E-BOOK - FERNANDO F. AZEVEDO

Banco por Dentro: O Elevador de Arquitetura

Do negócio ao ledger, dos eventos à IA, arquitetura de sistemas financeiros na AWS, para arquitetos e desenvolvedores que sobem e descem o elevador entre estratégia e código.

Fernando F. Azevedo

1ª edição - 2026 - 16 capítulos - ~119 páginas

PARTE I

O Elevador de Arquitetura

O modelo mental de Gregor Hohpe aplicado a bancos: por que o arquiteto sobe e desce entre o andar executivo e a sala de máquinas, e o que se perde quando ele fica preso num andar só.

01 - Por que o arquiteto anda de elevador

O penthouse fala de estratégia, risco e receita. A sala de máquinas fala de idempotência, partição e latência. Arquitetura é o elevador que conecta os dois, e em banco, quem não anda nele decide no escuro.

Em todo banco grande existe um abismo silencioso entre quem decide a estratégia e quem escreve o código que a executa, e esse abismo custa caro, em dinheiro, em reputação e, às vezes, em licença de funcionamento. O arquiteto moderno não é o melhor programador da sala nem o executivo mais articulado do comitê: é a pessoa que sobe e desce entre esses dois mundos sem perder a fluência em nenhum deles. Este livro começa aqui porque tudo o que vem depois, ledgers, eventos, segurança, IA, plataforma, só faz sentido quando você entende por que o elevador existe e por que, em bancos, ele trava com uma frequência que nenhuma outra indústria tolera.

O prédio corporativo e o elevador que ninguém mantém

Gregor Hohpe descreve a empresa moderna como um prédio de muitos andares. No penthouse vivem os executivos: eles falam de estratégia competitiva, apetite a risco, posicionamento regulatório, receita por produto e satisfação do cliente. Na sala de máquinas, no subsolo, vivem os engenheiros: eles falam de filas de mensagens, locks de banco de dados, latência de rede, criptografia de chave assimétrica e janelas de deploy. Entre esses dois extremos há dezenas de andares intermediários, gerentes de produto, líderes técnicos, analistas de negócio, times de compliance, cada um com seu próprio vocabulário, seus próprios incentivos e sua própria visão parcial do sistema.

O problema não é a distância vertical em si. O problema é que o elevador está quebrado. As decisões descem do penthouse como slides de PowerPoint cheios de intenção e vazios de restrição técnica. As informações sobem da sala de máquinas como tickets de incidente e relatórios de capacidade que ninguém no penthouse sabe ler. No meio do caminho, cada andar filtra, traduz mal e adiciona ruído. O resultado é previsível: a estratégia que chegou lá embaixo não é mais a que foi concebida lá em cima, e o sistema que foi construído lá embaixo não é mais o que o negócio precisava.

O arquiteto sênior é, na definição de Hohpe, a pessoa que passa a vida nesse elevador. Não porque goste de reuniões, ninguém gosta, mas porque entende que a única forma de garantir que uma decisão estratégica produza o efeito técnico correto, e que uma limitação técnica seja entendida como risco de negócio antes de virar incidente, é estar presente nos dois andares com credibilidade suficiente para ser ouvido em ambos. Esse é o trabalho. Não é glamoroso. É essencial.

> Dezesseis anos traduzindo entre andares: Ao longo de mais de dezesseis anos trabalhando em sistemas financeiros, de processadoras de cartão a bancos digitais, de corretoras a infraestruturas de pagamento instantâneo, aprendi que o momento mais perigoso em qualquer projeto não é quando o time não sabe a resposta técnica. É quando o time técnico e o time de

negócio acreditam, sinceramente, que estão falando da mesma coisa e não estão. Um executivo diz 'precisamos de alta disponibilidade' e imagina que o sistema nunca cai. O engenheiro ouve 'alta disponibilidade' e pensa em SLO de 99,9% com failover automático. Esses dois mundos são compatíveis, mas não são idênticos, e a diferença entre eles, quando não é explicitada, vira um incidente às 2h da manhã numa sexta-feira antes de um feriado prolongado. Meu trabalho, o tempo todo, foi subir e descer esse elevador carregando contexto em ambas as direções: levando as restrições da sala de máquinas para o penthouse antes que virassem surpresas, e levando as intenções do penthouse para a sala de máquinas antes que virassem sistemas errados.

Por que bancos sofrem mais quando o elevador trava

Toda grande organização tem o problema do elevador travado. Mas bancos pagam um preço desproporcional quando ele falha, por três razões que não existem com a mesma intensidade em nenhuma outra indústria.

Primeiro: dinheiro real não tem rollback. Quando um e-commerce tem um bug de precificação, ele cancela os pedidos, emite um comunicado e segue em frente. Quando um banco credita o valor errado em milhares de contas, o problema jurídico, contábil e regulatório pode durar anos. A irreversibilidade das transações financeiras significa que uma decisão técnica equivocada, sobre idempotência, sobre ordenação de eventos, sobre consistência eventual, não é um débito técnico: é um passivo financeiro real. Isso será o tema central do Capítulo 6, mas precisa ser estabelecido aqui: em banco, a sala de máquinas e o penthouse compartilham o mesmo balanço patrimonial.

Segundo: a licença regulatória é um ativo frágil. Um banco opera porque o Banco Central do Brasil autorizou. Essa autorização pode ser suspensa, restringida ou revogada. O BACEN não aceita 'estávamos em processo de migração' como justificativa para falhas de controle. Decisões de arquitetura sobre segregação de funções, rastreabilidade de operações, criptografia de dados em repouso e em trânsito, e continuidade operacional não são escolhas técnicas opcionais, são condições de existência do negócio. Quando o arquiteto não sobe ao penthouse para explicar que uma determinada escolha de design cria um gap de conformidade, ele não está sendo modesto: está sendo negligente.

Terceiro: confiança é o produto real. Um banco não vende contas correntes. Vende a crença de que seu dinheiro está seguro, que a transação vai completar, que o extrato é verdadeiro. Essa crença é construída ao longo de décadas e destruída em horas. Um Pix que some, um limite que some sem explicação, uma conta bloqueada sem notificação, cada um desses eventos é, na percepção do cliente, uma traição. E cada um deles tem uma causa raiz que mora na sala de máquinas: uma condição de corrida, um timeout mal configurado, uma fila sem dead-letter. O arquiteto que não conecta esses dois mundos deixa o banco vulnerável a danos que nenhum hotfix conserta.

Duas línguas, um único sistema

O maior obstáculo para o elevador funcionar não é a falta de vontade, é a falta de um vocabulário compartilhado. O penthouse e a sala de máquinas falam línguas genuinamente diferentes, e a tentação de fingir que não é assim produz os piores tipos de mal-entendido: os que ninguém percebe até ser tarde.

A tabela a seguir: Dois andares, duas línguas: mapeia os conceitos centrais de cada andar lado a lado. Não é uma tabela de curiosidade: é uma ferramenta de trabalho. Quando um executivo fala em 'resiliência operacional', o arquiteto precisa saber imediatamente que isso se traduz em RPO,

RTO, estratégias de multi-região na AWS, e decisões sobre consistência de dados que têm custo e complexidade mensuráveis. Quando um engenheiro fala em 'latência de P99 acima de 800ms no processamento de eventos', o arquiteto precisa saber traduzir isso para o penthouse como 'um em cada cem Pix está demorando mais do que o limite regulatório permite, e isso é risco de multa e de intervenção do BACEN'.

