical problem; it has a financial integrity problem. This chapter descends to the conceptual core to show that ledger and idempotency are not design choices: they are first-class functional requirements, and the architect who fails to defend them in the penthouse loses money in the engine room.

The ledger is the Git of money

Think about what Git does: it never overwrites a commit. Every change is a new immutable object; the current state of the repository is a projection over history. The accounting ledger works exactly the same way, and not by accident, this property was formalised by Luca Pacioli in the fifteenth century precisely because money demands absolute traceability.

The rule is simple and non-negotiable: never UPDATE a balance. Always INSERT journal entries. The balance of an account is the algebraic sum of all entries associated with it since its opening. This means the question "what was the balance of this account at 14:32 yesterday?" has a deterministic, auditable answer, simply filter entries with timestamp $\leq$ '2024-01-15 14:32:00' and sum. With a mutable balance field, that question cannot be answered with certainty unless you maintained a separate log, which is, ironically, a ledger.

Double-entry adds the second layer of guarantee: every debit has an equal and opposite credit. When the bank transfers R\$ 1,000 from Alice's account to Bob's, two entries are created atomically: a debit on Alice's account and a credit on Bob's, both with the same value, the same transaction_id, and the same timestamp. The sum of all entries in the system, at any moment, must be zero. If it is not zero, money was created or destroyed, and that is a bug with regulatory consequences, not an eventual inconsistency to be resolved later.

On AWS, this model translates to an append-only journal_entries table in DynamoDB or Aurora PostgreSQL, with a view or materialised query that projects balances. The temptation to maintain a current_balance field updated on every transaction is understandable, it seems more performant, but it creates a duplicated source of truth that, under partial failure, diverges silently.

The golden rule of the ledger: Never UPDATE a balance. The balance is a projection, the sum of entries up to instant T. This property is what allows you to answer 'what was the balance at 14:32 yesterday?' with surgical precision and auditable evidence. A mutable balance field is a convenient lie that the regulator, the auditor, and the customer will collect with interest when the inconsistency surfaces. If you need read performance, use a materialised incremental projection, but keep the ledger as the single source of truth.

Idempotency: the functional requirement that the network hides

There is a scenario that every financial systems architect needs to have ingrained in muscle memory: the network drops after the client sends the payment request, but before the server confirms receipt. The client does not know whether the payment was processed. The correct behaviour is to retry, but if the system is not idempotent, the retry generates a second payment. The money already left the account on the first attempt; now it leaves again.

This is not an edge case. It is the normal behaviour of any distributed system under real load. Timeouts happen. Load balancers restart. Message queues deliver events more than once: SQS, for example, guarantees at-least-once delivery, not exactly-once. Treating duplication as an exception is the most expensive design error I have seen in banking systems.

Idempotency is a first-class functional requirement, not an implementation detail. This means every

financial operation must carry an idempotency key (`idempotency_key`) generated by the client, a UUID v4, for example, and the server must guarantee that the same key processed twice produces exactly the same result with no additional side effect. In practice: the second call with the same key returns the first call's response, without creating a new entry in the ledger.

On the consumer side, the same logic applies: every consumer processing messages from an SQS queue, an SNS topic, or a Kinesis stream must be idempotent from day zero. The design question is not 'how do I avoid duplicates?', but rather 'what happens when I process this message twice?' If the answer is 'it creates two entries', the system is wrong. If the answer is 'it detects it already processed this and returns with no effect', the system is correct.

The table below compares the expected consistency behaviour when the asset is money versus other types of data, and why the guarantees we accept in content systems are unacceptable in financial systems.

Replay and duplication are normal behaviour, not exceptions: SQS delivers at-least-once. Kinesis allows explicit replay. EventBridge can reprocess events on destination failure. If your consumer is not idempotent, each of these mechanisms, which exist to increase resilience, becomes a vector for payment duplication. Do not design for the happy path and then try to add idempotency as a patch. Design idempotency first, as a business constraint.

Authorisation, settlement, and the reconciliation point

An architectural error I frequently see in banking systems built in haste is the direct coupling of authorisation and settlement. The flow seems reasonable on the surface: the client requests a transfer, the system authorises, debits and credits in the same database transaction, and returns success. Simple, atomic, correct, until the day you need to integrate with an external clearing house, or when settlement must occur at T+1, or when the regulator requires a separate audit trail for each phase.

Authorisation and settlement are distinct events in time and regulatory meaning. Authorisation is the promise: the bank verified that funds exist, blocked the amount, and committed to settle. Settlement is the effective transfer of ownership. Between the two, there is a state: 'authorised, pending settlement', that must be represented explicitly in the ledger as provisioned entries, not as a flag in a transactions table.

The reconciliation point is the mechanism that verifies, periodically or in real time, that the sum of authorised and settled entries is consistent with the positions reported by external counterparties: SPB, clearing houses, custodians. This point is not a monthly report. It is a continuous, automated process that must raise alerts within minutes when a divergence is detected, not within days.

On the architecture elevator, the conversation about authorisation versus settlement starts in the penthouse: the CFO and Chief Risk Officer need to know that the bank has exposure during the window between authorisation and settlement, and that this exposure is measurable and monitored. The architect who does not ride up to have that conversation will implement a system that works in development and creates systemic risk in production. The engine room, the SQS queues, the Kinesis streams, the reconciliation lambdas, only makes sense when the floor above has understood why each piece exists.

Anti-patterns that cost real money

After sixteen years working with financial systems, I can list the anti-patterns that appear most often, and that cost the most, whether in incidents, regulatory fines, or architectural rework:

1. Direct UPDATE on balance. We have already discussed why. The practical problem is that, beyond losing auditability, UPDATE under high concurrency requires locks that degrade throughput. The append-only ledger scales horizontally with far more elegance.
2. Consuming events without idempotency. The consumer processes the message, calls the payment service, the service returns a timeout, the consumer does not commit the offset, the message is redelivered, the payment is executed twice. This bug exists in production in more systems than anyone would like to admit.
3. Coupling authorisation and settlement without a reconciliation point. The system works perfectly under normal conditions. At the first integration with an external clearing house that reports a divergence, there is no mechanism to identify where the value was lost. The investigation takes days; the regulator finds out.
4. Treating reconciliation as a monthly report. Reconciliation is a failure detection process. Running it monthly is equivalent to checking security logs once a month, by the time you discover the problem, the damage is already done. In modern financial systems, reconciliation must be continuous, automated, and integrated into the operational alerting pipeline.
5. Using the same database for ledger and operational data without model separation. The ledger has a fundamentally different access model, append-only, analytical reads over history, while operational data has frequent transactional reads and writes. Mixing the two in the same schema creates contention and makes independent evolution of each domain difficult.

Each of these anti-patterns has a version that seems reasonable during development and reveals its real cost only under partial failure, high load, or regulatory audit. The architect's job is to make that cost visible before it materialises.

Non-negotiable principles of ledger and idempotency

- The ledger is append-only: INSERT journal entries, never UPDATE balances. Balance is always a projection calculated over immutable history.
- Double-entry guarantees conservation: the sum of all entries must be zero. If it is not, money was created or destroyed, that is a bug, not an acceptable inconsistency.
- Idempotency is a business functional requirement, not an engineering detail. Every financial operation must have an idempotency key and the system must guarantee that reprocessing produces no additional effect.
- Event consumers must be idempotent from day zero. At-least-once delivery is the queue standard; duplication is normal behaviour, not an exception.
- Authorisation and settlement are distinct events with explicit representation in the ledger. Coupling them without a reconciliation point creates invisible systemic risk.
- Reconciliation is continuous failure detection, not a periodic report. Divergences must generate alerts within minutes, not days.

Consistency: what changes when the asset is money

Criterion Typical e-commerce / SaaS Banking system Why the difference matters

--- --- --- ---

Consistency model Eventual is usually enough Strong in the core, eventual at the edges A wrong balance for 2s is already wrong money

Balance operation UPDATE on the record INSERT into the ledger (append-only) Audit and historical reconstruction

Duplicating a message Usually tolerable Unacceptable, double payment Idempotency is a requirement, not a

bonus

Losing a message Retry and move on Unacceptable, reconciliation is mandatory Outbox, DLQ and safe reprocessing

Source of truth The app's database The ledger + reconciliation with the regulator It must match BACEN, not just internal state

The architect between the penthouse and the engine room

There is a real tension between what the product team wants, speed, features, time-to-market, and what the ledger demands, immutability, idempotency, continuous reconciliation. This tension is not resolved in the engine room. It needs to be resolved in the penthouse, in the language of business risk.

When I ride up to the penthouse to talk with the CTO or Chief Risk Officer of a bank, I do not start with the technical architecture. I start with the question: 'If a payment is processed twice due to a network failure, how long does it take to detect? Who is notified? What is the reversal process?' If the answer is vague: 'we catch it in the monthly close', then the risk already exists, regardless of how the system was implemented.

The technical conversation that follows, about idempotency keys, about idempotent consumers, about the separation between authorisation and settlement, only carries weight when the floor above has understood the cost of not having them. The architect who goes straight to the engine room without making that translation will implement correctly and be ignored in the next prioritisation decision.

The double-entry ledger and idempotency are not academic abstractions. They are the mechanisms by which a bank proves, at any instant, that it has neither created nor destroyed money, and that it can demonstrate this to BACEN, to the external auditor, and to the customer calling to ask where their transfer went. This capacity for proof is what separates a financial system from a system that processes payments. And building it correctly is, fundamentally, the responsibility of the architect who knows how to ride the elevator.

My direct opinion: ledger and idempotency are non-negotiable: In sixteen years, I have never seen a financial system that started with mutable balances and weak idempotency and then corrected that painlessly. Migrating from a mutable balance model to an append-only ledger in production is one of the riskiest and most expensive operations a bank can undertake, and it always happens after an incident that exposed the problem to the regulator. My recommendation is direct: if you are designing a new financial system, start with the correct ledger. If you are evolving a legacy system, treat the migration to an append-only ledger as a regulatory risk project, not a technical refactoring. The cost of doing it right at the start is a fraction of the cost of correcting it later.

Frequently asked questions about ledger and idempotency

If balance is calculated by sum, does it not become slow for accounts with many years of history?

This is a legitimate performance concern, not a correctness concern. The standard solution is a periodic checkpoint: a consolidated balance up to a cutoff date, plus incremental entries after that cutoff. The checkpoint is derived data, never the source of truth, and can be invalidated and recalculated at any time. DynamoDB with streams and Lambda, or Aurora with materialised views, implement this pattern efficiently.

How do I implement idempotency keys in a REST API on AWS?

The most robust pattern is: the client generates a UUID v4 and sends it in the Idempotency-Key header. The server, before processing, checks an idempotency table (DynamoDB is ideal for its low latency and native TTL) whether that key has already been processed. If yes, it returns the stored response. If no, it processes, stores the response with the key, and returns. API Gateway with Lambda can implement this pattern natively. The key TTL should be sized to cover the client's maximum retry window, typically 24 hours for financial operations.

Does double-entry require debit and credit to be inserted in the same database transaction?

In a monolithic system with a single relational database, yes, and that is the simplest approach. In a distributed system with multiple services, atomicity is guaranteed by saga with compensation: if the credit fails after the debit has been inserted, a compensation entry (debit reversal) is inserted. The critical point is that the ledger never enters an inconsistent state, it may enter a 'pending compensation' state, which is explicit and monitored, but never a state where the debit exists without the corresponding credit without a record.

The correct ledger is the foundation of everything that follows

The following chapters will build on this foundation: the banking reference architecture on AWS (Chapter 07), the event-driven nervous system (Chapter 08), and data as a product with auditable lineage (Chapter 09) only make complete sense when the underlying ledger is immutable, idempotency is guaranteed, and reconciliation is continuous. A financial system without these foundations is a system that works until it fails, and when it fails, it fails in ways that the regulator and the customer do not forgive. The architect who understands this is not being a perfectionist: they are being precise about where the risk actually lives.

PART III

Architecture Descends to the Engine Room

The reference architecture of a modern bank on AWS: core banking, events, data, the runtime platform and generative AI with guardrails, each piece tied to a business pain.

07 - Reference banking architecture on AWS

A reference view for modern financial systems on AWS: channels and BFFs, domains in containers and serverless, governed events, data as a product, and AI with guardrails, designed for FinOps and resilience.

A reference architecture is not an AWS service catalog with arrows between boxes, it is a declaration of intent about how zones of responsibility connect, how risk flows between them, and where cost is justified by the value delivered. In this chapter, I walk through the reference view I use as a starting point in engagements with banks and fintechs, explaining not just what is in the diagram, but why each piece is there and what happens when it is not.

Why I use a reference, not a recipe: After sixteen years working in financial systems, I have learned that the worst mistake an architect can make is delivering a diagram as if it were a universal truth. The reference I present here is deliberately opinionated: it reflects decisions about where serverless makes sense, where containers are necessary, and how events create

reversible boundaries between domains. You will disagree with some choices, and that is exactly the point. A good reference provokes the right conversation, it does not eliminate the need to think.

The diagram zones: edge, domains, events, data, and operations

The diagram accompanying this chapter divides the platform into five functional zones. These are not stacked layers, they are regions with distinct responsibilities that communicate through explicit contracts.

Edge and identity zone. All external traffic enters through CloudFront, which delivers edge protection, caching, and TLS termination before any application logic. WAF sits immediately behind it, applying managed and custom rules that the security team can evolve without touching business code. Cognito and IAM Identity Center solve two different problems: the former handles end-customer identities (federated authentication, MFA, OIDC tokens); the latter manages internal identities and access to administrative consoles and APIs. API Gateway is the boundary between the external world and the BFFs: Backend for Frontend, which translate each channel's language (mobile, internet banking, open finance) into the internal contracts of the domains. This separation matters: when the regulator requires a new field in the open finance statement, the change is contained in the open finance BFF, not propagated to the account domain.

Domains zone. This is where the business capabilities described in Chapter 04 live. Domains with complex state, long life cycles, or runtime control requirements, such as the credit engine or the payments processor, run on EKS or ECS. Domains oriented to discrete events, without persistent state between invocations, use Lambda and Step Functions. This is not a religious choice: it is an operational model decision I discuss in detail in Chapter 10. What matters here is that both models coexist on the same platform and share the same event contracts.

Events, data, and AI: the zones that sustain platform intelligence

Events zone. MSK (managed Kafka) is the backbone for high-throughput, retention-heavy streams, transactions, positions, risk alerts. EventBridge complements with schema-based routing for lower-volume but high-semantic business events, such as credit approval or completed onboarding. SQS appears at points where queue semantics, guaranteed delivery, dead-letter, controlled visibility, matter more than the pub/sub model. These three services are not interchangeable: each solves a different class of problem, and mixing them without criteria generates operational complexity without benefit. In Chapter 08, I go deeper on the reasoning for when to use each.

Data zone. Aurora PostgreSQL serves transactional domains that need ACID and expressive SQL, the ledger, portfolio position, customer registry. DynamoDB serves domains that need sub-millisecond latency and predictable horizontal scale, sessions, product cache, preferences. S3 with Glue and Lake Formation forms the governed data lake, where product data (Chapter 09) is catalogued, versioned, and made available with column- and row-level access control. No data leaves this zone without passing through Lake Formation, that is the governance boundary the auditor will ask about.

AI zone. Bedrock with Guardrails is not an ornament, it is the layer that allows language models to be used in regulated flows without sacrificing auditability. Chapter 11 covers this in depth. What matters in the reference view is that AI is inside the platform, not glued on from the outside: it consumes events, writes to product data, and is governed by the same identity and access controls as the rest of the architecture.

Operations zone. CloudWatch and OpenTelemetry form the observability plane. Security Hub

aggregates findings. GuardDuty detects anomalous behavior. Config records every resource configuration change. KMS manages keys with auditable rotation. This zone is not optional in a regulated environment, it is the evidence that Chapter 12 will require.

A modern banking platform on AWS, reference view

Not a service catalog: a model of how the zones connect. Channels at the edge, isolated domains, events as the contract between them, governed data, and AI treated as a capability with boundaries.