Essa tradução não é simplificação. Simplificar é perder informação. Traduzir é preservar a consequência enquanto muda o vocabulário. O executivo não precisa saber o que é uma dead-letter queue, mas precisa saber que sem ela, mensagens de transação podem se perder silenciosamente e nunca serão reprocessadas. O engenheiro não precisa saber o custo exato de uma multa do BACEN, mas precisa saber que o campo de auditoria que ele está pensando em omitir para simplificar o schema é um requisito regulatório inegociável.

O arquiteto que domina essa tradução bidirecional não é um intermediário passivo. É um multiplicador de decisões: cada conversa que ele facilita entre andares evita semanas de retrabalho e, em casos extremos, evita incidentes que custam mais do que o projeto inteiro. Nos capítulos seguintes, cada decisão técnica será apresentada com sua tradução para o andar de cima, porque essa é a única forma de arquitetura que realmente funciona em um banco.

O que este capítulo estabelece para o livro inteiro

- O arquiteto sênior é definido pela capacidade de transitar entre o penthouse estratégico e a sala de máquinas técnica com fluência e credibilidade em ambos os andares.
- Em bancos, o elevador travado não é apenas ineficiência organizacional, é risco financeiro, regulatório e reputacional com consequências irreversíveis.
- A pergunta central do arquiteto sênior não é 'qual tecnologia usar?', mas 'que risco de negócio isto reduz, que capacidade habilita, e que compromisso cria para o futuro?'.
- Traduzir entre andares não é simplificar: é preservar a consequência enquanto adapta o vocabulário ao interlocutor.
- Cada capítulo deste livro apresentará decisões técnicas com sua tradução explícita para o andar de negócio, porque arquitetura sem essa ponte não existe de verdade.

Dois andares, duas línguas

Critério Penthouse (negócio) Sala de máquinas (engenharia)

--- --- ---

Vocabulário Margem, risco, NPS, churn, regulação Latência, idempotência, throughput, SLO

Horizonte Trimestre, ano, posicionamento Sprint, release, incidente

Unidade de decisão Capacidade de negócio e investimento Serviço, contrato de API, evento

Medo principal Perder mercado, multa do regulador Acordar às 3h por causa de um deploy

O que o arquiteto entrega Trade-off em linguagem de risco e opção Decisão implementável com mecanismos

A pergunta que define o arquiteto sênior

Existe uma pergunta que separa o arquiteto sênior do arquiteto que ainda está crescendo, e ela não tem nada a ver com conhecimento técnico. Engenheiros excepcionais fazem a pergunta: 'qual é a melhor tecnologia para resolver este problema?' É uma pergunta legítima e necessária. Mas o arquiteto sênior faz uma pergunta diferente, anterior e mais difícil: 'que risco de negócio esta decisão reduz, que capacidade ela habilita para o banco, e que compromisso ela cria para os próximos três a cinco anos?'

A diferença não é semântica. Quando você pergunta qual tecnologia usar, você está olhando para

dentro do sistema. Quando você pergunta que risco reduz e que compromisso cria, você está olhando para o sistema como um instrumento a serviço de um negócio regulado, que tem clientes reais, obrigações legais e uma estratégia que vai mudar. Essa segunda pergunta força o elevador a se mover: ela exige que você suba ao penthouse para entender o contexto de negócio antes de descer à sala de máquinas para escolher a ferramenta.

Na prática, isso significa que quando alguém propõe migrar o processamento de Pix para uma arquitetura event-driven com Amazon EventBridge e Amazon SQS, a pergunta certa não começa com 'EventBridge ou Kafka?'. Começa com: 'Que falha operacional estamos tentando eliminar? Qual é o custo regulatório de uma mensagem perdida nesse fluxo? Essa mudança habilita alguma capacidade nova para o produto, ou é puramente defensiva? E quais times vão precisar mudar como trabalham para que isso funcione em produção?' Só depois de responder essas perguntas, que vivem no penthouse, é que faz sentido descer à sala de máquinas e comparar as propriedades técnicas das opções.

Este livro é organizado em torno dessa pergunta. Cada capítulo começa no andar de negócio, com a capacidade, o risco ou o requisito regulatório, e desce até a implementação técnica na AWS com as trade-offs explicitadas. O elevador vai subir e descer o tempo todo. Prepare-se para a viagem.

> O arquiteto que só fica na sala de máquinas: Existe um padrão de falha que vi repetir em projetos bancários ao longo de anos: o arquiteto tecnicamente brilhante que nunca sobe ao penthouse. Ele produz designs elegantes, escolhe as tecnologias certas, escreve ADRs impecáveis, e entrega um sistema que o negócio não consegue operar, que o compliance não consegue auditar, e que o produto não consegue evoluir sem quebrar tudo. Não porque o design seja ruim tecnicamente. Porque foi concebido sem as restrições e intenções que só existem no andar de cima. Em banco, esse padrão é especialmente perigoso porque o custo do retrabalho não é só tempo de engenharia: é risco regulatório acumulado, é dívida de auditoria, é produto que não chegou ao mercado enquanto o concorrente chegou.

O que este livro é, e o que não é

Este não é um livro de receitas de AWS para bancos. Não é um catálogo de serviços com casos de uso financeiros colados. É um livro sobre como pensar como arquiteto sênior em um ambiente onde as decisões técnicas têm consequências regulatórias, financeiras e reputacionais reais, e onde a única forma de tomar boas decisões é manter o elevador em movimento entre a estratégia e a implementação. Cada capítulo vai subir e descer. Cada decisão técnica vai ser apresentada com seu contexto de negócio. E cada trade-off vai ser nomeado explicitamente, porque em banco, trade-offs não nomeados são riscos não gerenciados.

02 - A anatomia dos andares de um banco

Entre o penthouse e a sala de máquinas existem andares intermediários, produto, jornada, domínio, dado, plataforma. Mapeá-los é o que evita que toda conversa vire 'integração genérica'.

Todo banco tem um penthouse onde se fala em risco, margem e regulação, e uma sala de máquinas onde correm threads, filas e bytes, mas entre esses dois extremos existem pelo menos cinco andares que a maioria dos arquitetos nunca nomeia direito, e é exatamente aí que os projetos encalham. Mapear esses andares intermediários não é exercício acadêmico: é o que separa uma conversa de arquitetura de uma reunião de integração genérica que termina sem

decisão. Neste capítulo desço cada andar com você, nomeio as três distinções que se confundem o tempo todo e mostro o que acontece quando o elevador trava.

> Minha leitura depois de 16 anos em sistemas financeiros: Já participei de centenas de sessões de arquitetura em bancos, de reuniões de diretoria até war rooms de incidente. O padrão de fracasso que mais se repete não é falta de tecnologia: é falta de vocabulário compartilhado entre andares. O time de produto fala em 'jornada de crédito', o time de engenharia fala em 'serviço de proposta', o time de dados fala em 'tabela TB_PROPOSTA', e ninguém percebe que os três estão descrevendo facetas diferentes do mesmo fenômeno de negócio. Quando isso acontece, cada andar constrói sua própria representação do mundo e a integração vira o produto acidental, e o mais caro, do projeto. O diagrama que apresento neste capítulo é a minha ferramenta de alinhamento número um em qualquer engajamento bancário.

O prédio tem mais andares do que você imagina

A metáfora do elevador de Gregor Hohpe coloca estratégia no topo e implementação na base, e isso é correto, mas insuficiente para um banco. Um banco é uma das organizações mais estratificadas que existem: regulação do Banco Central, apetite de risco do conselho, metas de produto do C-level, jornadas do cliente desenhadas por UX, domínios de negócio geridos por times estáveis, eventos que carregam fatos auditáveis, dados que precisam de linhagem rastreável, plataformas que abstraem infraestrutura e operações que garantem SLAs sob fiscalização do BACEN e do Banco Central Europeu quando há filiais no exterior.

Como mostra o diagrama a seguir, eu organizo esses andares em sete camadas: Estratégia (apetite de risco, posicionamento competitivo, obrigações regulatórias), Capacidades de Negócio (o que o banco sabe fazer de forma repetível e mensurável), Produto e Jornada (como essas capacidades são empacotadas e entregues ao cliente), Domínios e Eventos (onde vivem as regras, as decisões e os fatos de negócio), Dados (linhagem, qualidade, governança e produto analítico), Plataforma e Runtime (o que abstrai a infraestrutura dos times de produto) e Operação e Segurança (evidência contínua de confiabilidade e conformidade).

Cada andar tem seu próprio vocabulário, seus próprios artefatos e seus próprios stakeholders. O arquiteto que sabe se mover entre eles, subindo para traduzir uma decisão técnica em linguagem de risco, descendo para traduzir uma diretriz regulatória em requisito de design, é o que entrega valor real. O que fica preso num único andar, seja ele o penthouse das estratégias ou a sala de máquinas do Kubernetes, perde a capacidade de influenciar o que importa.

As três distinções que se confundem o tempo todo

Sem nomear corretamente os andares intermediários, três confusões se instalam e transformam qualquer iniciativa bancária em CRUD distribuído caro e frágil.

Capacidade Tela. Uma capacidade de negócio é uma função que o banco executa de forma repetível e com resultado mensurável: 'Concessão de Crédito Pessoal', 'Liquidação de TED', 'Gestão de Garantias'. Uma tela é uma interface que acessa essa função. Confundir os dois leva a roadmaps que descrevem funcionalidades de UI e nunca questionam se a capacidade subjacente é saudável. Já vi bancos reformularem o aplicativo de crédito três vezes em dois anos sem tocar no motor de decisão de crédito, que continuava produzindo taxas de inadimplência acima do esperado. A tela mudou; a capacidade não evoluiu.

Domínio Microserviço. Um domínio é uma fronteira de linguagem, decisão e responsabilidade, um espaço onde um conjunto de conceitos tem significado preciso e onde um time tem autoridade para tomar decisões sem pedir permissão para outro time. Um microserviço é uma unidade de

deployment. Você pode ter um domínio implementado em um monólito bem estruturado ou fragmentado em trinta microserviços sem coesão. O que importa primeiro é a fronteira de responsabilidade; a granularidade de deployment é uma decisão técnica derivada. Bancos que pulam direto para microserviços sem definir domínios criam acoplamento distribuído, o pior dos dois mundos.

Evento Mensagem Técnica. Um evento de negócio é um fato que aconteceu no mundo real e que tem relevância auditável: 'PropostaDeEmprestimoAprovada', 'LimiteDeCreditoRevogado', 'TransacaoSuspeitaIdentificada'. Ele carrega contexto suficiente para que qualquer consumidor entenda o que ocorreu sem precisar consultar o sistema de origem. Uma mensagem técnica é um envelope de transporte. Tratar eventos como mensagens técnicas é o que produz sistemas onde o consumidor precisa fazer oito chamadas REST para reconstruir o contexto de um fato que o produtor já conhecia inteiramente. Esse antipadrão custa caro em latência, acoplamento e rastreabilidade, três dimensões críticas sob fiscalização do BACEN.

Por que essas três distinções importam tanto num banco

- Capacidades de negócio são o vocabulário do penthouse, sem elas, o roadmap técnico não tem âncora estratégica.
- Domínios bem definidos são a pré-condição para times autônomos, autonomia sem fronteira é caos com latência.
- Eventos de negócio são a matéria-prima da auditoria e da rastreabilidade exigidas pelo BACEN e pela Resolução CMN 4.893.
- Confundir os três transforma integração em produto acidental, o mais caro e o mais difícil de evoluir.
- Nomear corretamente cada andar reduz o custo de onboarding de novos arquitetos e engenheiros em projetos de alta rotatividade.
- A distinção evento mensagem técnica é o que habilita event sourcing e CQRS como padrões de auditoria, não apenas de performance.

Os andares do elevador num banco

A conversa de arquitetura sobe e desce por estes níveis. Cada subida traduz detalhe em risco e capacidade; cada descida traduz intenção em decisão implementável.

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)

Fluxos

- estrategia -> capacidade: desce: intenção -> capacidade
- capacidade -> jornada: expressa em
- jornada -> dominio: realizada por
- dominio -> dados: produz/consome
- dominio -> plataforma: roda em
- plataforma -> operacao: operada por
- operacao -> estrategia: sobe: risco, custo, capacidade

Antipadrões: quando o elevador trava entre andares

Reconhecer um elevador travado é tão importante quanto saber operá-lo. Ao longo da minha carreira em sistemas financeiros, aprendi a identificar sintomas específicos que indicam que a organização perdeu a capacidade de mover contexto entre andares, e que qualquer decisão arquitetural tomada nesse estado vai custar caro para desfazer.

Comitê que só aprova tecnologia. Quando o único fórum de governança de arquitetura discute escolhas de framework e versão de biblioteca, mas nunca questiona se a capacidade de negócio que está sendo construída resolve uma dor real, o elevador está preso no subsolo técnico. A consequência típica é um portfólio de serviços tecnicamente corretos que ninguém usa ou que duplicam funcionalidade sem que ninguém perceba.

Roadmap sem dor de negócio. Um roadmap que lista iniciativas como 'Migração para Kubernetes', 'Refatoração do Módulo de Crédito' ou 'Adoção de GraphQL' sem vincular cada item a uma capacidade de negócio com métrica de resultado é um roadmap de sala de máquinas disfarçado de estratégia. Já vi esse padrão consumir orçamentos de oito dígitos sem mover nenhum indicador de negócio.

Diagrama com 40 caixas e zero donos. Quando o diagrama de arquitetura tem dezenas de componentes e nenhum deles tem um time ou pessoa responsável identificada, o que parece ser um mapa é na verdade uma fotografia de dívida técnica. Sem dono, não há decisão; sem decisão, não há evolução.

Decisões só na memória de uma pessoa. Em bancos com alta rotatividade de arquitetos, e isso é mais comum do que se admite, o conhecimento sobre por que um sistema foi desenhado de determinada forma existe apenas na cabeça de quem o construiu. Quando essa pessoa sai, o time começa a desfazer decisões corretas por não entender o contexto que as motivou. Architecture Decision Records (ADRs) não são burocracia; são o mecanismo que mantém o elevador operando mesmo com troca de operadores. Voltarei a esse tema no Capítulo 14.

> Pular andares é construir a coisa certa no lugar errado: O erro mais caro que já vi em projetos bancários não foi escolher a tecnologia errada, foi construir a solução tecnicamente correta no andar errado. Uma regra de negócio crítica implementada diretamente num pipeline de dados em vez de num domínio de negócio com dono claro. Um evento de negócio modelado como campo de banco de dados em vez de fato imutável e auditável. Uma capacidade de negócio inteira vivendo dentro de um único microserviço sem fronteira de domínio. Cada um desses casos cria uma dívida que não é técnica, é arquitetural, e arquitetural é mais difícil de pagar porque exige realinhamento organizacional, não apenas refatoração de código.

Como usar o mapa de andares no dia a dia

O diagrama dos andares do elevador não é um artefato para apresentar uma vez e arquivar. Eu o uso como ferramenta de diagnóstico ativo em três situações recorrentes em projetos bancários.

Na entrada de um novo engajamento, uso o diagrama para fazer uma pergunta simples a cada stakeholder: 'Em qual andar você passa a maior parte do seu tempo e em qual andar você sente mais dificuldade de ser compreendido?' As respostas revelam onde estão as lacunas de comunicação antes de qualquer análise técnica. Um CTO que responde 'passo o tempo no andar de plataforma mas tenho dificuldade com o andar de estratégia' me diz que há um problema de tradução entre o que a tecnologia entrega e o que o negócio espera, e que meu trabalho prioritário é construir essa ponte.