Borda e Identidade / Edge & Identity

- CloudFront + WAF + Shield (edge)
- Cognito / IAM Identity Center contextual authz (security)
- API Gateway / BFFs (edge)

Domínios / Domains (runtime)

- EKS / ECS critical domains (compute)
- Lambda + Step Functions event-driven flows (compute)

Eventos / Events

- Amazon MSK high volume - ordering (messaging)
- EventBridge domain events (messaging)
- SQS / DLQ decouple - peaks (messaging)

Dados / Data

- Aurora relational domains (data)
- DynamoDB low latency (data)
- S3 + Glue + Lake Formation governed data (storage)

IA / AI

- Bedrock + Guardrails Knowledge Bases - AgentCore (ai)

Operação / Operations & Security

- CloudWatch + OpenTelemetry (ci)
- Security Hub - GuardDuty - Config - KMS (security)

Flows

- cliente -> waf: HTTPS
- waf -> apigw
- apigw -> cognito: authz
- apigw -> eks: synchronous
- eks -> eb: publishes events
- eb -> lambda: triggers
- eks -> msk: high volume
- lambda -> sqs
- eks -> aurora
- lambda -> dynamo
- msk -> lake: ingestion
- lambda -> bedrock: RAG / agents

- bedrock -> lake: Knowledge Base

What the diagram does not show, and you need to know

- AWS account boundaries (multi-account) do not appear in the diagram for visual clarity, but they are a day-zero decision: each business domain in a separate account is the default defensive posture in regulated environments.
- The diagram does not show latencies or SLOs, those numbers depend on the product, not the platform, and must be defined before any service choice.
- No arrow in the diagram represents a synchronous call between distinct domains. If you need to draw one, it is a signal that the domain boundary is wrong.
- The diagram is AWS region-agnostic, but region choice has data sovereignty implications that BACEN and LGPD make non-trivial.
- The operations zone is not a phase-two add-on, it must be provisioned before the first business service, not after.

FinOps as a design principle, not a post-hoc optimization

The first principle underpinning this design is FinOps, and I need to be precise about what that means here. I am not talking about cost dashboards or Reserved Instances. I am talking about an architectural decision: cost per transaction is a product metric, not an infrastructure metric.

When a payments domain processes a transfer, the cost of that operation, compute, storage, data transfer, API calls, must be measurable and attributable. This has two practical consequences. First: every piece of the architecture must justify its operational cost relative to the value it delivers. An EKS cluster running a service that processes two hundred requests per day is a design problem, not an optimization problem. Second: the choice between serverless and container is not aesthetic, it is financial and operational.

The position I adopt in this reference is: serverless and pay-per-use where the execution model is compatible; container where the domain requires runtime control, persistent state, or performance characteristics that the serverless model cannot guarantee economically. Lambda with event-driven architecture has near-zero marginal cost during low-demand periods, something an EKS cluster will never achieve, even with Karpenster and aggressive scaling. On the other hand, a derivatives pricing engine that needs dedicated CPU, predictable memory, and zero cold-start latency does not belong in Lambda.

The practical consequence for the architect riding up to the executive floor: when the CFO asks why the AWS bill grew thirty percent last quarter (and they will ask), you need to be able to answer in terms of transaction volume, not instance hours. That is the conversation this reference makes possible, because cost is distributed by domain, and each domain maps to a business capability with associated revenue.

Reversibility: the architecture that keeps options open

The second principle is reversibility, and here the architecture elevator appears most explicitly. On the executive floor, the business needs agility: launching new products, responding to regulatory changes, integrating partners. In the engine room, the engineering team needs autonomy: evolving a domain without coordinating with all others, swapping an implementation without rewriting integrations. These two imperatives are the same imperative seen from different floors, and events are the mechanism that reconciles them.

When two domains communicate via event, the contract between them is the event schema, not the

internal implementation of either. The account domain publishes an AccountUpdated event with a versioned schema. The notification domain consumes that event. If tomorrow the account domain migrates from Aurora to DynamoDB, or from ECS to Lambda, the notification domain does not know and does not need to know. This is the reversibility the diagram materializes: events as contracts between domains allow swapping one domain's implementation without rewriting the others.

The practical implication is that a good architecture does not try to predict the future, it keeps the maximum number of options open at the lowest maintenance cost. This has an important corollary: synchronous coupling between domains is technical debt with compound interest. Every direct REST call between distinct domains is a dependency that will be costly when either needs to evolve. I am not saying synchronous calls are always wrong: I am saying they need to be a conscious decision, with the reversibility cost explicitly accepted.

In the Brazilian banking context, this has an additional regulatory dimension. BACEN evolves norms frequently: PIX, open finance, DREX. Each new regulation is a change vector that will hit specific domains. An architecture with reversible boundaries absorbs these changes in a localized way. A tightly coupled architecture turns every new norm into a six-month project with production regression risk. As the diagram below shows, the reference zones are designed precisely so that this isolation is possible.

How to read the reference diagram in a real engagement

1. Identify the business capabilities present: Before looking at any AWS service, map which capabilities from Chapter 04 the bank already has and which it is building. Each capability goes into a domain zone, not into a specific service.
2. Mark the regulatory risk flows: Identify which data flows cross regulatory boundaries, customer data, transaction data, position data. These flows determine where Lake Formation, KMS, and Config are mandatory, not optional.
3. Validate event contracts between domains: For each arrow in the diagram that crosses a domain boundary, ask: what is the event schema? Who is the canonical producer? What is the versioning policy? If there is no answer, the boundary is not ready yet.
4. Apply the Well-Architected pillars as a checklist: The Well-Architected block accompanying this chapter reads each reference zone through the six pillars. Use it as a review script before any go-live decision.

The reference does not replace your bank's context: Every time I present this reference, someone asks whether they can simply adopt it as-is. The answer is no. The reference is a starting point for the right conversation, about domain boundaries, about operational model, about risk tolerance. A wholesale bank with ten thousand transactions per day has different cost and complexity constraints than a retail fintech with ten million. The reference works for both, but the choices within it will be different. The architect's job is precisely to make those choices with evidence, not by analogy.

Frequently asked questions about the reference architecture

Why MSK and not Kinesis for the event backbone?

Kinesis is a valid choice for flows with lower operational complexity. MSK (Kafka) is preferable when the bank already has Kafka competency, when it needs long retention with granular replay, or when it wants workload portability between cloud and on-premises. The decision should be based on the team's operational model, not the architect's familiarity with either.

EKS or ECS for containerized domains?

ECS has lower operational overhead and is sufficient for most cases. EKS makes sense when the bank needs cross-cloud portability, when it already has investment in Kubernetes ecosystem tooling, or when platform teams have the maturity to operate the control plane. Do not choose EKS because it is more sophisticated, choose it because your team's operational model justifies the additional complexity.

How does the reference relate to BACEN resilience requirements?

BCB Resolution No. 85 and BACEN business continuity norms require documented RTO and RPO for critical systems. The reference supports this through native multi-AZ in managed services, event replication in MSK, and backup strategies in Aurora and S3. But the reference does not define the numbers, that is the responsibility of the bank's risk management process, not the technical architecture.

Reading the reference through the Well-Architected pillars

- security: Contextual identity, least privilege, KMS encryption, an immutable CloudTrail, and account/OU segregation. Security is evidence, not opinion.
- reliability: Isolated domains, queues with DLQs, end-to-end idempotency, multi-AZ by default, and a recovery plan tested per journey, not generically.
- cost: Serverless where traffic is irregular, containers where scale is constant, and cost per transaction as a product metric watched continuously.
- operational-excellence: Golden paths, observability from day one, actionable runbooks, and mechanisms that make the good path the easy path.
- performance: Aurora for relational consistency, DynamoDB for low latency, cache and CQRS where reads and writes have different profiles.
- sustainability: Pay-per-use cuts idle capacity; continuous right-sizing and event-driven architecture avoid polling and wasted compute.

Reading the reference through the pillars: what comes next

A reference view walked through zone by zone is necessary, but not sufficient. The architect who rides up to the risk committee floor needs to be able to answer questions that are not in the diagram: what happens when MSK becomes unavailable? Who has access to customer data in the data lake and how is that audited? What is the cost of a PIX transaction in this architecture at a scale of ten million operations per day?

These questions are answered when you read the reference through the six pillars of the AWS Well-Architected Framework, operational excellence, security, reliability, performance efficiency, cost optimization, and sustainability. The block that follows this text does exactly that: it walks through each reference zone and points to where each pillar manifests, where there are tensions between pillars, and what questions the architect must be able to answer before considering the architecture ready.

What I want to make clear before you read that block is this: the pillars are not a compliance checklist. They are a shared vocabulary for having difficult conversations, about trade-offs between cost and reliability, between delivery speed and security, between domain autonomy and centralized governance. Using Well-Architected as a review tool is one of the most efficient ways I know to move from the technical diagram to the business risk conversation without losing precision along the way. The following chapters in this Part III will go even deeper into each of these dimensions, events

in Chapter 08, data in Chapter 09, platform in Chapter 10. The reference presented here is the map; the next chapters are the territory.

What this reference delivers, and what it does not

This reference architecture delivers a model of how zones of responsibility connect in a modern banking platform on AWS, two explicit design principles (FinOps and reversibility), and a shared vocabulary for conversations between the executive floor and the engine room. It does not deliver a production-ready design, does not define SLOs, does not replace the bank's risk management process, and is not valid without adaptation to each institution's specific context. Use it as a starting point for the right questions, not as an answer to all of them.

08 - Event-driven as the bank's nervous system

Banks have always been event-driven, long before the term was fashionable. The question is whether business facts stay hidden in coupled systems or become explicit contracts, with schema, versioning and ownership.

Banks have always been event-driven, long before the term graced conference keynotes. A settled transaction, a proposal changing state, a signed contract, a recalculated credit limit, a suspected fraud, a revoked consent: all are business facts that happened in the real world and that the system must record, propagate, and honor. The architectural question was never 'should we use events?', it was, and remains, 'are these facts explicit contracts with schema, ownership, and traceability, or are they trapped inside coupled synchronous calls that nobody dares evolve anymore?'

My view after 16 years in financial-grade systems: Every time I walk into a bank and see a payment service calling six other services synchronously, each of those calling three more: I recognize the pattern immediately: someone tried to 'decouple' without understanding that real decoupling requires the business fact to be a first-class citizen. The result is the worst of both worlds: the fragility of a distributed system with the temporal coupling of a monolith. Event-driven done right is not about messaging technology; it is about making domain facts explicit, versioned, and auditable. When I ride up to the penthouse to discuss real-time fraud exposure with the Chief Risk Officer, and then ride down to the engine room to review a Kafka topic design, I am doing exactly the same work, translating the same business fact between two vocabularies. The elevator does not stop in the middle.

Business facts are contracts, not notifications

There is a distinction that separates mature banking architectures from those that become operational liabilities within two years: the difference between an event as notification and an event as domain contract.

A notification says: "something happened, do what you want with it." A domain contract says: "the fact `TransactionSettled` occurred at 14:23:07 UTC, with these mandatory fields, in this versioned schema v2.1, emitted by the Settlement domain, and any consumer can depend on this contract without consulting the emitter." The difference is not philosophical, it is operational. When BACEN demands traceability of a foreign exchange operation, you do not want to answer "the event was published, but we have no registered schema and the consumer may have interpreted the `grossAmount` field differently from the emitter."

A well-formed banking domain event carries, at minimum: a unique and immutable fact identity (eventId), a domain-occurrence timestamp (not a queue-publication timestamp), a correlation identifier for cross-system tracing, a schema version, and the owning domain emitter. These fields are not bureaucracy, they are the attributes that enable safe reprocessing, regulatory audit, and independent consumer evolution.

Governance starts at the schema. A Schema Registry, whether the Confluent Schema Registry over MSK, or the AWS Glue Schema Registry, is not an infrastructure tool; it is where the contract between domains is written and versioned. When the Credit team evolves the LimitRecalculated event to include a scoreModelVersion field, the Schema Registry ensures old consumers keep working (backward compatibility) while new consumers can opt into the additional field. Without this, a schema change in production is an unannounced contract change, and in a bank, unannounced contracts become incidents.

Choosing the right backbone: MSK, EventBridge, or SQS

The question "which messaging service should I use?" is, in practice, the wrong question. The right question is: "what is the consumption model, the retention requirement, the coupling pattern, and the governance level that this business fact demands?"

In the banking context, three AWS services dominate the event backbone, and each has a distinct role. Amazon MSK (managed Kafka) is the nervous tissue for high-frequency, high-criticality domain events, settlements, ledger movements, real-time fraud events, where configurable retention, deterministic replay, and consumer groups with explicit offsets are requirements, not differentiators. The operational cost is real: MSK requires careful broker sizing, partition management, and consumer lag monitoring. That cost is justified when volume, latency, and replay needs make any alternative an architectural concession.

Amazon EventBridge excels at integrations between domains and between AWS accounts, the event bus where a domain publishes without knowing who will consume, and where schema-based routing rules allow new consumers to connect without touching the emitter. For a bank building a platform architecture with multiple product teams, EventBridge is the extensibility mechanism: the Onboarding domain publishes ClientApproved and the Card, Account, and CRM domains subscribe independently. The limitation is throughput and the absence of native replay with Kafka-style offset semantics, for very high-volume events, EventBridge is not the right place.

Amazon SQS solves a different problem: reliable point-to-point delivery with native DLQ, message visibility, and trivial Lambda integration. For asynchronous commands: "process this portability request", "send this receipt": SQS is frequently the simplest and most operationally safe choice. The table below compares the three services on the criteria that matter most in a bank.

Choosing the event backbone on AWS

Criterion Amazon MSK / Kafka EventBridge SQS

--- --- ---

Best for High volume, ordering, long replay Domain events, fan-out, rules, SaaS Point-to-point decoupling, peak absorption

Ordering Per partition No strong guarantee Optional FIFO

Retention / replay Days to months, native replay Short (limited archive/replay) Until processed (+ DLQ)

Operational cost Higher, a cluster to operate Low, serverless Very low, serverless

In a bank, use for Transaction streams, ledger feeds Cross-domain events, orchestration Load buffers, legacy integration

Distributed coupling is worse than the monolith: When a service publishes an event without a registered schema, without a configured DLQ, without consumer idempotency, and without a declared owner, the result is not a decoupled system, it is a distributed monolith where implicit contracts are hidden in consumer code that nobody remembers who wrote. In a bank, this has direct regulatory consequence: if you cannot prove what happened to a specific business fact, you do not have an audit, you have hope.

The Transactional Outbox pattern, eliminating the dual write

This is the point where most teams make mistakes, and where the mistake has direct financial consequence. The problem is simple to state and hard to resist: when a service needs to write state to the database and publish an event, the naive solution is to perform both operations in sequence, first INSERT to the database, then publish to Kafka. The problem is that between those two operations there is a failure window. If the process crashes after the INSERT and before the publish, state was written but the event was never emitted. A payment happened in the ledger, but no downstream consumer knew, the balance was debited, the notification never arrived, the reconciliation system never recorded it. The inverse is also possible: the event is published, but the database transaction fails or is rolled back. Now downstream consumers reacted to a fact that does not exist.

The Transactional Outbox pattern resolves this with a fundamental guarantee: state and event are written in the same local database transaction. The service does not publish directly to the broker, it writes to an outbox table within the same ACID transaction that writes the business state. A separate process (the relay or CDC connector) reads that table and publishes to the broker with at-least-once semantics. If the relay fails, it simply re-reads the table and republishes, the event may arrive more than once, but it will never fail to arrive.

The immediate consequence is that consumers must be idempotent: processing the same event twice must produce the same result as processing it once. This is not a limitation of the pattern; it is a property that every banking event consumer should have regardless, because networks fail, brokers redeliver, and systems restart. The combination of Outbox + idempotent consumer delivers the guarantee that financial systems require: no fact is lost and no fact is duplicated in effect.

On AWS, the relay can be implemented with Debezium on Amazon MSK Connect reading the RDS PostgreSQL binlog via CDC, or with a scheduled Lambda process polling the outbox table, the second option is operationally simpler but introduces latency proportional to the polling interval. For settlement events where latency matters, CDC is the right choice. For lower-criticality asynchronous notifications, Lambda polling is acceptable and easier to operate.

Implementing the Transactional Outbox in a bank on AWS

1. Create the outbox table in the same domain RDS instance: The domain_outbox table must have: event_id UUID PRIMARY KEY, aggregate_id, event_type, schema_version, payload JSONB, created_at, published_at (nullable). The index on published_at IS NULL is what the relay uses to find pending events.
2. Write business state and event in the same transaction: In the service code, within the same BEGIN/COMMIT: UPDATE accounts SET balance = ... and INSERT INTO domain_outbox (event_id, event_type, payload) VALUES (...). If the transaction rolls back for any reason, the event disappears too, there is never a divergence.
3. Configure the relay with CDC (Debezium + MSK Connect): The Debezium connector monitors the RDS PostgreSQL binlog and publishes each INSERT on the outbox table as a message to the

corresponding MSK topic. The message key must be the `aggregate_id` to guarantee per-aggregate ordering. Enable logical replication on RDS and configure the replication slot with adequate retention.