Em revisões de design, uso o diagrama para verificar se cada decisão foi tomada no andar correto. Uma decisão sobre granularidade de eventos pertence ao andar de Domínios e Eventos, não ao andar de Plataforma. Uma decisão sobre retenção de dados pertence ao andar de Dados com input do andar de Estratégia (regulação), não ao andar de Operação. Quando uma decisão está sendo tomada no andar errado, ela geralmente otimiza para o critério errado.

Em discussões de incidente, o diagrama ajuda a separar causa raiz técnica de causa raiz arquitetural. Um incidente de latência pode ter causa técnica (configuração de pool de conexões) ou causa arquitetural (um domínio que virou gargalo porque absorveu responsabilidades de três outros domínios). Tratar a segunda como a primeira é o que garante a reincidência.

O objetivo final deste capítulo é simples: antes de escrever uma linha de código ou desenhar uma caixa num diagrama, você precisa saber em qual andar está. Essa consciência de localização, essa capacidade de dizer 'estou tomando uma decisão de domínio, não de plataforma', é o que distingue um arquiteto que constrói sistemas duráveis de um que constrói sistemas que funcionam na demo e falham em produção. No próximo capítulo, explorarei como manter esse contexto ao subir e descer, porque o elevador em movimento é onde o trabalho real acontece.

Perguntas frequentes sobre os andares do banco

Preciso ter todos os sete andares formalizados antes de começar a construir?

Não, mas preciso ter pelo menos os andares adjacentes ao que estou construindo. Se estou desenhando um domínio, preciso entender a capacidade de negócio acima e a plataforma abaixo. Formalizar tudo de uma vez é waterfall com outro nome.

Como convencer um time de engenharia a pensar em capacidades de negócio se eles só querem falar de serviços?

Começo pela dor deles: mostro como a falta de fronteira de capacidade é a razão pela qual o mesmo bug aparece em três serviços diferentes. Quando a conexão entre confusão técnica e ausência de vocabulário de negócio fica visível, a resistência cai.

O modelo de andares se aplica a fintechs pequenas ou só a bancos grandes?

Aplica-se a qualquer organização que processa dinheiro de terceiros sob regulação. Numa fintech pequena, uma pessoa pode habitar vários andares ao mesmo tempo, mas os andares existem e as confusões entre eles causam os mesmos danos, só que mais rápido porque há menos margem para erro.

O que este capítulo muda na sua prática

Depois deste capítulo, você deve ser capaz de entrar em qualquer reunião de arquitetura bancária e identificar em qual andar a conversa está acontecendo, e se ela deveria estar acontecendo em outro. Deve conseguir nomear a diferença entre capacidade, domínio e evento sem hesitar, e reconhecer os quatro sintomas de elevador travado antes que eles se tornem incidentes de

produção ou fracassos de projeto. Mais importante: deve entender que pular andares não é agilidade, é construir a coisa certa no lugar errado, e em sistemas financeiros esse custo é sempre maior do que parece no momento da decisão.

03 - Subir e descer sem perder contexto

Andar de elevador é uma habilidade treinável: subir é transformar detalhe técnico em risco, custo e opção; descer é transformar intenção estratégica em decisão implementável. Este capítulo dá o método.

Subir e descer entre estratégia e sala de máquinas não é talento inato, é uma habilidade com método, praticável e ensinável. O arquiteto que domina essa travessia não apenas projeta sistemas melhores: ele se torna o único profissional na sala capaz de traduzir consequência nos dois sentidos, protegendo o banco de decisões técnicas invisíveis que viram risco regulatório, e de visões estratégicas grandiosas que nunca encontram chão de implementação.

O que significa subir, e por que a maioria para no mezanino

Subir não é simplificar. Subir é reenquadrar: pegar um detalhe técnico e expressá-lo na moeda que circula no andar de cima, risco, custo, capacidade de negócio, opção estratégica. A maioria dos arquitetos para no mezanino: sobe o suficiente para falar com gerentes de produto, mas não o suficiente para sentar com o Chief Risk Officer ou com o diretor de conformidade e ser levado a sério.

Tome um exemplo concreto que atravessa este livro inteiro: idempotência ponta a ponta no Pix. No andar técnico, idempotência é um atributo de design, cada operação pode ser repetida sem efeito colateral adicional. Isso envolve chaves de idempotência no SPI, deduplicação no ledger, rastreabilidade de EndToEndId, e controle de estado em eventos assíncronos. É real, é complexo, e a maioria dos engenheiros consegue descrever isso com precisão.

Mas o executivo no penthouse não compra 'idempotência'. Ele compra a frase que só o arquiteto pode construir: 'essa decisão de design elimina uma classe inteira de pagamento duplicado, que significa perda financeira direta, reclamação formal no BACEN e risco de imagem em escala de rede social, e simultaneamente cria a opção de multiplicar o volume transacional por cinco sem reescrever o core de pagamentos.' Agora você está no andar certo. Você transformou um atributo técnico em risco evitado, em custo evitado, e em uma opção real: no sentido financeiro do termo: a capacidade de agir no futuro sem pagar o custo hoje.

A heurística que uso é direta: se você não consegue escrever uma frase que faça sentido para o CRO e outra que faça sentido para o engenheiro de plataforma, você ainda não entendeu o problema completamente. Não é sobre ter dois discursos, é sobre ter uma compreensão profunda o suficiente para que a mesma verdade caiba em registros diferentes.

> Minha visão: a frase dupla como teste de compreensão: Depois de dezesseis anos trabalhando em sistemas financeiros, de corretoras a bancos digitais, de migrações de mainframe a plataformas Pix em produção, aprendi que o arquiteto que não consegue escrever as duas frases não está com problema de comunicação. Está com problema de entendimento. Comunicação é consequência; compreensão é causa. Quando forço essa disciplina em revisões de arquitetura, invariavelmente descubro decisões que pareciam técnicas mas eram, na verdade, escolhas de negócio não reconhecidas, e vice-versa. A frase dupla não é retórica:

é um instrumento diagnóstico.

Heurística: escreva sempre as duas frases: Para qualquer decisão de arquitetura, escreva: (1) uma frase no andar de cima, o que isso significa em termos de risco, custo ou opção para o negócio; (2) uma frase no andar de baixo, qual restrição de design, padrão ou mecanismo implementa essa intenção. Se você não consegue escrever as duas com precisão, você ainda não entendeu o problema. Não avance para a solução.

O que significa descer, e a regra de não descer mais do que o necessário

Descer é o movimento inverso e igualmente crítico: pegar uma pressão estratégica e traduzi-la em restrições de design implementáveis. A palavra-chave é restrição, não solução. O arquiteto que desce com uma solução pronta roubou do engenheiro o espaço de criatividade e responsabilidade. O arquiteto que desce com restrições claras e critérios explícitos habilitou a equipe a encontrar a melhor solução dentro do espaço correto.

Considere a pressão estratégica: 'reduzir o tempo de aprovação de crédito de dias para minutos'. No penthouse, isso é uma decisão de posicionamento competitivo, o banco quer capturar o momento de intenção do cliente, que tem meia-vida de minutos em canais digitais. Descer essa intenção de forma ingênua produz um requisito vago: 'o sistema deve ser rápido'. Isso não é arquitetura, é desejo.

Descer com método produz um conjunto de restrições e decisões encadeadas:

- • Motor de decisão com política versionada: a lógica de crédito precisa ser auditável, testável e modificável sem redesploy do core. Isso implica separar o motor (regras, modelos, variáveis) do runtime de execução, uma restrição de design, não uma escolha de tecnologia.
- • Simulação síncrona com timeout e fallback: a jornada do cliente exige resposta em segundos. O design deve prever o que acontece quando os dados de bureau chegam com latência acima do SLO, o fallback não é erro, é política de negócio codificada.
- • Formalização assíncrona por eventos: a aprovação pode ser comunicada em tempo real, mas a constituição do contrato, a atualização do ledger e a notificação regulatória acontecem de forma assíncrona, garantindo consistência eventual sem bloquear a jornada.
- • SLO de jornada, não de serviço: o compromisso de desempenho não é 'o microserviço de score responde em 200ms', é 'o cliente recebe decisão em menos de 90 segundos em 99% dos casos'. Isso muda o que você monitora, o que você alerta, e o que você reporta.

A regra que carrego comigo é: nunca desça mais do que o necessário. Cada andar que você desce sem necessidade aumenta o risco de over-specification, de transformar uma restrição legítima em uma solução prematura que amarra a equipe e cria dívida técnica antes do primeiro commit.

O roteiro do elevador: cinco paradas para não perder contexto

1. Parada 1: Declare a capacidade e o resultado: Antes de qualquer diagrama ou decisão técnica, articule a capacidade de negócio que está sendo construída ou protegida, e o resultado mensurável esperado. 'Processar pagamentos Pix com idempotência garantida' é uma capacidade. 'Eliminar reclamações de duplicidade no BACEN e habilitar crescimento de volume sem reescrita' é o resultado. Se você não consegue declarar os dois com precisão, volte ao penthouse.

2. Parada 2: Modele domínios, eventos e dados: Desça ao andar de domínio: identifique os bounded contexts relevantes, os eventos de negócio que carregam estado e intenção, e os dados

que precisam de linhagem auditável. Nesta parada, você ainda não escolheu tecnologia, você está mapeando o espaço do problema com a linguagem do negócio. Em um banco, isso significa identificar quais eventos são fatos imutáveis (transações, aprovações, recusas) e quais são projeções derivadas.

3. Parada 3: Compare opções com critérios explícitos: Nunca apresente uma única solução. Gere pelo menos três opções arquiteturais e avalie-as contra critérios explícitos derivados das restrições identificadas nas paradas anteriores: custo operacional estimado, complexidade de operação, aderência regulatória, reversibilidade, tempo até produção. Os critérios devem ser visíveis e rastreáveis, não implícitos na escolha final. Isso protege a decisão de revisão futura e cria responsabilidade compartilhada.

4. Parada 4: Registre a decisão e crie mecanismos: Uma decisão não registrada não existe, ela se torna folclore. Use Architecture Decision Records (ADRs) com contexto, opções consideradas, critérios, decisão e consequências previstas. Mais importante: crie mecanismos que tornem a decisão difícil de violar inadvertidamente, guardrails de IaC, políticas de SCPs na AWS, testes de contrato, alertas de desvio. A decisão deve viver no código e na infraestrutura, não apenas no documento.

5. Parada 5: Revise em produção com dados reais: A arquitetura só existe de verdade em produção, o capítulo 13 aprofunda isso. Nesta parada, o arquiteto volta ao elevador com dados observados: latência real de jornada versus SLO declarado, taxa de eventos de compensação (indicador de falhas de idempotência), custo real de infraestrutura versus estimativa. Esses dados sobem ao penthouse como evidência para revisar premissas estratégicas ou descem à sala de máquinas como restrições corrigidas. O ciclo nunca fecha, ele itera.

Manter contexto enquanto o elevador se move

O maior risco da travessia não é subir ou descer, é perder o fio de contexto no meio do caminho. Isso acontece de formas previsíveis: a reunião de estratégia termina e o arquiteto vai direto para uma sessão de refinamento técnico sem registrar as restrições que acabou de ouvir; ou o engenheiro traz um problema de performance e o arquiteto responde com uma solução técnica sem verificar se o problema tem relevância no andar de cima.

O mecanismo que uso para preservar contexto é deliberadamente simples: um parágrafo de contexto no topo de cada ADR e de cada documento de design, escrito antes de qualquer decisão técnica, que responde a três perguntas, qual capacidade de negócio está em jogo, qual o risco se essa capacidade falhar, e qual a pressão de tempo que condiciona a decisão. Esse parágrafo é o cordão umbilical entre o penthouse e a sala de máquinas. Quando a equipe discute opções técnicas, ele está sempre visível.

No contexto bancário brasileiro, manter esse contexto tem uma dimensão regulatória adicional. O BACEN não pergunta qual tecnologia você usou, ele pergunta qual risco você gerenciou e como você pode provar isso. Quando o arquiteto sobe com evidências técnicas traduzidas em linguagem de risco, e desce com restrições regulatórias traduzidas em decisões de design, ele está construindo a ponte que torna o banco auditável por design, não por esforço retroativo.

Isso também muda a natureza das revisões de arquitetura. Em vez de sessões onde engenheiros apresentam diagramas e executivos aprovam sem entender, você tem conversas onde cada decisão tem uma frase no andar de cima e uma no de baixo, e qualquer pessoa na sala pode verificar a coerência entre as duas. Esse é o ambiente onde arquitetura de qualidade acontece: não em isolamento técnico, mas em diálogo contínuo entre andares.

O elevador como prática institucional, não habilidade individual

Tudo que descrevi até aqui pode soar como uma habilidade pessoal do arquiteto, e em parte é. Mas o objetivo final não é ter um arquiteto que anda de elevador: é ter uma organização que sabe andar de elevador. Isso significa times de produto que entendem as restrições técnicas que condicionam suas decisões de roadmap. Significa engenheiros que conhecem o risco de negócio por trás dos SLOs que estão implementando. Significa executivos que conseguem ler um ADR e entender por que uma decisão foi tomada, mesmo sem entender os detalhes de implementação.

Essa maturidade não acontece por decreto. Ela acontece quando o arquiteto pratica o método de forma consistente e visível, quando cada decisão importante tem as duas frases, quando cada revisão de arquitetura começa com contexto de negócio, quando cada incidente de produção é analisado tanto no andar técnico quanto no andar de risco. Com o tempo, a linguagem do elevador se torna a linguagem da organização.

Nos capítulos seguintes, esse método vai se materializar em domínios específicos: o capítulo 06 vai descer ao ledger e à idempotência com a precisão que o Pix exige; o capítulo 08 vai mostrar como eventos são o tecido nervoso que conecta os andares em tempo real; o capítulo 12 vai demonstrar que segurança é evidência, e evidência só existe quando o arquiteto sabe subir com ela. Em todos esses casos, o elevador é o mesmo. O que muda é o andar de destino e a moeda de tradução.

A habilidade de subir e descer sem perder contexto é, em última análise, o que distingue o arquiteto que projeta sistemas de quem apenas os descreve. É o que torna a arquitetura consequente, não como artefato, mas como prática viva dentro de um banco que precisa ser confiável, auditável e capaz de evoluir.

Pontos-chave do capítulo

- Subir é reenquadrar detalhe técnico na moeda do andar de cima: risco evitado, custo evitado, opção criada. Não é simplificar, é traduzir com precisão.
- Descer é traduzir pressão estratégica em restrições de design implementáveis, nunca em soluções prontas. Restrição habilita; solução prematura amarra.
- A heurística das duas frases é um teste de compreensão: se você não consegue escrever uma frase para o CRO e outra para o engenheiro de plataforma, você ainda não entendeu o problema.
- O roteiro do elevador tem cinco paradas: declarar capacidade e resultado; modelar domínios e eventos; comparar opções com critérios explícitos; registrar a decisão e criar mecanismos; revisar em produção com dados reais.
- No contexto bancário brasileiro, o BACEN exige que riscos sejam gerenciados e provados, o arquiteto que traduz decisões técnicas em linguagem de risco constrói auditabilidade por design.
- O objetivo final não é um arquiteto que anda de elevador: é uma organização inteira que aprendeu a linguagem da travessia entre estratégia e implementação.