4. Guarantee consumer idempotency with a deduplication store: The consumer must check the `event_id` in a deduplication store (DynamoDB with a 7-day TTL is a common choice) before processing. If the `event_id` already exists, the event is silently discarded and the offset is committed. If it does not exist, process and write the `event_id` to the store in the same logical operation.

5. Monitor consumer lag and DLQ as an operational SLA: Consumer lag on MSK is a health indicator for the event flow, not just a performance metric. Growing lag on a settlement topic may mean a downstream service is failing silently. CloudWatch alarms for lag > threshold and for any message in the DLQ must be treated with the same urgency as a 5xx error alarm.

Event governance: ownership, schema, and lifecycle

Technology without governance is automation of chaos. In a banking event-driven architecture, governance means answering three questions for every domain event: who owns it, what is the contract, and what is the lifecycle.

The owner is not the team that created the Kafka topic, it is the business domain responsible for the event's semantics. The Credit domain owns `LimitRecalculated`; the Payments domain owns `TransactionSettled`. This distinction matters during an incident: if the schema changed incompatibly and consumers broke, the owner is who must respond, not the infrastructure team. Without declared ownership, every schema change becomes a political negotiation between teams.

The contract is the versioned schema registered in the Schema Registry, with an explicit compatibility policy. For banking events, I recommend `BACKWARD_TRANSITIVE` as the default: any new version can be read by consumers of any prior version. This allows the emitter to evolve without coordinating with each consumer, which is exactly the decoupling that justifies the event-driven architecture. New fields are optional; existing fields never change type; removed fields go through a deprecation period with an announced sunset version.

The lifecycle defines how long the event exists, who can consume via replay, and what happens when the contract must be broken (major version). In a bank, settlement events have a regulatory retention requirement: BACEN may require operation traceability for years. This means MSK topic retention must be aligned with the institution's data retention policy, and that historical event replay is a legitimate use case the architecture must support, not an accident it tolerates.

When I ride up to the penthouse and the Chief Compliance Officer asks "can we prove what happened with that foreign exchange operation last March?", the answer depends entirely on decisions made in the engine room months earlier: the event was recorded with a domain-occurrence timestamp, the schema was registered, retention was configured correctly, and the relay never lost a message. Event governance is not an infrastructure concern, it is a compliance concern that manifests in infrastructure.

Golden rules for banking domain events

- A domain event is an immutable fact, it describes what happened, never an instruction of what to do. Name it in the past tense: `TransactionSettled`, not `SettleTransaction`.
- Never perform a dual write (database + broker as separate operations). Use Transactional Outbox to guarantee that state and event are atomic at the source.
- Every event needs: a unique `eventId`, a domain-occurrence timestamp, `correlationId`, schema

version, and emitting domain. These fields are not optional in financial systems.

- Every banking event consumer must be idempotent. At-least-once delivery is the real guarantee of any distributed system, design for it, not against it.
- DLQ is not a trash bin, it is an observability contract. Any message in a DLQ is a business fact that was not processed. Treat it with the same urgency as a production error.
- Schema Registry with BACKWARD_TRANSITIVE policy is the mechanism that enables independent evolution of emitters and consumers. Without it, every schema change is a coordination window, and coordination at scale is the enemy of velocity.

Banking event-driven: the architect's verdict

Event-driven in banks is not a modernization choice, it is the belated recognition that business facts have always existed and have always needed to be propagated. The real choice is between propagating those facts explicitly, governed, and auditably, or continuing to hide them in synchronous calls that nobody can evolve anymore. The Transactional Outbox, the Schema Registry, domain ownership, and consumer idempotency are not implementation details, they are the pillars that separate an event-driven architecture the regulator can audit from one the architect has to explain why it failed. Choose the pillars before choosing the broker.

09 - Data as product, lineage and proof

In a bank, data isn't a by-product: it's evidence, a regulatory obligation, an input to risk, and the basis for AI. Mature architecture doesn't just ask where to store, it asks who owns it, what the lineage is, and what the obligation is.

In banking, data is not a system by-product: it is regulatory evidence, risk input, personalization foundation, and, when poorly governed, a liability waiting for the right moment to materialize as a fine, fraud, or model failure. Mature data architecture does not start by choosing between Redshift and Athena, it starts with a domain question most teams never ask out loud: what fact does this data represent, for whom, with what retention obligation, and what is the risk if it leaks?

The question that separates data architecture from pipeline engineering: After sixteen years building platforms in financial institutions, I learned that the biggest mistake is not choosing the wrong technology, it is starting with technology. I have seen petabyte lakes nobody trusted, feature stores without owners, and risk dashboards that contradicted the system of record. The problem was invariably the same: data arrived in the lake as a dump, not as a product. Nobody had answered who the responsible producer is, what the quality contract is, what the lineage back to the source is, and what regulatory obligation that data carries. When the architect rides up to the penthouse and hears 'we want AI for personalization,' the correct technical response is not to provision a SageMaker endpoint, it is to ask whether LGPD consent is segregated, whether behavioral data lineage is auditable, and whether quality is contractualized. Only then does the conversation about models make sense.

Banking data has identity before it has an address

Consider three data objects that coexist in any mid-size Brazilian bank: a Pix transaction, a credit score, and an LGPD consent record. Technically, all three are rows in some database. Architecturally, they are radically different objects.

The Pix transaction is an immutable business fact: it occurred at an instant, has legal value, must be retained for five years per BCB Resolution No. 1 and its derivatives, and any subsequent alteration is fraud, not correction. The credit score is a derived and temporal datum: it is the result of a model applied to signals at a given moment, it expires, and its lineage must be auditable because BACEN may ask, during supervision, why that customer was denied on that date. The LGPD consent record is an access control datum: it does not describe the customer, it authorizes or prohibits other customer data from being used for a specific purpose, and its absence must block entire pipelines.

When the architect does not distinguish these three objects before designing the lake, the outcome is predictable: the Pix transaction ends up overwritten by a poorly written ETL job, the credit score loses its lineage after a schema migration, and behavioral browsing data flows into a personalization model without verifying whether consent for that purpose exists. Each of these errors has a name in the regulatory vocabulary: record tampering, inability to explain automated decisions, and improper use of personal data.

The domain question: what fact, for whom, with what obligation, is not philosophical. It is the first risk control in data architecture.

Data as product: from data lake as dump to governed marketplace

The concept of data product solves the governance problem where purely technical frameworks fail: at the incentive level. When an engineering team delivers data to the lake without a contract, without an SLA, and without a declared owner, the downstream consumer silently assumes the quality risk. Nobody is accountable when data arrives corrupted, late, or undocumented. The lake becomes a dump because the accountability model allows it.

Applying product logic inverts that incentive. Each domain, credit, payments, onboarding, anti-fraud: publishes its data as a product with an explicit contract: versioned schema, update SLA, measurable quality definition, named owner, and access policy. The consumer subscribes to the product, not to the bucket. If quality falls below the contract, the producer is notified and held accountable, exactly as happens with a service API.

In practice, this changes three things in the AWS architecture. First, AWS Glue Data Catalog stops being merely a technical schema catalog and becomes the data product registry: each table has business metadata, owner, sensitivity classification, and SLA. Second, Lake Formation implements attribute-based access control (TBAC/ABAC) that respects access contracts, the marketing team does not access credit score data without explicit approval from the risk domain. Third, AWS Glue jobs feeding the medallion layers (Bronze -> Silver -> Gold) include quality validations as a mandatory step, not optional, data that fails validation does not advance to the next layer.

The diagram below shows how these pieces articulate in the governed data platform I use as a reference for financial institutions on AWS.

What defines a mature banking data product

- Versioned schema published in Glue Catalog with business metadata, not just technical metadata
- Named domain owner, an individual accountable for quality and SLA, not a generic team
- Declared sensitivity classification: personal data, financial data, operational data, public data
- Purpose-based access policy, not just technical role-based access
- Traceable lineage from source to final consumer, auditable by the regulator
- Measurable quality SLA with automated alerts, not a verbal promise of 'reliable data'

Lineage and proof: data without history has no value in an audit

Data lineage is the record of where data came from, what transformations it went through, and who consumed it. In any sector, this is good practice. In banking, it is an obligation. When BACEN questions an automated credit decision, the answer cannot be 'the model said no', it must be 'model version 2.3, trained on data from period X to Y, received these input signals, produced this score, and the decision was made based on this cutoff policy in effect on that date.' Each link in that chain must be traceable.

On AWS, lineage can be implemented in layers. At the infrastructure level, AWS Glue automatically records job transformations and Amazon S3 with versioning enabled preserves each object state. At the catalog level, Glue Data Catalog integrated with Apache Atlas or solutions like Amazon DataZone allows mapping dependencies between tables and jobs. At the model level, Amazon SageMaker Experiments and SageMaker Model Registry record which dataset trained which model version, indispensable for Article 20 of the LGPD (right to explanation of automated decisions).

But technical lineage without process governance is incomplete. I have participated in audits where the technical lineage was perfect on paper, jobs documented, schemas versioned, but nobody could say whether the input data had gone through an anonymization step before feeding the model, because that step was in a manual script outside the official pipeline. Lineage must capture the entire path, including the shortcuts teams create under delivery pressure.

The architect who rides up to the penthouse and hears 'we need to explain our credit decisions to the regulator' must ride down to the engine room and ask: is there an auditable record of every transformation between raw data and the decision? If the answer is no, the conversation about AI models needs to pause.

A governed data platform on AWS

From ingestion to decision, with governance cutting across everything. The point isn't the services, it's that each zone has an explicit owner, lineage and access policy.

Ingestão / Ingestion

- MSK / Kinesis real-time events (messaging)
- Glue / DMS batch & CDC from legacy (compute)

Armazenamento / Storage (medallion)

- S3 Bronze raw & immutable (storage)
- S3 Silver clean & conformed (storage)
- S3 Gold data products (storage)

Governança / Governance

- Lake Formation fine-grained perms (security)
- Glue Catalog lineage & schema (security)

Consumo / Consumption

- Redshift / Athena analytics (data)
- Feature Store risk & fraud (ai)

Flows

- stream -> bronze

- batch -> bronze
- bronze -> silver: quality
- silver -> gold: data products
- If -> gold: governs access
- catalog -> silver: lineage
- gold -> redshift
- gold -> feature: feeds models

Poorly governed data is regulatory risk with an expiration date: LGPD, BACEN, and CMN do not require immediate perfection, they require evidence of control and an improvement trajectory.

But personal data flowing into AI models without verified consent, credit scores without auditable lineage, and transactions without adequate retention are liabilities that accumulate silently. When the incident occurs, a leak, a supervisory challenge, or a data subject complaint, the absence of governance is not a mitigating factor: it is an aggravating one. LGPD fines can reach 2% of revenue. Reputational risk has no ceiling.

From platform to model: what data architecture answers before any AI

When leadership rides up to the penthouse with two objectives: AI personalization and fraud reduction, and asks the architect for a solution, the temptation is to go straight to the machine learning services catalog. I always resist that temptation. My response starts four floors below, in the data engine room.

For AI personalization, the questions data architecture must answer first are: is LGPD consent for the personalization purpose captured, segregated, and verifiable at pipeline execution time? Is behavioral data separated from financial transaction data, or are they mixed in a schema that makes it impossible to apply different controls? Is the quality of behavioral data sufficient, what is the rate of lost events, what is the average ingestion delay? Is the cost of keeping that data hot for feature serving compatible with the expected model value? None of these questions are answered by the data science team, they are data architecture questions with business and regulatory consequences.

For real-time fraud reduction, the problem is different but equally structural. Fraud requires signals in milliseconds, not in batch hours. This means the Feature Store (Amazon SageMaker Feature Store with online store enabled) must be fed by event streams, not nightly jobs. It means the pipeline from Chapter 8, the event-driven nervous tissue, must be integrated with the data layer so that a Pix transaction event triggers feature updates before the approval decision is made. And it means human intervention must be designed into the flow: when the model flags fraud with low confidence, is there an operational review process? Who decides? In how much time? Is that process auditable?

The governed data platform, with its Bronze, Silver, and Gold layers, with Lake Formation controlling access, with Glue Catalog recording lineage, and with Redshift and Athena serving different consumption layers, is not the destination. It is the infrastructure that makes it possible to answer these questions with evidence, not with hope.

How to start a governed data architecture in banking, pragmatic sequence

1. Inventory domains and classify data by obligation: Before any infrastructure, map business domains and, for each relevant data type, answer: is it personal data (LGPD), financial data (BACEN), operational data, or public data? What is the minimum and maximum retention? Who is

the responsible producer? This inventory is the input for all subsequent architecture decisions.

2. Define the ownership model and product contract: Each domain names a responsible data owner. That owner signs a product contract: versioned schema, update SLA, measurable quality criteria, and purpose-based access policy. Without a contract, there is no product, there is only loose data.

3. Implement Lake Formation with attribute-based access control: Configure AWS Lake Formation as the central access control point for the lake. Use sensitivity tags (PII, financial, operational) and purpose policies to ensure access is granted based on who needs the data and why, not just what technical role the user has.

4. Instrument lineage in Glue Catalog and transformation jobs: Enable lineage tracking in AWS Glue and enrich the Glue Catalog with business metadata. For each transformation job, explicitly record sources, applied rules, and destination. Consider Amazon DataZone for catalog governance at scale with multiple domains.

5. Validate quality as a layer gate, not a post-hoc report: Implement quality validations (completeness, uniqueness, schema consistency, business rules) as mandatory steps in Glue jobs that promote data from Bronze to Silver and Silver to Gold. Data that fails validation does not advance, it is routed to quarantine with an alert to the product owner.

Frequently asked questions about data architecture in banks

Redshift or Athena, which to use for analytics in banking?

That is the wrong question to start with. The answer depends on the access pattern: Redshift for complex, frequent queries over structured data with predictable latency (regulatory reports, operational dashboards); Athena for ad-hoc exploration, forensic audit, and queries over semi-structured data in S3. In mature platforms, both coexist: Redshift serves the Gold layer for recurring consumers, Athena serves the Silver layer for analysts exploring data. What determines the choice is the consumer's SLA and access pattern, not the data team's preference.

Is data mesh suitable for Brazilian banks?

Data mesh is an organizational model, not a technology. Its principles, domain ownership, data as product, self-service platform, federated governance, are highly compatible with banks that already have well-defined business domains. The challenge in Brazilian banks is federated governance: LGPD and BACEN regulations require centralized access and retention controls that must be implemented consistently even when ownership is distributed. The solution is a centralized governance layer (Lake Formation + corporate policies) over distributed data ownership, not one or the other.

How to handle third-party data (credit bureaus, Open Finance) in the platform?

Third-party data has its own contractual and regulatory obligations that must be reflected in the data product classification. Open Finance data, for example, has data subject consent with defined scope and expiration, the ingestion pipeline must verify whether consent is still valid before each use, not just at ingestion. Credit bureau data has redistribution restrictions that must be in the product's access policy. The practical rule is: third-party data enters the Bronze layer with explicit origin, contract, and usage restriction metadata, and never advances to Gold without those restrictions being verified.

What separates a mature banking data platform from an immature one

An immature platform stores data and hopes someone uses it well. A mature platform publishes

data products with contracts, auditable lineage, measurable quality, and purpose-based access control, and treats poorly governed data as the regulatory risk it is. The difference is not in the technology chosen: it is in who is responsible, for what, with what evidence. When the architect can ride up to the penthouse and say 'our credit decision from yesterday is explainable, auditable, and defensible before BACEN,' it is because the engine room was built with that requirement in mind from day one.

10 - Platform and runtime: choose by operating model

EKS, ECS, Lambda or EC2? The mature decision doesn't start with a favorite technology, it starts with the workload and the team's operational maturity. And it ends in golden paths, not preferences.

The question 'EKS or Lambda?' comes up in nearly every platform discussion I have been part of, and it is almost always the wrong question. What truly matters is who will operate this at three in the morning, how the team deploys without fear, and how the organization recovers from an incident before the regulator notices. Runtime is a consequence of those answers, not the starting point.