PARTE II

O Banco por Dentro

O negócio antes da tecnologia: intermediação, spread, o mapa de capacidades, os trilhos de

pagamento, a regulação do BACEN e o ledger de dupla entrada como o coração contábil.

04 - O que um banco faz, capacidades, não telas

Antes de qualquer diagrama de sistema: intermediação financeira, spread, e o mapa de capacidades que descreve o que o banco sabe fazer, independente de como está implementado hoje.

Antes de desenhar qualquer diagrama de sistema, o arquiteto precisa entender o que o banco realmente faz, não quais telas ele exibe, não quais APIs ele expõe, mas quais funções de negócio ele executa com consequências financeiras reais e irreversíveis. Este capítulo abre a Parte II com uma pergunta deliberadamente simples: o que é um banco, visto de dentro? A resposta muda tudo que vem depois.

O banco como sistema distribuído com garantias fortes

Quando explico bancos para engenheiros de software experientes, uso uma analogia que provoca desconforto produtivo: um banco é um cache distribuído com garantias excepcionalmente fortes. Um depósito é um write: o cliente entrega dinheiro e o banco registra uma obrigação. Um saque é um read com side-effect: o banco devolve valor e decrementa a obrigação. Até aqui, familiar. O que rompe a analogia confortável é o crédito: quando o banco empresta R$ 100 mil, ele está fazendo um read especulativo sobre dinheiro que não existe fisicamente naquele instante naquela conta. O banco está apostando que vai captar mais do que vai perder em calotes, que o spread vai cobrir o custo de operação e o risco de inadimplência, e que a regulação vai manter as regras do jogo estáveis o suficiente para que o modelo funcione.

O que torna isso radicalmente diferente de qualquer sistema de software convencional é a ausência de tolerância a inconsistência. Em sistemas distribuídos modernos aceitamos eventual consistency como trade-off razoável: uma mensagem perdida é reprocessada, um estado divergente é reconciliado em segundos. Em um banco, cada inconsistência é dinheiro real saindo ou entrando de forma incorreta. Não existe 'perdemos uma mensagem, tudo bem'. O saldo de um cliente não pode ser eventualmente consistente, ele precisa ser exato, auditável e rastreável a cada centavo, a qualquer momento, inclusive durante um processo de auditoria do BACEN que pode acontecer sem aviso.

Isso não significa que bancos não usam arquiteturas assíncronas ou sistemas distribuídos, usam, e cada vez mais. Significa que as garantias de consistência precisam ser explicitamente projetadas, não assumidas como default do framework. O arquiteto que sobe do engine room para o penthouse carrega essa consciência: o que parece uma decisão técnica sobre idempotência ou ordering de eventos é, na verdade, uma decisão sobre risco financeiro e conformidade regulatória. Voltaremos a isso com profundidade no Capítulo 06, quando tratarmos do ledger e da idempotência como mecanismos centrais.

Glossário mínimo do andar de negócio

- Intermediação financeira: O banco capta dinheiro de quem tem sobra e empresta para quem precisa, assumindo o risco e o prazo no meio.
- Spread bancário: A diferença entre a taxa que o banco cobra de quem toma e a que paga a quem deposita. É a margem do negócio central.
- Ledger (livro-razão): O registro contábil central, imutável e auditável, onde todo movimento de dinheiro é escriturado por dupla entrada.

• Liquidação (settlement): O momento em que o dinheiro de fato troca de mãos e o débito/crédito se torna definitivo e irreversível.

• BACEN: Banco Central do Brasil, regulador e supervisor; opera a infraestrutura central (SPI, STR) e define as restrições de design do sistema.

• PLD/FT: Prevenção à Lavagem de Dinheiro e ao Financiamento do Terrorismo, controles obrigatórios, não opcionais, embutidos em produto e arquitetura.

> Por que a analogia do cache importa para o arquiteto: Uso essa analogia não para simplificar o banco, mas para criar um ponto de entrada honesto para engenheiros que chegam com padrões de sistemas web na cabeça. O perigo real não é o engenheiro que não conhece bancos, esse pergunta. O perigo é o engenheiro que acha que conhece porque já integrou um gateway de pagamento. Intermediação financeira, spread, risco de crédito e obrigação regulatória são conceitos que mudam a natureza das decisões técnicas. Quando o arquiteto não os internaliza, ele projeta um sistema que funciona em demo e quebra em produção no primeiro fechamento contábil.

Intermediação financeira: o negócio central antes de qualquer feature

Um banco existe, em sua essência econômica, para fazer uma coisa: captar dinheiro de quem tem e emprestar para quem precisa, cobrando mais do que paga. Essa diferença, o spread, é a receita bruta do negócio de intermediação. Sobre ela incidem o custo de operação (pessoal, tecnologia, agências, compliance), o custo de risco (provisões para inadimplência, capital regulatório exigido pelo BACEN via Basileia III/IV) e, por fim, o lucro.

Por que isso importa para o arquiteto? Porque cada capacidade técnica que construímos serve diretamente a um dos lados dessa equação. O sistema de captação (contas correntes, poupança, CDBs, LCIs) serve ao lado do passivo, o banco está devendo ao depositante. O sistema de crédito (empréstimos pessoais, financiamentos, cartão de crédito rotativo) serve ao lado do ativo, o banco é credor. O sistema de pagamentos (Pix, TED, boleto, cartões) é a infraestrutura de movimentação que conecta os dois lados e, cada vez mais, é também fonte de receita por tarifas e dados transacionais.

Quando um arquiteto não entende essa estrutura, ele trata todos os sistemas como equivalentes em criticidade e tolerância a falha. Mas eles não são. Uma falha no sistema de crédito durante a janela de análise de uma proposta pode ser recuperada em minutos sem consequência financeira direta. Uma falha no sistema de liquidação do Pix durante o horário de funcionamento do SPI pode gerar multas regulatórias, danos reputacionais e, em casos extremos, intervenção do BACEN. A tolerância a falha não é uma decisão técnica, é uma função do risco de negócio e da regulação aplicável.

O glossário mínimo do andar de negócio, apresentado a seguir, formaliza esses conceitos com a precisão necessária para que engenheiros e arquitetos falem a mesma língua que os diretores de risco e os auditores. Sem esse vocabulário compartilhado, o elevador não funciona, o arquiteto sobe ao penthouse e não consegue se comunicar, ou desce ao engine room e perde o contexto regulatório.

O que o spread financia, e por que o arquiteto precisa saber

• Custo de captação: juros pagos a depositantes e investidores; define o piso da taxa de crédito.

• Provisão para devedores duvidosos (PDD): reserva contábil obrigatória proporcional ao risco da carteira de crédito.

• Capital regulatório (Basileia): parcela do patrimônio que o banco não pode usar

operacionalmente, custo de oportunidade direto.

• Custo operacional de TI: cada sistema que construímos é uma linha no P&L; arquitetura ineficiente corrói margem.

• Risco de liquidez: o banco precisa honrar saques mesmo quando a carteira de crédito está imobilizada, o sistema de caixa é crítico.

• Receita de serviços (tarifas, câmbio, custódia): complementa o spread e depende diretamente da qualidade dos sistemas de pagamento e custódia.

Mapa de capacidades de negócio de um banco

O que o banco sabe fazer, agrupado por família de capacidade. Nenhuma caixa aqui é um sistema, são funções de negócio que sobrevivem a qualquer reescrita técnica.

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)

Fluxos

• conta -> ledger: escritura

• credito -> ledger: escritura

• pix -> liq: dispara

• liq -> ledger: debita/credita

• kyc -> conta: habilita

• fraude -> pix: monitora

Capacidade não é tela: o mapa que precede qualquer solução

O erro mais comum que vejo em projetos de modernização bancária é começar pelo sistema. A equipe recebe um mandato: 'modernizar o crédito consignado' ou 'implementar open finance', e imediatamente parte para escolha de tecnologia, definição de microsserviços e desenho de APIs. O mapa de capacidades nunca é desenhado. O resultado invariável é um sistema que implementa o fluxo atual digitalizado, sem questionar se aquele fluxo faz sentido, e que se torna impossível de evoluir porque ninguém sabe onde uma capacidade termina e outra começa.

Uma capacidade de negócio é uma função que o banco sabe executar, com um resultado

mensurável, independente de como está implementada hoje. 'Concessão de crédito' é uma capacidade. 'Tela de análise de crédito no sistema legado X' não é, é uma implementação específica, possivelmente ruim, de parte dessa capacidade. Essa distinção parece óbvia escrita assim, mas desaparece completamente sob pressão de prazo e quando os stakeholders descrevem o negócio em termos de telas e relatórios.

O diagrama de mapa de capacidades apresentado a seguir organiza as funções de um banco em domínios coesos: conta e relacionamento, crédito, cartões, pagamentos, KYC e onboarding, prevenção a fraudes e PLD, e ledger/contabilidade. Cada domínio tem linguagem própria, modelo de dados distinto, controles regulatórios específicos e, crucialmente, tolerância a falha diferente. Pix exige latência de resposta em milissegundos e disponibilidade próxima de 100% durante o horário do SPI, o BACEN monitora e pune desvios. Crédito exige decisão auditável, simulação reproduzível e formalização com validade jurídica, a latência de minutos é aceitável, a perda de trilha não é. KYC exige evidência persistente, imutável e acessível para auditoria anos depois, o throughput é baixo, mas a durabilidade é absoluta.

Quando o arquiteto não tem esse mapa, tudo vira integração genérica. Cada domínio recebe o mesmo padrão arquitetural, o mesmo SLA, o mesmo modelo de dados. O resultado é um sistema que é simultaneamente caro demais onde poderia ser simples e frágil demais onde precisa ser robusto. O mapa de capacidades é o instrumento que permite ao arquiteto calibrar cada decisão técnica contra o risco de negócio correto: e é por isso que ele vem antes de qualquer diagrama de solução.

> Por que o mapa de capacidades vem antes da solução: Sem o mapa, o arquiteto não tem como responder à pergunta mais importante que vai receber no penthouse: 'se esse sistema falhar, o que o banco perde?' Com o mapa, a resposta é precisa: 'perdemos a capacidade de originação de crédito por N horas, o que impacta X contratos por dia e Y de receita estimada, além de risco de SLA com correspondentes bancários.' Sem o mapa, a resposta é 'o sistema de crédito fica fora do ar', o que não significa nada para um diretor de risco ou para o BACEN.

Cada capacidade tem sua própria linguagem, e o arquiteto precisa falar todas

Uma das habilidades mais subestimadas do arquiteto sênior em bancos é a capacidade de mudar de vocabulário conforme muda de domínio, sem perder precisão técnica. Isso é o que o elevador exige na prática: subir ao andar de crédito e falar em 'score de bureau', 'política de crédito', 'faixa de risco' e 'CCB' (Cédula de Crédito Bancário); descer ao engine room e falar em 'modelo de decisão versionado', 'feature store', 'contrato imutável com hash' e 'event sourcing para auditoria de decisão'. São o mesmo fenômeno, visto de andares diferentes.

O domínio de pagamentos fala em liquidação, compensação, janela D+0, SPI, ISPB e finalidade de transação. O domínio de KYC fala em due diligence, PEP (Pessoa Exposta Politicamente), lista OFAC, CNPJ matriz/filial e prova de vida. O domínio de fraude e PLD fala em tipologia, comunicação ao COAF, limiar de R$ 50 mil, watchlist e análise comportamental. O domínio de ledger fala em partidas dobradas, plano de contas COSIF, competência versus caixa e conciliação de posição.

Cada um desses domínios tem dados que não podem ser compartilhados livremente entre si: não por limitação técnica, mas por exigência regulatória e por princípio de minimização de dados (LGPD aplicada ao contexto financeiro). O arquiteto que desenha um data lake centralizado onde todos os domínios leem e escrevem livremente está criando um problema de governança que vai aparecer na primeira auditoria do BACEN ou na primeira investigação de PLD.

A tabela que acompanha o mapa de capacidades, apresentada na sequência, detalha, para cada domínio, os controles regulatórios primários, o tipo de dado sensível envolvido, a tolerância a latência e a consequência de falha. Ela é o instrumento de calibração que transforma o mapa de capacidades de um artefato estratégico em um guia operacional para decisões de arquitetura. O arquiteto que domina essa tabela consegue, em qualquer reunião técnica ou executiva, conectar instantaneamente uma decisão de design a uma consequência de negócio, e isso é precisamente o que distingue um arquiteto de elevador de um arquiteto de sala de máquinas.

Perguntas frequentes sobre capacidades de negócio em bancos

Posso usar o mesmo microsserviço para duas capacidades diferentes se a lógica for parecida?

Tecnicamente sim, mas o custo tende a ser alto a médio prazo. Capacidades com regulações diferentes evoluem em ritmos diferentes e por razões diferentes, uma mudança de política de crédito não deve forçar um deploy no sistema de pagamentos. A semelhança superficial de lógica esconde divergências profundas de controle, auditoria e tolerância a falha. O critério de separação deve ser o domínio regulatório e o modelo de dado, não a similaridade de código.

O mapa de capacidades substitui o modelo de domínio (DDD)?

Não, eles operam em níveis diferentes. O mapa de capacidades é estratégico: diz o que o banco faz e com que consequências. O modelo de domínio (DDD) é tático: diz como cada capacidade é estruturada internamente em agregados, entidades e serviços. O mapa vem primeiro e informa os bounded contexts do DDD. Sem o mapa, os bounded contexts tendem a refletir a estrutura organizacional atual, não as capacidades de negócio reais.

Como o BACEN usa esse conceito de capacidades na regulação?

O BACEN não usa o termo 'capacidade' explicitamente, mas suas normas (Resolução CMN 4.893 sobre PSTI, Circular 3.909 sobre Pix, Resolução BCB 85 sobre open finance) são escritas em termos de funções de negócio e seus controles, não em termos de sistemas ou tecnologias. Isso significa que o mapa de capacidades é o nível de abstração correto para mapear obrigações regulatórias: cada capacidade pode ser rastreada às normas que a governam, independente de como está implementada.

O mapa de capacidades como pré-condição de qualquer projeto bancário

Nenhum projeto de modernização, migração para cloud ou implementação de nova regulação em um banco deveria começar sem um mapa de capacidades validado com as áreas de negócio, risco e compliance. Não porque seja uma formalidade arquitetural, mas porque sem ele o arquiteto não tem como calibrar criticidade, não tem como definir fronteiras de sistema com fundamento, e não tem como traduzir consequências técnicas em linguagem de risco para o penthouse. O mapa não é o destino: é o instrumento de navegação que torna o elevador funcional.

05 - Os trilhos e as regras

Como o dinheiro se move entre instituições no Brasil: Pix, TED, cartão, boleto, o que o BACEN exige para operar, e o que realmente muda quando você é uma fintech em vez de um banco

completo.