The wrong floor to start the decision

When the CTO of a bank rides up to the penthouse, he is not thinking about Kubernetes pods or Lambda functions. He is thinking about operational risk, about maintenance windows that the Central Bank requires documented, about the cost of a payment system failure at 23:59 on a Friday night. When the engineer descends to the engine room, he thinks about Docker images, cold starts, and concurrency limits. The architect who rides the elevator must translate both languages without losing either.

The classic mistake is starting the platform decision in the middle of the building, at the layer of technology preference. A team that grew up with Kubernetes will want EKS for everything. A team formed by ex-serverless startup engineers will want Lambda for everything. Neither position is wrong in itself; both are dangerous when they become universal.

The mature decision starts with three business questions that come directly from the penthouse: What is the failure tolerance of this domain? A customer onboarding domain can tolerate degradation for minutes; an interbank settlement domain cannot. What is the load pattern? Predictable, short spikes favor serverless; sustained, uniform load favors containers. What is the operational maturity of the responsible team? A team without Kubernetes experience that inherits a production EKS cluster has not gained power, it has gained responsibility without preparation.

Only after those answers does it make sense to open the decision matrix and map the workload to the appropriate runtime. As the table below shows, each combination of banking domain and runtime carries a distinct set of operational verdicts, and ignoring those verdicts is the most expensive way to learn architecture.

My position after 16 years in financial systems: I have never seen a mid-to-large bank that ran well on a single runtime. Every real banking architecture I have audited or designed is hybrid: digital channels on containerized BFFs, critical settlement and accounting domains on ECS or EKS with progressive deployment and fine rollback control, asynchronous event-driven processes on Lambda and Step Functions, hot data on Aurora and DynamoDB, cold data on S3. The question was never 'which runtime wins the debate', it was 'which runtime best serves this specific workload, operated by this specific team, within this risk envelope'. Anyone insisting on

uniformity is optimizing for the architect's comfort, not for the bank's resilience.

The decision matrix and what it actually measures

The decision matrix accompanying this chapter: [DECISION MATRIX], organizes the main runtimes available on AWS (EKS, ECS/Fargate, Lambda/Step Functions) against the dimensions that matter most in a banking context: operational responsibility model, attack surface and security posture, latency and behavior under load, cost of failure and recovery strategy, and speed of change delivery.

What the matrix does not do, and it is important to be explicit, is declare an absolute winner. It maps conditional verdicts. EKS wins when the team has platform maturity, the workload requires fine-grained scheduling control, and the organization already operates Kubernetes in other contexts. ECS/Fargate wins when the team wants container semantics without the overhead of managing the control plane, a trade-off that makes sense for business domains that are not the core of the engineering platform. Lambda/Step Functions win when the invocation pattern is sporadic or event-driven, when the team wants to pay per execution rather than reserved capacity, and when the orchestration logic for long processes needs durability without manually managing queues.

In the context of BACEN and CMN, there is an additional dimension that most decision frameworks ignore: deploy auditability. Brazilian regulation requires that changes to critical systems be traceable, with evidence of approval and documented rollback. EKS and ECS with GitOps (ArgoCD, Flux) deliver this naturally via manifest history. Lambda with SAM or CDK delivers it via CloudFormation change sets. The difference is not in the runtime, it is in how the CI/CD pipeline is built on top of it. This means the runtime choice and the deploy strategy choice are coupled decisions, and treating them separately is a design error.

A practical detail I learned the hard way: Lambda cold start is not just latency, it is SLA risk. In payment domains where the end-to-end response SLA is 500ms, an 800ms cold start in an uninitialized Java function breaks the contract. The solution is not to abandon Lambda, it is to use Provisioned Concurrency, SnapStart for JVM, or simply choose a lighter execution runtime. But this needs to be in the decision, not discovered in production.

Choosing the runtime for a banking domain

Amazon EKS (Kubernetes)

Pros

- Multi-team standardization and portability
- Fine runtime control, service mesh, operators
- Strong for constant platform scale

Cons

- High operational load and required maturity
- Cluster cost even when idle
- Complexity that becomes risk for a small team

Verdict: When there's platform scale, multiple teams and real operational capacity to run Kubernetes.

Amazon ECS / Fargate

Pros

- Containers with far less load than Kubernetes
- Native integration with the AWS ecosystem
- A good middle ground for teams without dedicated SRE

Cons

- Less flexible than EKS for advanced cases
- Smaller third-party ecosystem

Verdict: When the team wants containers and predictability without taking on Kubernetes complexity.

Lambda + Step Functions

Pros

- Minimal operations, automatic scaling
- Ideal for event-driven flows and integration
- Zero cost when idle

Cons

- Demands discipline in idempotency and limits
- Cold starts and execution ceilings
- Observability and IAM need rigor

Verdict: When the flow is event-driven and intermittent, and low operations matter more than fine control.

Golden paths: when the right way is the easiest way

There is a permanent tension between product team autonomy and platform governance. Product teams want to move fast, choose tools, experiment. The platform wants to standardize, audit, control costs. In banks, this tension is amplified by the regulator: any deviation from standard can become an audit finding.

The solution that works, and that I have seen work in financial organizations that managed to scale engineering without losing control, is the concept of golden paths: paved roads that make the correct choice the easiest choice. A golden path does not prohibit alternatives; it makes them more costly in effort. If the service template already comes with observability configured, SAST in the pipeline, minimal IAM policy, and service mesh integration, the team that decides to leave that template needs to justify it and bear the maintenance cost of the deviation.

In practice, a banking golden path on AWS has at least five components: service template (repository with project structure, Dockerfile or SAM template, pre-configured CI/CD pipeline); observability by default (CloudWatch structured logs, X-Ray tracing, business metrics via EMF already configured in the template); security by default (IAM roles with least privilege, secrets via Secrets Manager, image scanning in ECR, dependency scanning in the pipeline); policy as code (SCPs in AWS Organizations, Config Rules, OPA/Rego for Kubernetes manifest validation if EKS is the chosen runtime); and incident runbook (documentation on how to escalate, how to roll back, who to contact, integrated with the on-call system).

The most valuable side effect of golden paths is cultural: governance stops being a fight. When the safe path is also the fast path, the product team does not experience the platform as an obstacle, they experience it as an accelerator. This changes the dynamics of the entire engineering organization, and is especially critical in banks where regulatory pressure already creates enough friction.

How to start a golden path without a mature platform: Do not wait for a complete Internal Developer Platform to get started. Begin with a single repository template on GitHub/CodeCommit that already solves the five most painful points for the team: project structure, basic pipeline with SAST, environment variables via Secrets Manager, structured logging, and a runbook README. That simple template is already a golden path. Iterate on it every sprint based on what teams complain about. A platform is not a project, it is an internal product with real users.

The operational model is the real decision

Returning to the elevator: when rising to the penthouse with the runtime decision, what the board and CRO of a bank need to understand is not which AWS service was chosen. They need to understand who is responsible for what when something fails. That is the translation the architect must make.

With EKS, the platform team is responsible for the AWS-managed control plane, node groups, CNI, ingress controller, service mesh, and network policy. The product team is responsible for the application, Dockerfile, Kubernetes manifest, and business logic. When a pod crashes at 3am, who wakes up? When the cluster needs a Kubernetes version upgrade, who plans the maintenance window and communicates it to the regulator?

With ECS/Fargate, the model shifts: AWS manages the control plane and compute infrastructure provisioning. The platform team is still responsible for task definition, VPC networking, IAM policies, and deploy strategy. The product team is responsible for the image and logic. The infrastructure on-call is simpler, but the cost per compute unit is generally higher than equivalent EC2, and scheduling control is lower.

With Lambda, the model goes further: AWS manages execution, scaling, and function availability. The team is responsible for code, timeout, configured memory, retry policy, and dead-letter queue. Infrastructure on-call practically disappears, but business logic on-call increases, because silent failures in asynchronous processing are harder to detect without well-configured observability.

The runtime choice, therefore, is a choice of where operational responsibility concentrates. Banks with large, mature platform teams can absorb EKS. Smaller digital banks with agile product teams but no dedicated platform engineering benefit from ECS/Fargate and Lambda. And most real banks live in the middle: a deliberate combination that maps responsibility to the maturity level of each team, domain by domain. That deliberate combination, not uniformity, is the signal of a mature architecture.

Key takeaways from this chapter

- Runtime choice starts with business questions, failure tolerance, load pattern, team operational maturity, not technology preference.
- Real banking architectures are hybrid: BFFs in containers, critical domains in EKS/ECS, event-driven processes in Lambda/Step Functions, data in Aurora/DynamoDB/S3.
- Runtime choice and deploy strategy are coupled decisions; change auditability is a regulatory requirement in Brazil, not optional.
- Golden paths make the correct choice the easiest one: template, observability, security, policy-as-code, and runbook by default, not by additional effort.
- The operational model, who operates at 3am, how deployment works, how recovery happens, is the real decision; the runtime is the technical answer to that model.

- Runtime uniformity optimizes for architect comfort; deliberate combination by domain optimizes for bank resilience.

Chapter verdict

There is no universal answer to 'which runtime to use'. There is a universal question that must be answered first: what is the operational model this domain requires and that this team can sustain? When that question is answered honestly, considering real maturity, not aspirational maturity, the runtime choice becomes a natural consequence, not a battlefield. Golden paths turn that consequence into a reproducible standard. And reproducible standards are what separates a banking platform that scales from a collection of individual heroes who never sleep.

11 - Generative AI with guardrails: value without a black box

Generative AI brings knowledge, automation and decision closer, but in a bank it only creates sustainable value with security, evaluation, traceability and limits. Bedrock helps; the architecture defines the boundaries.

Generative AI in a bank is not a feature, it is a capability with an owner, a risk policy, an SLO, and a fallback plan, or it is nothing more than an eternal pilot that never reaches production. The language model is the easiest component to replace; the guardrails surrounding it are what protect the bank, the customer, and the operating license. In this chapter I close Part III by showing how architecture descends to embeddings, tools, and authorization logs, and rises back to the penthouse as measurable productivity, service scale, and regulatory evidence.

My position after 16 years in financial systems: Every time I present generative AI to a banking risk committee, the first question is not 'which model?', it is 'who is accountable when it goes wrong?' That question is exactly the right one. In every project I have architected with Bedrock, the differentiator was not the LLM choice; it was the discipline of treating the AI system like any other critical service: prioritized backlog, defined SLO, telemetry from day zero, prompt versioning as code, and continuous evaluation with real datasets. Banks that skip this discipline and jump straight to the copilot demo are building technical and regulatory debt simultaneously. My thesis is simple: a guardrail without telemetry is a promise without proof; RAG without curation is expensive search with an intelligent appearance; an agent without contextual authorization is operational risk waiting for an incident to materialize.

The elevator rises with value, descends with accountability

In the penthouse, the executive sees three concrete promises of generative AI: a service copilot that reduces average resolution time; a contract analyzer that accelerates legal onboarding; an internal assistant that democratizes compliance knowledge. These promises are legitimate. The problem begins when the architect does not ride the elevator down to show what supports each of them in the engine room.

When I descend, I find four layers that must exist before any model is called in production. The first is governed data: RAG is only trustworthy if the knowledge base has curation, versioning, and traceable lineage, which connects directly to Chapter 9, where I treated data as a product. The second is contextual authorization: every action an agent executes must carry the user's identity, permission scope, and session context; without this, the agent acts as a user with excessive privileges. The third is quality telemetry: every inference must be logged with versioned prompt,

response, latency, tokens consumed, and, when possible, explicit or implicit user feedback. The fourth is operationalized risk policy: not a governance document in a drawer, but guardrails that are configured, tested, monitored, and have active alerts.

This descent is not bureaucracy, it is what allows the architect to rise back to the penthouse with evidence, not hope. The risk committee does not want to know whether the model is good; it wants to know whether the system is auditable, whether the cost is predictable, and whether there is a human in the loop when a decision carries regulatory consequence.

How Amazon Bedrock structures the boundaries, and why the model is the most interchangeable component

As the diagram below shows, the reference architecture I use in banking projects with Bedrock is not organized around the model, it is organized around the boundaries. Guardrails at input and output, RAG over governed data with Knowledge Bases, agent actions restricted by contextual authorization, and telemetry captured at every layer.

Bedrock Guardrails operationalize policies that would otherwise exist only in documents: content filters configurable by category and intensity, PII detection and masking before any sensitive data reaches the model or returns to the user, blocking of prohibited topics (such as unregulated investment advice), and protection against prompt injection, an underestimated attack vector in banking environments where the agent has access to real tools. Every guardrail decision is logged, which transforms policy into auditable evidence.

Knowledge Bases solve the problem of RAG without curation. Instead of indexing raw documents and hoping that retrieval will be relevant, the architecture requires every knowledge source to pass through an ingestion pipeline with validation, versioning, and lineage metadata. The model does not access data; it accesses curated fragments with traceable provenance. This is the difference between a system an auditor can inspect and one they must trust blindly.

Agents and AgentCore close the automation loop, but with a non-negotiable architectural constraint: no tool is called without the user's authorization context being validated at execution time. The agent does not inherit system permissions, it carries the user context and session scope, and every tool call is recorded with those attributes. This is what separates a banking agent from a script with an LLM in front of it.

The three sins of generative AI in banking

In architecture projects, antipatterns teach more than patterns because they show where pressure for speed defeats engineering discipline. In banking generative AI, I identify three recurring sins.

First sin: RAG without curation. The team indexes regulatory documents, product manuals, and internal policies without a quality pipeline. The result is a knowledge base that mixes outdated versions of regulations with valid documents, without validity metadata. The model retrieves plausible but incorrect fragments, and the user receives responses with an authoritative appearance based on obsolete information. The inference cost is real; the response quality is illusory. Curation is not optional, it is the SLO of RAG.

Second sin: agent without contextual authorization. The agent is configured with service credentials that have broad access to bank APIs, and user authorization is checked only at the presentation layer. When the agent chains tool calls, check balance, verify limit, initiate transfer, it operates with privileges the authenticated user would not have if accessing the APIs directly. This is not just a security risk; it is a violation of the least-privilege principle that any security audit will identify. The

authorization context must descend with the request, all the way to the last tool called.

Third sin: guardrail without telemetry. The team configures content filters and presents them to the risk committee as evidence of control. But without structured logging of guardrail decisions, how many requests were blocked, by which category, with what content masked, the policy exists only as configuration, not as proof. At the first BACEN audit on AI use in customer service, the question will be: 'show me the control logs for the last 90 days'. Without telemetry, the answer is silence.

RAG and agents in the bank with Amazon Bedrock

Every interaction passes guardrails on input and output, retrieves governed context, and only executes actions with contextual authorization and telemetry. The model is the easiest component to swap; the boundaries are what protect the bank.

Fronteira de IA / AI boundary

- Bedrock Guardrails PII - injection - policy (security)
- Budget + blacklist daily token limit (security)

Orquestração / Orchestration

- AgentCore / orchestrator (ai)
- Bedrock model (swappable) (ai)

Conhecimento / Knowledge

- Knowledge Base RAG on governed data (data)
- Tools domain APIs (authz) (compute)

Evidência / Evidence

- Telemetry + eval logs - quality - cost (ci)

Flows

- user -> guard: asks
- guard -> budget
- guard -> agent: if approved
- agent -> model: infers
- agent -> kb: retrieves context
- agent -> tools: action with authz
- agent -> telemetry: logs everything
- model -> guard: filters output

Generative AI as a banking capability: the non-negotiable pillars

- Capability owner with backlog, SLO, and documented risk policy, not an innovation project without accountability.
- Prompt versioning as code: every version tracked, tested against evaluation datasets, and promoted through a pipeline.
- Daily token budget with alerts and circuit breaker: AI cost is operational cost, not a surprise at month end.
- Blacklist of prohibited terms and topics configured and tested, not assumed to be covered by the base model.
- Explicit fallback plan: what the system does when the model is unavailable, when the guardrail

blocks, when response confidence is below threshold.

- Continuous evaluation with labeled datasets: quality is not subjective perception, it is a metric with baseline and trend.

Cost control and continuous evaluation: the discipline that separates pilot from production

One of the clearest signs that a generative AI implementation is not ready for banking production is the absence of a token budget as an operational control. A token is a unit of both cost and risk: a misconfigured agent can consume in minutes what was budgeted for a day, and a successful prompt injection can force the model to generate long, costly responses in a loop. The daily token budget with progressive alerts and automatic circuit breaker is not a cost optimization, it is a risk control.

In the architecture I use, the budget is implemented in two layers. The first is at the AWS account level, with billing alerts configured to fire before the limit, not after. The second is at the application level, with a per-session and per-user token counter that interrupts the interaction when the threshold is reached and logs the event as an anomaly to be investigated. This covers both the financial risk and the abuse vector.

Continuous evaluation is the mechanism that closes the quality loop. Instead of relying on the product team's qualitative perception, the architecture maintains a set of evaluation datasets with representative cases from the banking domain, product questions, compliance scenarios, balance inquiries, service situations, with expected answers annotated by domain experts. With each new prompt version, the evaluation pipeline runs automatically, calculates quality metrics (factual accuracy, estimated hallucination rate, policy adherence), and blocks promotion if thresholds are not met. This is what transforms prompt versioning from an engineering practice into regulatory evidence: every version in production has an associated quality scorecard, with date, dataset, and result.

How to treat generative AI as a banking capability, deployment sequence

1. Define the capability, not the feature: Before choosing a model or framework, document: which business problem this capability solves, who owns it, what the quality and availability SLO is, and what the associated risk policy is. Without this, the project is born as a pilot and dies as a pilot.
2. Build the governed data foundation before RAG: Define the ingestion pipeline with quality validation, document versioning, validity metadata, and traceable lineage. Bedrock Knowledge Base is the destination; data curation is the prerequisite.
3. Configure and test guardrails before any demo: Implement content filters, PII detection, topic blacklist, and prompt injection protection. Test with real adversarial cases. Enable structured logging of all guardrail decisions from day one.
4. Implement contextual authorization in agents: Every tool called by the agent must receive and validate the user's authorization context. Never use broad-scope service credentials as a substitute for per-user authorization. Log identity, scope, and timestamp on every call.
5. Establish token budget and continuous evaluation: Configure daily budget with alerts and circuit breaker. Create the initial evaluation dataset with domain experts. Integrate automatic evaluation into the prompt version promotion pipeline.

What the risk committee always asks about generative AI

How do we ensure that customer data (PII) does not leak to the model or to other users?

Bedrock Guardrails detect and mask PII at input before the data reaches the model, and at output before the response reaches the user. Additionally, the RAG architecture does not store customer data in the Knowledge Base, it stores only policy and product documents. Transactional data is accessed via agent tools with contextual authorization, never injected directly into the prompt. Every masking operation is logged with timestamp and category, generating an auditable trail.

If the AI makes an error in a credit analysis, who is accountable? Can we use AI in regulated decisions?

In regulated flows, credit, KYC, anti-money laundering, generative AI acts as an assistant, not a decision-maker. The model can synthesize information, highlight inconsistencies, and suggest due diligence questions, but the final decision always belongs to a human who is identified and recorded in the system. This is not a technology limitation; it is a governance requirement that BACEN and the LGPD already signal as an expectation. The architect must ensure that the technical flow reflects this distinction: the model output is a recommendation with a confidence level, not an executable instruction.

How do we control cost? Generative AI can generate billing surprises.

Cost control is architectural, not just financial. We implement a daily token budget with three layers: billing alerts at the AWS account level (firing at 70% and 90% of the limit), a per-session token counter in the application with automatic interruption at threshold, and weekly review of consumption patterns to identify anomalous sessions. Additionally, prompt versioning with cost evaluation per version allows identification of whether a prompt change increased average token consumption without proportional quality gain.

How do we prove that the AI system quality is adequate and maintained over time?

Quality in generative AI is not perception, it is a metric with baseline, trend, and promotion threshold. The architecture maintains evaluation datasets with cases annotated by banking domain experts, organized by category (product, compliance, service). With each new prompt version, the CI/CD pipeline runs automatic evaluation and calculates metrics for factual accuracy, policy adherence, and out-of-scope response rate. The version is only promoted to production if it meets the defined thresholds. The evaluation history is retained as regulatory evidence, associated with the prompt version and dataset used.

The model is the easiest component to replace, the boundaries are the asset

I close this chapter, and Part III, with the statement that most unsettles teams passionate about language models: the LLM is the component with the lowest replacement cost in the entire architecture. Models evolve, new ones are released, benchmarks change, prices fall. What is not easy to replace is the curated knowledge base, the evaluation pipeline with proprietary datasets, the guardrails calibrated for the bank's regulatory context, the structured telemetry that accumulates months of auditable evidence, and the risk committee's trust built through incidents that did not happen.

This inversion of perspective is what separates a banking generative AI architecture from an API

wrapper with a nice interface. The wrapper can be built in days; the architecture takes months because the boundaries need to be designed, tested, operationalized, and proven. But when the next higher-performing model is released, the swap takes hours, because the boundaries are already there.

In Hohpe's elevator, generative AI rises to the penthouse as service productivity, scale without proportional hiring, acceleration of legal processes, and democratization of compliance knowledge. It descends to the engine room as embeddings indexed over governed data, tools with contextual authorization, guardrails with structured telemetry, token budgets with circuit breakers, and continuous evaluation pipelines with proprietary datasets. The architect who can make this translation in both directions, and document every decision with trade-off reasoning, is what the bank needs. Not the one who knows which model has the best benchmark this week.

Generative AI in banking: when it becomes real architecture

Generative AI becomes real banking architecture when it has an owner, an SLO, an operationalized risk policy, telemetry from day zero, prompt versioning as code, continuous evaluation with proprietary datasets, and a tested fallback plan. Amazon Bedrock offers the right building blocks: Guardrails, Knowledge Bases, Agents, but it is architectural discipline that transforms those blocks into an auditable, predictable, and evolvable system. The model is the easiest component to replace; the boundaries are the asset the bank builds over time. Any implementation that cannot answer the four risk committee questions in this chapter is still a pilot, regardless of how many users are already using it.

PART IV

Security, Regulation and Operations

What separates a pretty diagram from a real banking system: security as evidence, compliance as a design constraint, and operations as the only place where architecture truly exists.

12 - Security as evidence, not as opinion

Banks must prove: who accessed, when, why, with what privilege, on which data. Security in financial architecture is less an isolated checklist and more continuous evidence, built into the design.

Security in banking architecture is not a layer you add at the end, it is a design constraint that cuts across identity, cryptography, logging, retention, networking, and incident response from the very first diagram. The question the regulator asks is not 'do you look secure?' but rather 'can you prove who accessed that data, when, with what privilege, through which channel, and with what compensating control active at that moment?', and the answer must exist as immutable, correlatable evidence retained for the required period.

My take: security as evidence is a design posture, not a compliance posture: After more than sixteen years working with financial systems, the single distinction that most separates architectures that survive audits from those that collapse under regulatory pressure is simple: the former were designed to produce evidence, the latter were designed to function and then tried to retrofit evidence. When a bank suffers an incident and BACEN or a DREX-audit requests the

complete trail of a transaction, who initiated it, who approved it, which key encrypted it, which AML rule was evaluated, which personal data was accessed and by whom, that trail either exists in structured, immutable form, or it does not. There is no middle ground. Treating compliance as a final bureaucratic layer is the most expensive architectural decision a bank can make, because the cost of rewriting logging, retention, and access control under the pressure of a real incident is an order of magnitude higher than having them embedded from the start.

Compliance is a design constraint, not a delivery checklist

When the architect rides up to the penthouse and speaks with the Chief Risk Officer about regulatory obligations, they hear phrases like 'retain AML evidence for five years', 'segregate data by jurisdiction', 'prove who approved each operation above a certain threshold'. These phrases sound like compliance requirements. But when the architect descends to the engine room and begins designing, they realize that each one is, in fact, a non-functional requirement that changes concrete architectural decisions.

'Retain AML evidence for five years' implies: storage choice with immutable retention policy (S3 Object Lock in COMPLIANCE mode), definition of which events constitute AML evidence, an indexing schema that enables retrieval by customer, period, and operation type within the timeframe required for regulatory response, and access control that prevents even the administrator from deleting records before the deadline. This cuts across storage, IAM, event schema, and operational process.

'Segregate data by jurisdiction' implies: multiple AWS accounts organized via AWS Organizations with SCPs that prohibit cross-region replication of sensitive data, KMS with per-region keys without export permission, and data pipelines that respect jurisdictional boundaries as an invariant, not as an optional configuration.

'Prove who approved' implies: each approval is a signed event, with federated identity traceable to the human user via IAM Identity Center, an auditable timestamp in CloudTrail, and correlation with the business event it authorized. A separate approval log is not enough, it must be correlatable with the actual transaction.

The architect who treats these constraints as a final layer invariably discovers that they cut across every component of the system. Rewriting under audit pressure is the worst possible moment to learn this.

Security as opinion: the bank loses: The most dangerous phrase I have ever heard in a banking architecture review is 'I think it is secure enough'. The right question is never whether it looks secure, it is whether you can prove it. A control you cannot audit is a promise, not a protection. When the regulator arrives, when the incident happens, when the external auditor requests evidence, 'I think' is not an acceptable answer. The difference between a bank that navigates a regulatory audit with confidence and one that enters panic mode is precisely this: the first has structured, immutable evidence; the second has undocumented intention.

HSM in practice: the system requests signatures, never the key

The most important principle of cryptographic key management in financial systems is so simple it is easy to overlook: the key never leaves the hardware. Keys that sign transactions, that generate card cryptograms, that protect communication with BACEN, these keys are born inside the hardware, live inside the hardware, and die inside the hardware. The system never sees the key material; it sends data to the HSM and receives back a signature, a cryptogram, or encrypted data.

In the AWS context, this materializes in two complementary forms. AWS KMS offers a managed

model where keys reside in HSMs operated by AWS, with support for CMKs (Customer Managed Keys) that allow the bank to control usage policy, rotation, and audit via CloudTrail, every call to kms:Decrypt, kms:GenerateDataKey, or kms:Sign generates a record with caller identity, accessed resource, and result. AWS CloudHSM offers dedicated, single-tenant HSMs where the bank holds exclusive control of key material: AWS has no access. For operations requiring FIPS 140-2 Level 3 certification with exclusive bank custody, CloudHSM is the path; for the vast majority of envelope encryption and token signing operations, KMS with CMKs is sufficient and operationally simpler.

A pattern I use consistently in payment systems: envelope encryption with a key hierarchy. The Data Encryption Key (DEK) encrypts the data; the Key Encryption Key (KEK) encrypts the DEK; the KEK lives in KMS or CloudHSM. Data at rest is never exposed without an explicit, audited call to KMS. When the regulator asks 'who decrypted that customer record on March 14th at 03:47?', the answer is in CloudTrail: identity, resource, timestamp, result, immutable, because the CloudTrail log bucket is protected with S3 Object Lock and the encryption key for that bucket has a policy that prohibits deletion.

For card cryptogram generation (EMV, tokenization), the standard is even more restrictive: the HSM executes the algorithm internally, and the authorization system sends only the input data and receives the output cryptogram. No application software component touches sensitive cryptographic material. This is not paranoia, it is the model that card scheme networks (Visa, Mastercard) and BACEN require.

Zero trust and evidence on AWS

Every access is verified, every key is managed, every action is recorded immutably. Security isn't a wall at the edge, it's a property of every layer.

Identidade / Identity & Access

- IAM Identity Center least privilege (security)
- Organizations + SCPs account guardrails (security)

Proteção de Dados / Data protection

- KMS + HSM keys & encryption (security)
- Secrets Manager (security)
- Macie data classification (security)

Evidência / Evidence & Posture

- CloudTrail immutable trail (ci)
- Config continuous compliance (ci)
- Security Hub + GuardDuty (security)

Flows

- idc -> kms: key access
- scp -> idc: limits
- kms -> trail: use logged
- config -> hub: findings
- macie -> hub: sensitive data
- hub -> trail: correlates

Zero trust as an architecture of continuous evidence

The diagram that follows: Zero Trust and Evidence on AWS, materializes the model I describe in this chapter. But before analyzing it, it is important to understand that zero trust in banks is not a product you buy; it is an architectural posture that translates into concrete decisions at every layer.

At the identity level, the starting point is IAM Identity Center with federation to the bank's corporate directory. No human user has direct access via IAM users with long-lived keys, every session is federated, time-limited, and every action is traceable to the individual. SCPs (Service Control Policies) in AWS Organizations function as non-negotiable guardrails: even if an IAM policy permits an action, the SCP can block it at the organizational level. I use SCPs to prohibit, for example, disabling CloudTrail, creating IAM users with console access without MFA, or replicating production data to development accounts.

At the data level, Amazon Macie continuously scans S3 buckets for uncatalogued personal and financial data, not as a periodic audit, but as a continuous detector. When Macie finds a CPF or card number in a bucket that should not contain them, that is a signal that a data pipeline has leaked information to the wrong place. Integrated with Security Hub, this finding becomes an alert with severity, owner, and remediation deadline.

AWS Config records the state of every resource and its changes over time. When the regulator asks 'what was the configuration of the payment server's security group on March 14th?', Config has the answer, not as human memory, but as an immutable record. Config rules detect compliance deviations in real time: an S3 bucket without encryption, a security group with port 22 open to 0.0.0.0/0, an EC2 instance without a data classification tag.

GuardDuty analyzes VPC Flow Logs, DNS logs, and CloudTrail to detect anomalous behavior, an instance making calls to known C2 IPs, an IAM user running ListBuckets across all regions at 3 AM, a data access pattern that diverges from the historical baseline. This does not replace preventive controls, but it closes the loop: prevention + detection + evidence = auditable security posture.

The central point the diagram illustrates is that each component: IAM Identity Center, Organizations/SCPs, KMS, Secrets Manager, Macie, CloudTrail, Config, Security Hub, GuardDuty, is not an isolated security tool. It is an evidence producer that feeds a continuous, correlatable record of the environment's security state. The integration between them is what transforms tools into architecture.

The principles of security as evidence

- Compliance is a design constraint from diagram zero, treating it as a final layer guarantees rewriting under audit pressure.
- The cryptographic key never leaves the hardware: the system requests signatures and cryptograms from the HSM, never the key material.
- Every relevant access generates an immutable, correlatable record retained for the regulatory period: CloudTrail with S3 Object Lock is the minimum floor.
- SCPs in AWS Organizations are non-negotiable guardrails: they operate above any IAM policy and cannot be bypassed by an individual account.
- A control you cannot audit is a promise, not a protection, if there is no evidence, the control does not exist for the regulator.
- Zero trust is not a product, it is a posture: every security component must also be an evidence producer integrated into the detection and response cycle.

Auditability as a first-class capability

The architect who rides up to the penthouse to discuss regulatory risk and descends to the engine room to design data pipelines must maintain a clear thread: every action relevant to the business must generate a record that survives the worst-case scenario. Not just the worst technical scenario, component failure, data corruption, but the worst regulatory scenario: a security incident investigated by BACEN, an AML audit, a legal action requiring reconstruction of the chain of custody for a transaction.

Auditability in banking systems has three dimensions that must be explicitly addressed in the design:

Completeness: what needs to be recorded? Not just errors and exceptions, every read operation on customer data, every credit approval, every configuration change in a production system, every access to a cryptographic key. The scope of what constitutes a 'relevant action' must be defined with the compliance team before the first development sprint, not after.

Immutability: the record must be tamper-proof, including by internal administrators. S3 Object Lock in COMPLIANCE mode prevents deletion or modification even by users with administrative permissions, the only way to remove the lock is through a formal process with AWS approval. CloudTrail with log file integrity validation detects any attempt at retroactive modification. Logs sent to a separate AWS account, with restricted access and an SCP that prohibits deletion, create a second line of defense.

Correlatability: an isolated log does not tell a story. The regulator does not just want to know that 'user X accessed system Y at 14:32', they want to know which transaction was processed, which customer data was read, which business rule was applied, which compensating control was active. This requires that events from different systems, application, database, KMS, network, share a common correlation identifier. The design of the event schema (as discussed in Chapter 8 on event-driven) and the design of auditability are, in practice, the same problem seen from different angles.

When the regulator asks 'who moved that money and with what authorization?', the answer must exist in structured form, retrievable in minutes, and verifiable as authentic. Not as a post-hoc reconstruction from fragmented logs, as a query to a record that was designed to exist from the beginning.

How to embed evidence from diagram zero

1. Map regulatory constraints as explicit non-functional requirements: Before the first architecture diagram, list every relevant regulatory obligation (AML, LGPD, CMN Resolution 4.658, PCI-DSS) and convert it into a concrete requirement: retention period, data scope, access control, required evidence. These requirements enter the backlog with the same priority as functional requirements.
2. Define the audit event schema alongside the domain schema: Each domain event (transaction initiated, credit approved, customer data updated) must have a corresponding audit event with mandatory fields: actor identity, timestamp with timezone, affected resource, executed action, result, correlation identifier. This schema is defined before implementation, not after.
3. Configure multi-region CloudTrail with integrity validation and immutable destination: CloudTrail enabled in all regions, with log file integrity validation active, sending to an S3 bucket in a separate log archive account with Object Lock in COMPLIANCE mode and a bucket policy that denies deletion even to administrators. The KMS key encrypting the logs must have a policy that prohibits key deletion before the retention period.
4. Implement key hierarchy with envelope encryption and usage auditing: Sensitive data encrypted with DEK generated by KMS; DEK encrypted by CMK; every KMS call audited in CloudTrail. For

high-criticality operations (transaction signing, card cryptograms), use CloudHSM with exclusive bank custody. Automatic key rotation enabled with period defined by security policy.

5. Integrate Security Hub, GuardDuty, and Config into a detection and response cycle: Security Hub findings, GuardDuty alerts, and Config deviations must feed a response process with SLA defined by severity. Detection alone is not enough, each finding needs an owner, deadline, and evidence of remediation. This closes the loop: prevention, detection, evidence of response.

Frequently asked questions about security as evidence

KMS or CloudHSM? How to decide?

KMS with CMKs is sufficient for the vast majority of cases: envelope encryption of data at rest, token signing, secrets protection. CloudHSM is necessary when the bank requires exclusive custody of key material (AWS must have no access under any circumstance), when the operation requires FIPS 140-2 Level 3 certification with a dedicated HSM, or when the card scheme or BACEN requires a proprietary HSM for cryptogram generation. The operational cost of CloudHSM is significantly higher, use it where regulation or the threat model justifies it.

How long do I need to retain CloudTrail logs?

It depends on the jurisdiction and data type. For AML evidence in Brazil, BACEN Circular 3.978/2020 requires record retention for a minimum of five years. For personal data under LGPD, the period varies according to the legal basis for processing. For card operations under PCI-DSS, the minimum is one year with three months online. The practical recommendation is to define retention classes by event type, operational (1 year), regulatory (5 years), legal (applicable statute of limitations), and configure S3 lifecycle policies for automatic transition between storage classes, maintaining immutability throughout.

How to prove that a log has not been tampered with?

CloudTrail with log file integrity validation generates a digest file every hour, signed with an RSA key managed by AWS, that chains all log files from the period. To verify integrity, you run `aws cloudtrail validate-logs` and the service reconstructs the hash chain. If any file was modified or deleted, validation fails. Combined with S3 Object Lock (which prevents physical modification) and a separate account destination (which prevents access by production account administrators), you have three independent layers of integrity assurance.

Security as evidence: the minimum standard for serious banking architecture

The distinction between security as opinion and security as evidence is, at its core, a distinction of architectural maturity. Banks that treat compliance as a design constraint from the first diagram, embedding auditability, immutability, and correlatability into every layer, build systems that survive audits, incidents, and the passage of time. Banks that treat security as a final checklist invariably discover that the cost of rewriting under pressure is orders of magnitude greater than having it embedded from the start. The architect who can ride up to the penthouse and translate 'prove who approved' into concrete IAM, KMS, CloudTrail, and Config decisions, and descend to the engine room and implement them in a form the regulator can verify, is the architect the bank needs.

13 - Architecture only truly exists in production

An architecture you can't observe, operate or recover is a hypothesis. In financial systems, production is where decisions become consequences, and SLOs, resilience and FinOps are part of the design.

An architecture exists on paper, in diagrams, in committee presentations, but it only becomes real the moment a customer tries to make a Pix transfer at 11:47 PM on a Friday and the system decides, in milliseconds, whether that transaction completes or fails. Production is not the final destination of design: it is where every architectural decision becomes a measurable, regulatory, and financial consequence. In this chapter I close Part IV with the thesis that guides everything that came before: an architecture that cannot be observed, operated, and recovered is a hypothesis, and hypotheses don't pay anyone's salary.

The SLO the customer experiences is not the service SLO

There is a reasoning error I encounter repeatedly in architecture reviews: teams display dashboards showing 99.9% availability per service and interpret that as evidence of system health. It is not. A successful Pix transfer crosses, on average, between eight and fifteen internal components: API gateway, authentication service, limit validator, fraud engine, SPI integrator, ledger, notifier, reconciler. If each of those components has 99.9% independent availability and they are statistically independent (already an optimistic assumption), the composed availability of the journey is, at best, something close to $99.9\%^n$, and with ten components that drops to approximately 99.0%. That number still sounds high until you calculate that 1% failure in a bank with one million daily transactions means ten thousand impacted customers per day.

The change I propose is structural: define journey SLOs, not service SLOs. The relevant SLO is: "99.5% of initiated Pix payments complete successfully in under 8 seconds, measured from the client side, in 5-minute windows." That SLO crosses domains, includes external dependencies (card network, partner, BACEN) and is exactly what the regulator and the customer measure, even if they never use that vocabulary.

This profoundly changes how you instrument the system. Instead of per-service alerts, you need distributed traces per journey, with spans named by business step, not by microservice name. On AWS, this means X-Ray or OpenTelemetry with domain attributes propagated via context propagation, correlated in CloudWatch or a dedicated observability platform. The SLO error budget must be calculated over the journey, not the component. When the Pix journey error budget starts burning faster than expected on a Tuesday morning, the architect needs to know that before the customer complains, and before the Banco Central opens a ticket.

Questions the architect asks before the incident

- Partner, BaaS, or card network goes down: does my system degrade gracefully or stop with it? Which journey continues in degraded mode and which refuses with a clear message?
- An event arrives duplicated or out of order: does the ledger maintain consistency? Is the event processor idempotent by contract, not by hope?
- Spike on payroll day: what queues, what rejects with explicit backpressure, and what can never be dropped? Are there queues by journey criticality?
- Deploy fails in production: what is the recovery time? Is rollback automatic or does it depend on someone waking up? Is the blast radius contained to one domain?
- The AI model evaluating credit or fraud degrades or becomes unavailable: is there a deterministic fallback that is documented, tested, and actionable without manual approval?

- Has the runbook for the most likely incident been executed in a simulation in the last 90 days?
Has someone from the current team run it, not just the original author?

Resilience is what remains after the diagram meets reality

When I ride up to the executive floor to discuss business continuity, I use risk language: probability of impact, exposure window, cost of downtime per hour. When I descend to the engine room to implement that same conversation, the language shifts to circuit breakers, dead-letter queues, retry with exponential backoff and jitter, bulkheads per domain, and scheduled chaos engineering. The architect's skill lies in keeping these two conversations connected, and in ensuring that the technical decision to "use SQS with DLQ and a 30-second visibility timeout" is the direct implementation of the business decision that "no payment transaction can be silently dropped."

Resilience in financial systems has three dimensions that must be explicitly designed. The first is blast radius containment: when a component fails, the damage does not propagate to adjacent domains. On AWS, this translates to separate accounts per domain (or at least by criticality), VPCs with controlled peering, and SQS queues or SNS topics as isolation boundaries between services. The second dimension is graceful degradation with explicit contract: each journey needs a documented degraded mode, not improvised during the incident. A credit proposal can be queued for asynchronous processing if the decision engine is slow; a Pix cannot be silently delayed without notification to the customer and monitoring system. The third dimension is verifiable recovery: having a recovery plan is not enough; it must be executed in simulation regularly, with recovery time metrics recorded and compared to the contracted RTO.

AWS Fault Injection Simulator (FIS) is the tool I use to make chaos engineering part of the development cycle, not a special event. Injecting latency into the authentication service on a Wednesday afternoon, in a staging environment with mirrored traffic, reveals dependencies that no diagram captures. When the experiment result contradicts the diagram, the diagram is wrong, and it is better to discover that on Wednesday than on Friday at 11:47 PM.

The most beautiful architecture is the one the team operates at 3 AM: After sixteen years, I learned to evaluate an architecture with a simple question: if the on-call engineer is woken up at 3 AM with an alert, with no context, sleepy and under pressure, can they diagnose and mitigate the problem in under fifteen minutes using only the available runbook and dashboards? If the answer is no, if diagnosis requires tacit knowledge from whoever designed the system, or if the runbook assumes the person is rested and has time to think, then the architecture is not production-ready, regardless of how many nines of availability the design promises. Resilience is not a diagram. It is a runbook that works under sleep deprivation and pressure, executed by someone who joined the team three months ago. That is the real test.

Observability as a decision instrument, not a pretty graph

Observability in financial systems has a requirement that goes beyond what most monitoring platforms address by default: it must be auditable and correlatable with business events. When the Banco Central or an external auditor asks why a specific transaction was processed with 4.2 seconds of latency in a particular window, the answer cannot be "we don't have sufficient log granularity." Every relevant business event, journey start, credit decision, settlement, reversal, needs a correlated trace ID, a millisecond-precision timestamp, and a business context (transaction ID, domain, journey type) that enables post-mortem reconstruction.

On AWS, the combination I use as a starting point is: CloudWatch Logs Insights for ad-hoc correlation, X-Ray or OpenTelemetry with ADOT for distributed traces, CloudWatch Metrics with

business dimensions for journey SLOs, and EventBridge as an audit bus for domain events. For high-volume banks, ingestion into Amazon OpenSearch or a dedicated platform like Datadog or Grafana Cloud is frequently justified by reduced operational cost and real-time correlation capability.

The point I emphasize most with teams: observability must be instrumented in the design, not added afterward. When a service is designed without considering what business metrics it needs to emit, the result is a system that monitors itself, endpoint latency, CPU usage, HTTP errors, but does not monitor what the business needs to know. The question I ask in every service design review is: "if this service starts producing incorrect results without technically failing, how will we know?" That question, when answered honestly, defines the required instrumentation, and frequently reveals that the service needs to emit business events with explicit semantics, not just technical logs.

FinOps: cost per transaction as a product metric

There is a conversation that rarely happens in bank architecture reviews and should be mandatory: what is the unit cost of each journey the system executes? Not the total infrastructure cost, that number matters to the CFO but does not guide design decisions. The number that matters to the architect is: how much does it cost to process a Pix, approve a credit proposal, or generate a statement? When that number is visible in real time, it becomes a product metric, and starts guiding design decisions in the same way that latency and availability do.

The table below presents the FinOps framework I apply in banking contexts on AWS, organizing cost by business domain, with transaction granularity and team ownership attribution. The central principle is that each domain owns its cost: there is no "infrastructure cost" as an opaque category managed centrally. When the payments team sees that the cost per Pix increased 15% after a deploy, that information is as relevant as a latency increase.

Pay-per-use and right-sizing are the two main levers. On AWS, this means preferring Lambda and Fargate for workloads with unpredictable spikes (like boleto processing on due dates), and EC2 with Savings Plans or Reserved Instances for workloads with stable and predictable baselines (like the positions ledger). The mistake I see frequently is the inverse: reserved instances for workloads that vary 10x between peak and valley, and Lambda for batch processing that would be cheaper on EC2 Spot. The compute decision is not just technical, it has direct impact on the unit cost of the journey, and therefore on the business model.

FinOps in banking is not about spending less. It is about spending with enough visibility to know whether the business model is sustainable, and to identify, before the product scales, whether the unit cost is on the right trajectory.

How to structure journey SLOs in practice

1. Map critical journeys, not services: List the five to ten journeys that, if degraded, cause immediate regulatory, financial, or reputational impact. Pix, TED, credit proposal, authentication, and statements are typical candidates. Each journey will have its own SLO, availability, latency, and correctness.
2. Instrument the end-to-end trace with business context: Propagate trace ID and business attributes (journey type, domain, criticality) via HTTP headers and SQS/EventBridge message metadata. The root span should be named by the business journey, not the technical service.
3. Calculate error budget per journey and make it visible: In CloudWatch or the chosen observability platform, create a dashboard per journey with current error budget, burn rate, and projection to the end of the window. This dashboard is the decision instrument, not the service dashboard.

4. Define burn rate alerts, not absolute threshold alerts: An alert that fires when latency exceeds 500ms is reactive. An alert that fires when the Pix journey error budget is burning at 14x the normal rate is predictive, it warns that, if nothing changes, the SLO will be violated in less than an hour.

5. Validate the runbook for each critical journey in quarterly simulation: Schedule quarterly game days with fault injection via AWS FIS. The success criterion is not "the system survived", it is "the on-call engineer diagnosed and mitigated within the RTO using only the runbook, without help from whoever designed the system".

FinOps in a bank: cost as a product metric

Criterion Naive approach Mature FinOps approach

--- --- ---

Cost view Total bill at month end Cost per transaction and per journey, in real time

Capacity Provision for the peak and forget Pay-per-use + continuous right-sizing

Runtime decision Everything on one always-on cluster Serverless for irregular, containers for constant

Cost owner Infra, at the end of the line Each domain owns its cost, observable

Frequently asked questions about operating financial architectures

Is a journey SLO different from a contractual SLA with the customer?

Yes, and the distinction matters. The SLA is the external commitment, often conservative and with contractual penalties. The SLO is the internal objective, more rigorous, that serves as an early warning signal before the SLA is violated. The SLO should be more demanding than the SLA by a margin that allows reaction time, typically, if the SLA is 99.5%, the internal SLO is 99.7% or 99.8%.

How to attribute cost per journey when infrastructure is shared?

Use cost tags (Cost Allocation Tags in AWS) with domain and journey-type dimensions propagated to the resource level. For genuinely shared infrastructure (like a multi-tenant EKS cluster), use proportional utilization metrics to distribute cost, it is not perfect, but it is precise enough to guide design decisions and identify journeys with out-of-control unit costs.

Is chaos engineering viable in banks with strict regulation?

Yes, with appropriate scope and governance. Start in staging with mirrored (shadow) traffic, document each experiment as a resilience test case, and maintain execution history as evidence of operational due diligence. Regulators like the Banco Central value evidence that the bank actively tests its recovery capability, the opposite of unregulated chaos engineering is a bank that discovers its weaknesses during a real incident.

Production is where architecture proves its value

Throughout this chapter, the elevator went up and down many times: from the regulatory risk of Pix unavailability to the SQS visibility timeout configuration; from the CFO conversation about transaction unit cost to the cost allocation tag on the AWS resource; from the RTO definition in the continuity contract to the quarterly game day with AWS FIS. Each descent to the engine room was motivated by an executive floor decision, and each ascent carried technical evidence into a business conversation. That is the architect's work in production: not just designing systems that work, but ensuring they work in an observable, recoverable, and economically sustainable way, and that the team operating them at 3 AM has the tools and runbooks to prove it.

Riding Up: Decision and Transformation

Back to the penthouse: how to sell options, record decisions, build mechanisms that outlive the meeting, and lead transformation without turning everything into PowerPoint.

14 - Selling options and recording decisions

Architecture creates options, and options have value under uncertainty, that's Hohpe's most underrated thesis. Riding up is selling that value; riding down is recording the decision in an ADR and turning it into consequence.

Every architecture decision is, at its core, a bet on the future, and in a bank, where the regulator rewrites the rules, the market launches new competitors, and technology reinvents itself every cycle, betting without a hedge is professional recklessness. This chapter opens Part V with the thesis I consider the most underestimated in Gregor Hohpe's work: architecture does not deliver software, it delivers options, and options carry real, measurable financial value, especially under high uncertainty. Riding the elevator up means knowing how to sell that value to the penthouse; riding it down means turning the decision into an ADR and into concrete consequence.

Architecture as an options portfolio

Anyone who has worked with derivatives knows that a call option gives the holder the right, not the obligation, to buy an asset at a fixed price in the future. You pay a premium today for that right. If the asset rises above the strike, the option becomes profit; if it does not, you lose only the premium, not the full capital. The logic is exactly the same when we decide to decouple two domains via events instead of a direct synchronous call.

The premium is real: more infrastructure, more event contracts to maintain, more observability surface area. Nobody should pretend that cost does not exist. But what you buy with that premium is the right to replace the implementation of one domain without rewriting the others: the right to scale transaction processing independently of the notification engine, the right to migrate the payments core to a new provider without interrupting the anti-fraud flow.

In a low-uncertainty environment, say, an internal payroll system at a mature company with requirements stable for ten years, that option is worth little. The premium probably does not justify itself. Optimize cost, simplify, couple. But in a Brazilian bank operating under BACEN, LGPD, Pix, Open Finance, fintech pressure, and unpredictable regulatory cycles, uncertainty is structural. Here, paying for optionality is not fool's gold, it is risk management applied to engineering.

The problem is that this argument rarely appears so formulated in technical discussions. It appears as personal preference: 'I prefer events', 'microservices are more modern'. When the debate reaches the penthouse as technical taste, it loses. When it arrives as portfolio management of options under uncertainty, it finds natural interlocutors, because that is the native language of those who decide capital allocation.

The central thesis of this chapter: Architecture sells options. Under low uncertainty, options are worth little, optimize cost and simplify. In banking (high regulatory, competitive, and technological uncertainty), options are worth a great deal, and knowing when to pay for optionality and when not to is half the architect's job. The other half is recording that decision in a way that survives staff turnover and the pressure of the next sprint.

Riding the elevator up with the right language

Hohpe describes the architect as someone who moves between the penthouse, where strategy, risk, and capital live, and the engine room, where code, latency, and pipelines live. Most architects I know are technically solid in the engine room but lose context on the way up. They arrive at the penthouse talking about throughput, eventual consistency, and idempotency keys, and leave without budget, without priority, and without sponsorship.

The shift happens when the architect learns to translate technical decisions into business consequences: and, more specifically, when they learn to frame those consequences in terms of risk and optionality. Consider two framings for the same decision to decouple the FX domain from the compliance domain via EventBridge:

Technical framing: 'We will use asynchronous events to reduce temporal coupling between services.'

Options framing: 'We are paying a premium estimated at X weeks of additional effort to buy the right to replace the compliance engine, for example, when onboarding a new regulatory partner, without having to rewrite the FX flow. Given that we have at least two open regulatory processes that may require that swap in the next eighteen months, the premium appears justified.'

The second framing is not longer by accident. It contains the premium (cost), the right purchased (what the option enables), the trigger (when the option would be exercised), and the qualitative probability of exercise. It is a language the CFO and CRO recognize immediately.

This does not mean the architect must learn to build Black-Scholes spreadsheets. It means they need to develop the habit of asking, for each decoupling decision: what is the real premium? what right am I buying? in what scenarios would that right be exercised? what is my qualitative estimate that those scenarios occur? If they cannot answer those four questions, the decision is not yet mature enough for the penthouse.

When the option is worth little, and when it is worth a great deal: In systems with stable requirements and low regulatory risk, paying the decoupling premium is waste, simplify and couple. In Brazilian banks, the combination of BACEN with broad normative power, Open Finance cycles still evolving, fintech competitive pressure, and technological uncertainty (generative AI redefining products in real time) creates one of the highest uncertainty densities I have encountered in any sector. Here, architectural options are genuine hedges, not fool's gold.

The ADR as mechanism, not ceremony

After riding the elevator up and selling the option, the architect needs to ride it back down and record it. This is where the Architecture Decision Record (ADR) enters, not as bureaucracy, not as audit documentation, but as a mechanism that prevents the system from losing its memory.

A well-written ADR has five mandatory elements. The context describes the state of the world at the moment of the decision: what regulatory constraints were in force, what the expected load was, what existing dependencies existed. Without context, the decision looks arbitrary to those who arrive later. The options considered list the real alternatives that were evaluated, and here is the most neglected element: what was rejected and why. The option discarded today is the temptation of tomorrow. When a new engineer arrives and proposes exactly what was rejected eighteen months ago, the ADR is what prevents redoing the debate from scratch, or worse, repeating the mistake.

The decision itself must be stated unambiguously: not 'we will consider Aurora', but 'we will use Aurora PostgreSQL as the primary ledger for the banking core'. And the consequences: the most

important field and the most frequently omitted, must list what changes in the system as a direct result of that decision: which capabilities are enabled, which are inhibited, which technical debts are consciously accepted, which future revisions are anticipated.

The fifth element, which I add from personal experience in financial systems, is the review trigger: under what conditions should this decision be reopened? A regulatory change? Volume growth above a certain threshold? The availability of a new managed service? Without that trigger, the ADR becomes archaeology, found by accident, never updated, irrelevant. With it, the ADR becomes part of the living governance process of the platform.

As the decision matrix below shows, a real ADR for the ledger choice in the banking core: Aurora versus DynamoDB, is not a simple document. It carries trade-offs of consistency, cost model, operational capability, and future optionality that only make sense when recorded together, at the moment they were weighed.

How to structure an ADR in financial systems

1. Context: Describe the state of the world at the time of the decision: regulatory constraints (BACEN, LGPD), expected volume, existing dependencies, deadline pressures. Without context, the decision looks arbitrary to those who arrive later.
2. Options considered (including rejected ones): List all real alternatives evaluated. Explicitly record what was rejected and why, the option discarded today is the temptation of tomorrow, and the ADR is what prevents redoing the debate from scratch.
3. Decision: State the decision unambiguously and in the present tense: not 'we will consider X', but 'we will use X for Y'. Ambiguity here generates divergent reinterpretation over time.
4. Consequences: List what changes in the system: capabilities enabled, capabilities inhibited, consciously accepted technical debts, anticipated future revisions. This is the most important field and the most frequently omitted.
5. Review trigger: Explicitly define under what conditions this decision should be reopened: a specific regulatory change, volume growth above a threshold, availability of a new managed service. Without a trigger, the ADR becomes archaeology.

The ADR as a living governance artifact

There is an objection I hear frequently: 'ADRs go stale and nobody reads them'. That is true, when treated as documentation. When treated as a governance mechanism, the behavior changes.

The difference lies in three operational practices. First: the ADR lives in the code repository, not in a separate wiki. It is versioned alongside the system it governs. When code changes in a way that contradicts the ADR, that is visible, and should be a signal that either the ADR needs revision or the code change needs explicit justification. Second: the ADR has an explicit status: proposed, accepted, superseded, obsolete. A superseded ADR is not deleted; it is marked as superseded and points to the successor ADR, preserving the chain of reasoning. Third: the review trigger is monitored. If the trigger is 'transaction volume above 50,000 TPS' (illustrative estimate), that threshold should be on a dashboard, when reached, the ADR automatically enters the architecture review agenda.

In banks, this discipline has an additional dimension: auditability. BACEN and external auditors do not ask only what the system does, they ask why it was built that way, what alternatives were considered, and what risks were consciously accepted. A well-maintained ADR portfolio is the most honest and most defensible answer to that question. I have seen teams spend weeks reconstructing

the justification for an architecture decision for an audit, time that would have been zero if the ADR had existed.

The decision matrix I present below: Aurora versus DynamoDB for the banking core ledger, is a real example of the kind of trade-off that deserves this level of recording. It is not a trivial decision: it carries implications for transactional consistency, cost model at scale, team operational capability, and, crucially, future optionality for migration or expansion.

[DECISION MATRIX] ADR example: core ledger: Aurora vs. DynamoDB

The matrix below materializes the principles discussed in this chapter in a concrete case: the database choice for the primary ledger of the banking core. This is precisely the category of decision that cannot be made by technical preference, it must be made as options portfolio management, with explicit premium, right, and review trigger.

Observe, when reading the matrix, how each option carries not only technical characteristics but business consequences: what each choice enables and what it inhibits over a two-to-five-year horizon. Also observe what was rejected and why: the temptation to use DynamoDB for horizontal scalability is real, but the consequences for transactional consistency and ledger auditability are consequences the penthouse needs to understand before approving.

This is the elevator in operation: the technical decision about a database carries implications for operational risk, regulatory compliance cost, and strategic optionality that only make sense when presented together, in the same artifact, for audiences on both floors.

Example ADR: ledger in the core: Aurora vs. DynamoDB

Aurora PostgreSQL (relational)

Pros

- Strong consistency and native ACID transactions
- Mature SQL for reconciliation and audit
- A natural model for double entry

Cons

- Write scale needs sharding/Limitless
- Cost grows with sustained volume

Verdict: The default for the central ledger, where strong consistency and auditability are non-negotiable.

DynamoDB (key-value)

Pros

- Horizontal scale and predictable latency
- Minimal operations, pay-per-use
- Great for low-latency balance projections

Cons

- Limited transactions; modeling double entry is laborious
- Reconciliation and analytical queries are harder

Verdict: Strong for high-scale projections and reads, not for the central accounting record.

Make impact, not PowerPoint: Hohpe: Hohpe is direct: the architect's goal is to make impact, not produce slides. Measure yourself by what changed in the system, a decision recorded in an ADR that was actually implemented, a debate that did not need to be redone, an option exercised at the right moment because it was documented. A beautiful deck that changes nothing is noise. A two-page ADR that aligns ten engineers and a CTO for eighteen months is real leverage.

Closing the loop: from option to consequence

The architect's complete cycle in this chapter has three movements. In the first, they identify the decision as an option, with premium, right, and trigger. In the second, they ride the elevator up and sell that option to the penthouse in the language of risk and capital that the penthouse understands. In the third, they ride it back down and record the decision in an ADR with context, rejected alternatives, consequences, and a review trigger.

What closes the loop is follow-through. An option sold and not monitored is a promise unkept. The architect must ensure that review triggers are instrumented, in dashboards, in alerts, in governance processes, and that when a trigger fires, the ADR is revisited with the same seriousness with which it was created.

In financial systems, this loop closure has an additional dimension I cannot omit: reversibility. Some architecture decisions are easily reversible, swapping a serialization library, changing a configuration parameter. Others are practically irreversible within the relevant horizon, choosing the ledger data model, defining the core event topology. The ADR must explicitly record the estimated reversibility of the decision, because this directly affects how much premium is worth paying for the option of not committing to it.

When the architect masters this cycle, identify option, sell to penthouse, record in ADR, monitor triggers, revise when fired, they stop being the technical person the business tolerates and start being the partner the business seeks. That is the promise of the elevator: not that you know everything about every floor, but that you are capable of translating consequence between them with precision and with accountability.

Key points of the chapter

- Architecture delivers options, not just software, and options have real financial value, especially under high uncertainty like that of Brazilian banks.
- Every decoupling decision has a premium (cost today) and a right (future capability); articulating both is what makes the technical argument comprehensible in the penthouse.
- The ADR is not documentation, it is the mechanism that prevents the system from losing its memory and that avoids redoing debates or reversing decisions without understanding their consequences.
- The most important field in the ADR is consequences, and the most neglected. The second most important is what was rejected and why.
- The review trigger transforms the ADR from archaeology into living governance, it must be instrumented and monitored, not just written.
- Measure yourself by impact on the system, not by slides produced. An ADR that aligns a team for eighteen months is more valuable than any executive presentation that changes nothing.

The architect who sells options and records decisions

The maturity of a financial systems architect is not measured by the complexity of the solutions they propose, but by the quality of the options they preserve and the clarity with which they record the

ones they discard. Riding the elevator up with the language of options and riding it back down with well-written ADRs is what separates the architect who leaves a legacy from the architect who leaves a debt.

15 - Mechanisms and leading change

A decision without a mechanism evaporates. Transforming a bank isn't drawing the end state, it's building the mechanisms that make the organization decide better, sustainably, floor after floor.

Every architectural transformation in a bank starts with an enthusiastic meeting and ends, most of the time, exactly where it started, because enthusiasm is not a mechanism. The senior architect who understands this stops asking 'what is the right decision?' and starts asking 'what is the mechanism that makes the right decision happen on its own, after I leave the room?' That shift in question is the difference between writing documents and changing organizations.

Decision without mechanism is intention disguised as architecture

There is a recurring illusion in bank transformation programs: the idea that deciding is enough. Leadership approves the roadmap, the architect presents the reference diagram, the committee signs the minutes, and everyone leaves the room convinced that something changed. Nothing changed. What changed was the recording of an intention.

Gregor Hohpe has a phrase I use as a calibrator every time I evaluate a transformation program: slow chaos is not order. Slow process applied over chaos does not produce order, it produces slow chaos, which is even harder to diagnose because it looks organized. When a bank decides that 'every domain will publish events with versioned schema and guaranteed idempotency' without creating any adoption mechanism, what happens over the next six months is exactly that: each squad interprets the decision differently, some publish events without a schema, others create incompatible schemas, and the event catalog becomes an outdated document no one trusts. The decision existed. Order never arrived.

A mechanism is what keeps working after the meeting ended and the enthusiasm faded. It is the service template that already comes with observability built in, so the right path is also the easiest path. It is the policy-as-code in the CI pipeline that blocks the deploy of an event without a registered schema, so compliance does not depend on individual discipline, it depends on physics. It is the DLQ dashboard per domain with an identified owner, so the problem of a consumer that is not processing messages becomes visible before it becomes an incident. It is the quarterly incident review that feeds back into the template, so collective learning accumulates instead of evaporating.

The distinction I make with teams I work with is direct: a document convinces in the meeting; a mechanism convinces every day. And in banking, where leadership turnover is high and priority cycles change every quarter, only what has been institutionalized into a mechanism survives.

The transformation that lasted: In every modernization program I have followed closely, the variable that best predicts whether the transformation will last is not the budget, not the executive sponsor, and not the quality of the roadmap. It is whether the program created mechanisms that survive leadership changes. I have seen excellent initiatives die in six months because they depended on the enthusiasm of a VP who left. I have seen modest initiatives last for years because someone took care to embed the decisions into templates, pipelines, and automated policies. The document convinces those who are in the room. The mechanism convinces those

who were never there.

From decision to mechanism: the complete cycle

1. Architectural decision: Every domain publishes events with versioned schema, explicit idempotency contract, and mandatory correlation key. This decision is recorded as an ADR (Architecture Decision Record) with context, consequences, and review date, not merely as a presentation slide.
2. Adoption mechanism: A service template in the internal repository already includes the Outbox pattern implemented, schema registration in AWS Glue Schema Registry configured, and an idempotent consumer with a DynamoDB deduplication table ready to use. The team does not need to know the theory, they need to clone the template.
3. Governance mechanism: A policy-as-code in the CI pipeline, implemented with AWS Config Rules and GitHub Actions checks, blocks the deploy of any event whose schema is not registered in the central catalog. The block is automatic: there is no approval committee, no manual exception without an auditable justification record.
4. Visibility mechanism: An event catalog, fed automatically by the Schema Registry, exposes producer, consumers, active version, and version history. A dashboard in Amazon CloudWatch shows, per domain, the rate of messages in DLQ with the name of the responsible team. A visible problem has an owner. An invisible problem becomes silent debt.
5. Learning mechanism: A quarterly incident review, with a structured agenda and participation from domain leads, analyzes the period's incidents and identifies patterns that should feed back into the template. If three incidents in the quarter involved consumers that did not handle poison-pill messages, the template is updated with explicit dead-letter handling before the next cycle. Learning accumulates in the tool, not in one person's head.

The elevator between the penthouse and the engine room: two floors that need to advance together

The tension that the banking architect faces every day has a specific geometry: the penthouse wants speed, innovation, and time-to-market; the engine room carries critical systems that process settlements, calculate reserves, and report to BACEN, systems that cannot stop, cannot lose data, and cannot introduce accounting inconsistency. When the architect does not create mechanisms that allow both floors to advance together, what happens is predictable: either the penthouse runs over the engine room with changes that generate incidents, or the engine room paralyzes the penthouse with approval processes that take weeks.

The way out is not to choose a floor. The way out is to create mechanisms that reduce the friction of the right path on both floors simultaneously. In the penthouse, this means the architect translates technical risk into business language, not 'we have excessive synchronous coupling', but 'each new integration increases the probability of digital channel unavailability by X percentage points' (estimate, not measurement). In the engine room, this means the architect creates abstractions that allow the legacy team to evolve without rewriting everything, strangler fig over the banking core, events as a decoupling layer, versioned APIs that isolate the consumer contract from the internal implementation.

The mechanism that connects both floors is what I call the trust trail: a set of automated practices that allows the innovation team to move fast because the platform team has guaranteed that the floor is solid. Automated contract tests that validate that the new feature does not break the legacy

consumer. Feature flags with automatic rollback based on business metrics. Staging environments with synthetic data that respects the LGPD and is at the same time representative enough to validate financial behavior. These mechanisms do not eliminate the tension between floors, the tension is legitimate and productive. They eliminate the unnecessary friction that turns tension into paralysis.

I laugh internally when I hear 'we need a more agile architecture committee'. A more agile committee is more meetings. What the bank needs is less centralized decision-making and more distributed mechanism, guardrails that allow autonomy within safe limits, instead of centralized approval that creates bottlenecks.

Ownership as mechanism: whoever owns the problem owns the solution

One of the most underestimated, and cheapest to implement, mechanisms is the explicit and public assignment of ownership. Not ownership in the bureaucratic sense of 'formal responsible party', but in the operational sense of 'who wakes up at 2am when this breaks and has the autonomy to fix it'. The difference is enormous.

In distributed financial systems, the problem of diffuse ownership is especially serious. An event that traverses four domains before updating the customer's balance has four teams that can say 'not my problem' when the message disappears in the DLQ. Without an explicit ownership mechanism, the incident becomes a finger-pointing meeting. With an explicit ownership mechanism, the event catalog with an identified owner, the DLQ dashboard that sends an alert directly to the responsible team's channel, the runbook that defines the escalation protocol, the problem has an address before it happens.

In the AWS context, this translates into concrete practices: mandatory tags on all resources with owner, domain, and criticality, validated by AWS Config; CloudWatch alarms configured in the service template with automatic routing to the team's Slack or PagerDuty channel; runbooks stored in Systems Manager and linked directly in the alarm, when the alert fires, the link to the runbook is already in the notification body. The on-call person does not need to search for what to do: the mechanism delivers the context together with the problem.

Ownership without autonomy, however, is punishment disguised as responsibility. The mechanism only works if the team that owns the event also has the power to change the schema, adjust the consumer, and roll back without needing approval from three committees. This has a direct implication for how the architect draws domain boundaries: domain boundaries must coincide with operational autonomy boundaries. When they do not coincide, ownership is nominal and the mechanism fails at the first real crisis.

Mechanisms that make transformation last

- Service template with observability, idempotency, and schema built in: the right path is the easiest path, not the most disciplined one.
- Policy-as-code in CI/CD: compliance depends on physics, not manual review. Automatic block with no unaudited exception.
- Event catalog with visible owner, version, and consumers: a problem without an address has no solution, only a meeting.
- DLQ dashboard per domain with alert routed to the responsible team: operational visibility is not optional in a financial system.

- Quarterly incident review that feeds back into the template: collective learning accumulated in the tool, not in one person's head.
- ADRs with a review date: an architectural decision without a review deadline is dogma, not architecture.

Leading without PowerPoint: the architect as creator of conditions

There is a version of the senior architect who spends most of their time producing presentations to convince stakeholders. That version is necessary in the right doses, the penthouse needs context and narrative. But when the architect spends more time convincing than building mechanisms, they become a persuasion bottleneck: nothing advances without their presence, and when they leave the company, the transformation stops.

The version I advocate is different: the senior architect leads by creating conditions for the right decisions to happen without them. This means investing time in three activities that rarely appear in the job description but have the highest long-term return. First, building and maintaining templates: not delegating the template construction to a junior team and signing off, but getting hands dirty in the Outbox pattern implementation, understanding where DynamoDB Streams creates unexpected complexity, discovering that Glue's Schema Registry has specific behavior with Avro schemas containing optional fields. Second, instrumenting visibility: ensuring that dashboards exist, that alarms are calibrated for the financial context (a latency alarm that fires for every operation above 200ms in overnight batch is noise; the same alarm in a PIX transaction is critical), and that the runbook is linked where the on-call person will look. Third, facilitating collective learning: the quarterly incident review is not a post-mortem meeting; it is the mechanism by which the tacit knowledge of the most experienced team becomes explicit knowledge in the template.

This approach has a cost I need to name: it is slower in the short term and less visible to those who measure output by number of presentations delivered. The architect who spends three weeks refining a service template has no new slide to show at the steering meeting. But six months later, when twelve teams have onboarded using the template without a schema incident, the value is there, distributed, silent, and lasting. That is the signature of architectural work that truly changes organizations: it is not loud, but it is permanent.

The elevator, in this context, has a specific function: the architect goes up to the penthouse to understand which business problem the mechanism needs to solve, what regulatory risk the policy-as-code is mitigating, what operational cost the DLQ dashboard is avoiding, and goes down to the engine room to ensure that the mechanism is implemented with the technical precision that the financial context demands. The round trip is not ceremony; it is what guarantees that the mechanism solves the right problem in the right way.

Frequently asked questions about mechanisms and transformation

How do you convince leadership to invest in mechanisms when what they want to see is features delivered?

Do not sell the mechanism, sell what the mechanism prevents. 'We are going to create a service template' convinces no one in the penthouse. 'Each new domain onboarding without a template costs an estimated X weeks of rework and was the root cause of two of the three incidents last quarter' (estimate based on real data from your context) convinces. Translate mechanism into avoided risk and recovered speed.

What about when the team resists the template because 'every domain is

different'?

Resistance to a template is usually resistance to loss of autonomy, not a legitimate technical objection. The answer is template design: it should be opinionated in the parts that matter for security and compliance (schema registry, idempotency, observability) and extensible in the parts that vary by domain. If the template has no clear extension points, the problem is with the template, not the team. Rewrite the template before forcing adoption.

How do you keep mechanisms updated without becoming a maintenance bottleneck?

The learning mechanism, the quarterly review that feeds back into the template, needs a product owner, not an individual technical owner. The architect facilitates the process and makes the design decisions; the team contributes the real use cases. Version the template as you version an API: breaking changes have an explicit migration cycle, additive changes are automatically available. And accept that a slightly outdated template is still better than no template.

The architect who changes organizations

The difference between the architect who writes documents and the architect who changes organizations fits in one word: mechanism. Not mechanism as bureaucracy, more process, more committee, more approval. Mechanism as physics: that which makes the right path the path of least resistance, which works after the enthusiasm has faded, which survives leadership changes, which accumulates collective learning instead of letting it evaporate. In banking, where the cost of inconsistency is regulatory and the cost of unavailability is reputational, building these mechanisms is not support work for architecture, it is the central work of the senior architect.

16 - The architect as a translator of consequence

Closing the elevator: architecture is conversation, decision and consequence. In a bank, that consequence shows up in trust, risk, availability, audit, cost, experience and the speed of change.

We have reached the top floor. Throughout this book we traversed the entire building, from the penthouse where executives make risk decisions to the engine room where idempotency and double-entry bookkeeping determine whether the money balances, and the thread stitching every chapter was always the same: the architect exists to move context between those floors without losing it along the way. This chapter closes the elevator and delivers what the book promised: an operational synthesis of how banking architecture on AWS connects strategy to consequence.

My view after sixteen years in this building: The greatest failure I see in senior architects is not technical. It is the inability to translate consequence: to turn an infrastructure decision into risk language for the CFO, or to turn a BACEN regulatory directive into a design constraint for the engineering team. Excellent technology chosen without that translation becomes cost without return. Excellent regulation understood without that translation becomes paper compliance. The architect who rides the elevator fluently, without losing technical rigor in the penthouse and without losing business vision in the engine room, is the scarcest and most valuable asset a bank can have.

The building we traversed, and what each floor taught

We began by establishing why the architect needs to ride the elevator (Chapter 1) and mapped the

anatomy of a bank's floors (Chapter 2): the strategy-and-risk penthouse, the intermediate floors of product, operations, and compliance, and the engine room of runtime and data. Chapter 3 addressed the hardest skill, moving up and down without losing context, and Chapters 4 and 5 gave us the correct language: banks are sets of capabilities, not screens, and they operate within regulatory rails that are not obstacles but design constraints with real legal and reputational consequences.

Chapter 6 was the technical heart of the book: the ledger as a business invariant, idempotency as a survival property, and double-entry bookkeeping as the mathematical proof that the system is consistent. Without understanding that chapter, no architecture decision in banking core has a solid foundation. Chapter 7 materialized all of this into a reference architecture on AWS, not as a recipe to copy, but as a map of decisions with explicit trade-offs.

Chapters 8 through 12 built the layers that sustain the core: events as nervous tissue (replacing point-to-point integration with auditable asynchronous contracts), data as a product with traceable lineage (because in banking, data without provenance is data without regulatory value), platform and runtime chosen by operating model rather than hype, generative AI with guardrails that preserve the explainability the regulator demands, and security treated as evidence: not as a checklist. Chapter 13 was the most honest: architecture only exists in production, and designing to operate at three in the morning is as important as designing to scale. Chapters 14 and 15 closed the decision cycle: selling options and recording them in ADRs, turning decisions into mechanisms that survive personnel turnover.

Technology matters enormously, but only becomes architecture when connected to the right problem

Throughout the book I named technologies with deliberate precision: Amazon EventBridge and MSK as the event backbone, Amazon Aurora with Multi-AZ and write-forwarding for strong consistency in the transactional core, Amazon S3 and Lake Formation as a data foundation with column-level access control, Amazon Bedrock with configurable guardrails for generative AI, Amazon EKS with Karpenter for workloads requiring runtime control, AWS CloudTrail and Security Hub as an auditable evidence layer. I named these technologies not to advertise, but because service selection is a trade-off decision, and trade-offs without names are conversations without an object.

But the most common trap I see in banking architecture teams is reversing the order: choosing the technology first and then searching for the problem it solves. This produces architectures that are technically impressive and operationally fragile, systems nobody can operate at three in the morning because they were designed for an architecture presentation, not for a production incident.

The correct order is: understand the business capability, identify the dominant risk in that capability (consistency risk? latency? compliance? availability?), define acceptable trade-offs with the right stakeholders, and only then choose the service that best addresses that risk within the team's operational constraints. A bank that chooses Kafka without a team capable of operating Kafka in production has not bought resilience; it has bought complexity. A bank that chooses Lambda for the transactional core without modeling idempotency boundaries in a serverless environment has not bought agility; it has bought future inconsistency.

Technology without problem context is noise. Technology connected to the right risk, with explicit trade-offs and defined operating mechanisms, is architecture.

The architect as translator of consequence, the real work

The title of this chapter is not a metaphor. In banking, the consequence of an architectural decision shows up in very concrete places: in PIX availability at three in the morning on a Saturday, in the ability to produce a data lineage report for a BACEN audit within forty-eight hours, in the response time of a fraud prevention system that must decide in under two hundred milliseconds, in the ability to roll back a data migration without losing accounting consistency, in the compliance cost that appears on the CFO's income statement.

Translating consequence means being able to make the journey in both directions with equal fluency. Going up: taking a technical decision, for example, adopting eventual consistency in a balance service, and translating it into risk language for the executive committee: "this means that within windows of up to X milliseconds a customer may see a stale balance; the risk of regulatory complaint is Y; the trade-off is Z reduction in latency and W reduction in operational cost". Going down: taking a penthouse directive, for example, "we need to reduce customer onboarding time to two days", and translating it into design constraints for engineering: which SERPRO APIs need to be integrated asynchronously, what is the idempotency model for document reprocessing, how does the customer.approved event propagate downstream without creating synchronous coupling.

This translation is not automatic. It requires the architect to simultaneously maintain two vocabularies, two mental models, and two time scales, the executive's quarterly scale and the system's millisecond scale. Most of the serious problems I have seen in banking projects were not caused by a wrong technical choice. They were caused by loss of context in the transition between floors: a business requirement that reached engineering without the embedded regulatory constraint, or a technical limitation that never rose to the penthouse and was therefore never considered in the timeline decision.

What I take from here, and what I hope you take too

Writing this book was an exercise in riding the elevator in text. Each chapter required finding the right level of abstraction, high enough to be useful to a senior architect who needs to convince an executive committee, technical enough to be useful to an engineer who will implement the idempotency mechanism in production. I do not know whether I got every chapter right, but I know I tried honestly.

What I carry as a central conviction, after sixteen years building financial systems: banking architecture is not about technology, it is about trust. The customer's trust that their money is correct. The regulator's trust that the bank can prove what it did and when it did it. The engineering team's trust that the system can be changed without fear. The executive's trust that the architecture supports the strategy rather than blocking it.

AWS offers the right building blocks for that trust, managed services that reduce operational risk, security primitives that produce auditable evidence, data platforms that allow lineage tracing, AI models that can be configured with guardrails. But the blocks do not assemble themselves. They need an architect who understands the problem before choosing the solution, who records the decision before forgetting the context, who builds mechanisms that survive their own departure from the project.

If this book contributed to your riding the elevator with more fluency, to your speaking with executives without losing technical rigor and with engineering without losing business vision, then it fulfilled its purpose. The work continues. The elevator is waiting.

The 6 principles to take from this book

- The question is not 'which technology to use', it is 'what risk does this technology reduce and

what future option does it create or close'. Technology without that question is cost without architecture.

- Model capabilities, domains, and events before choosing a service. The domain model reveals the contracts; the contracts reveal the trade-offs; the trade-offs reveal the correct service choice.
- In the transactional core, strong consistency and idempotency are not design options, they are survival requirements. Eventual consistency in the ledger is regulatory and reputational risk, not a performance trade-off.
- Security is evidence, not opinion. Compliance is a design constraint, not a layer added at the end. Both need to be present in the first ADR, not in the last sprint before go-live.
- Architecture only exists in production. Design to operate at three in the morning: runbooks, observability with business context, alerts with severity calibrated to financial and regulatory impact.
- Sell options, record them in ADRs, transform them into mechanisms. A decision without a record is a decision that will be remade. A mechanism without an owner is a mechanism that will be ignored. Decisions that survive are the ones that became process.

Questions I frequently hear when closing this book

Is this book an implementation manual or a concepts book?

It is deliberately both, at different levels. Each chapter has a conceptual layer (the why and the trade-off) and a technical layer (the how and the service). A purely conceptual book does not help the engineer implement. A purely technical book does not help the architect justify the decision to the executive committee. The elevator needs both floors.

Do the patterns described here apply to small fintechs or only to large banks?

The principles apply to any institution operating under financial regulation, which includes fintechs with payment licenses, IPs, and SCDs. The implementation scale varies: a small fintech can start with a subset of the patterns and evolve. What does not scale down is the requirement for idempotency in the core and evidence in security, those are regulatory constraints, not size choices.

How do I convince a CTO or CFO who still sees architecture as cost, not investment?

Speak their language: risk and option. Do not say 'we need event sourcing'. Say 'without event traceability, a BACEN audit may require manual transaction reconstruction, the estimated cost of such an incident is X; the cost of implementing the correct pattern now is Y'. When architecture becomes quantified risk language, it stops being cost and becomes insurance. Chapter 14 covers this in detail.

Verdict: what defines the architect who makes a difference in banking

The modern architect in banking is not the deepest technical specialist in the room, there are better engineers in Kafka, SQL, and security. Nor is the most visionary strategist in the penthouse, there are executives with more business and regulatory context. The architect's unique value lies in translating consequence between floors: speaking with executives without losing technical rigor, speaking with engineering without losing business vision, and recording those translations in decisions that survive personnel turnover and deadline pressure.

In banking, this competence determines whether the organization decides with clarity or in the dark. A decision for eventual consistency in the ledger made without the correct translation to the

penthouse is an unpriced regulatory risk. A two-day onboarding directive made without the correct translation to the engine room is an impossible deadline disguised as a goal. The architect who makes that translation honestly, including the uncomfortable trade-offs, the risks nobody wants to name, the technical limitations that contradict the roadmap, is the architect who builds systems that last.

This book was written for that architect. The elevator is waiting. Go up.

References and further reading: The complete references for this chapter and the book are consolidated in the [REFERENCES] block that follows, organized by theme: software architecture and the elevator, banking systems and ledger, AWS and reference services, Brazilian financial regulation (BACEN, CMN, LGPD), and recommended reading for each floor of the building.

To go further

- Gregor Hohpe: The Software Architect Elevator (O'Reilly) (<https://architectelevator.com/book/>)
- BACEN: Sistema de Pagamentos Brasileiro (SPB) e Pix (<https://www.bcb.gov.br/estabilidadefinanceira/pix>)
- AWS Well-Architected Framework (<https://aws.amazon.com/architecture/well-architected/>)
- AWS: Financial Services Industry Lens (<https://docs.aws.amazon.com/wellarchitected/latest/financial-services-industry-lens/financial-services-industry-le>)
- Amazon Bedrock: Guardrails e Knowledge Bases (<https://aws.amazon.com/bedrock/>)
- Série Banco por Dentro (1-3), fernando.moretes.com/studies (<https://fernando.moretes.com/studies>)