Antes de desenhar qualquer serviço financeiro na AWS, você precisa entender os trilhos sobre os quais o dinheiro vai correr, e as regras que determinam quem tem permissão de operar esses trilhos. Ignorar essa camada é o erro mais caro que um arquiteto pode cometer: você pode construir um sistema tecnicamente impecável que o BACEN embarga na semana do go-live. Este capítulo desce até a sala de máquinas dos pagamentos brasileiros e sobe até o andar de estratégia regulatória, porque as duas perspectivas são inseparáveis.

A distinção que mais importa: autorização não é liquidação

Existe uma confusão conceitual que atravessa equipes inteiras de produto e engenharia, e ela custa caro quando chega à produção: autorização e liquidação são eventos distintos, separados no tempo, e com garantias completamente diferentes.

Autorização é uma promessa. Quando você passa o cartão no terminal, em milissegundos o emissor responde "aprovado", mas nenhum centavo se moveu ainda. O portador recebeu uma reserva de crédito, o lojista recebeu uma garantia condicional, e o sistema inteiro vai viver nesse estado intermediário por horas ou dias até que a liquidação aconteça. No modelo de quatro partes do cartão, portador, emissor, adquirente e bandeira, essa janela entre autorização e liquidação é uma feature deliberada: ela permite compensação, estorno, chargeback, reconciliação em lote. O custo é complexidade e latência financeira.

O Pix colapsou essa janela. Quando uma transação Pix é confirmada, autorização e liquidação acontecem no mesmo evento, em segundos, com finality irrevogável na conta de reservas do BACEN via SPI. Isso é tecnicamente mais difícil, não mais fácil. Não existe janela de correção. Um erro de roteamento, um problema de idempotência, uma falha de reconciliação, tudo isso precisa ser tratado antes da confirmação, porque depois não há como desfazer sem uma nova transação no sentido contrário, com todas as implicações regulatórias que isso carrega.

Essa distinção não é detalhe de produto. Ela define a arquitetura de resiliência, o modelo de compensação de erros, o design de idempotência (que o Capítulo 06 aprofunda), e até o perfil de risco que você precisa comunicar ao board. O arquiteto que não internaliza essa diferença vai projetar sistemas de Pix com a mentalidade de cartão, e vai descobrir o problema em produção.

> Minha leitura sobre a dificuldade real do Pix: Depois de trabalhar em sistemas de pagamento de alta criticidade, a afirmação que mais me irrita é 'Pix é simples porque é só uma API'. O Pix é uma das integrações mais exigentes que um time de engenharia financeira pode enfrentar, precisamente porque a simplicidade da experiência do usuário esconde uma exigência brutal de consistência eventual zero: você não pode errar e corrigir depois. Todo o investimento em idempotência, em circuit breaker, em reconciliação proativa, em trilha de auditoria imutável, ele existe para compensar a ausência dessa janela de correção que o cartão oferece de graça. Quando avalio arquiteturas de clientes, o primeiro sinal de maturidade que procuro é se o time entende essa assimetria.

Os trilhos: cada meio de pagamento tem sua física própria

Pagamentos no Brasil não são um mercado homogêneo. São quatro ecossistemas distintos, cada um com infraestrutura própria, operador próprio, modelo de liquidação próprio e, consequentemente, requisitos arquiteturais distintos. A tabela a seguir: Os trilhos de pagamento: o que cada um é para: organiza essa visão de forma comparativa. Aqui, quero construir a intuição que torna a tabela útil.

O Pix opera sobre dois sistemas do BACEN: o DICT, que resolve a chave (CPF, e-mail, telefone, chave aleatória) para a conta de destino, e o SPI, que executa a liquidação bruta em tempo real nas contas de reservas. Opera 24 horas por dia, 7 dias por semana, 365 dias por ano, sem exceção. Isso significa que a sua arquitetura de disponibilidade não pode ter janelas de manutenção convencionais. Qualquer PSP participante do SPI precisa garantir disponibilidade contratual com o BACEN; o custo de indisponibilidade não é apenas reputacional, é regulatório.

A TED opera sobre o STR (Sistema de Transferência de Reservas), também do BACEN, mas com horário definido, tipicamente até as 17h em dias úteis. A liquidação é bruta e final, mas o modelo de horário cria uma física completamente diferente: existe uma fila, existe um cutoff, e transações fora do horário precisam ser enfileiradas para o próximo dia útil. Arquiteturalmente, isso exige gestão de estado de transações pendentes e lógica de retry com semântica de data.

O boleto opera em modelo de liquidação em lote, com ciclo D+1 ou D+2. É o instrumento mais tolerante a latência, mas também o que exige maior robustez na reconciliação, porque o volume de boletos pagos em caixa, lotérica ou internet banking gera um fluxo de retorno que precisa ser processado e conciliado contra o ledger interno.

O cartão: crédito e débito, opera no modelo de quatro partes já mencionado. O débito liquida em D+1 via bandeira; o crédito pode levar de 28 a 30 dias para o lojista receber, dependendo do contrato com o adquirente. Cada uma dessas janelas é uma decisão de design: quem carrega o risco de crédito durante o intervalo, como o float é gerenciado, e qual é o custo de capital associado.

Os trilhos de pagamento, o que cada um é para

Critério Velocidade Disponibilidade Caso de uso principal

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

Pix Segundos (até ~10s) 24/7/365 Transferência e pagamento instantâneo P2P/P2B

TED Minutos, em horário Dias úteis, janela do STR Altos valores entre instituições

Boleto Horas a 1 dia útil Compensação em lote Cobrança, contas, recebíveis

Cartão Autoriza em ms, liquida em dias 24/7 (autorização) Compra no ponto de venda / e-commerce

O fluxo Pix de ponta a ponta: onde cada decisão técnica mora

Para tornar concreto o que acabei de descrever, o diagrama a seguir: Fluxo ponta a ponta de um Pix saindo: percorre cada etapa desde a intenção de pagamento do usuário até a confirmação de liquidação no SPI. Quero chamar atenção para três pontos críticos que o diagrama torna visíveis.

O primeiro é a resolução de chave via DICT. Antes de qualquer instrução de pagamento ser enviada ao SPI, o PSP pagador precisa consultar o DICT para traduzir a chave Pix em agência, conta e ISPB do banco destino. Essa consulta é síncrona e está no caminho crítico da experiência do usuário. Uma falha aqui não é apenas um timeout, é uma decisão: você retorna erro ao usuário ou tenta cache? Cache de DICT tem implicações de segurança (uma chave pode ter sido transferida para outra conta). O BACEN define TTL máximo para cache de DICT, e ignorar essa regra é um risco regulatório real.

O segundo ponto é a mensageria ISO 20022 entre PSPs via SPI. O protocolo de comunicação entre participantes do SPI é baseado em mensagens ISO 20022, com certificados ICP-Brasil para autenticação mútua. Isso não é uma API REST convencional, é um sistema de mensageria com semântica de entrega garantida, onde cada mensagem tem um identificador de fim a fim (EndToEndId) que precisa ser preservado imutavelmente em toda a cadeia de processamento.

Esse EndToEndId é a chave de idempotência do universo Pix, e qualquer arquitetura que não o trate como cidadão de primeira classe vai ter problemas de duplicidade.

O terceiro ponto é a confirmação de liquidação como evento de negócio. Quando o SPI confirma a liquidação, esse evento precisa propagar para o ledger interno, para o sistema de notificações, para o motor de limites, para a plataforma de PLD/FT, tudo isso de forma consistente. Aqui é onde a arquitetura event-driven (Capítulo 08) se torna não uma escolha estética, mas uma necessidade operacional: o evento de liquidação é o fato imutável a partir do qual todos os sistemas downstream derivam seu estado.

Fluxo ponta a ponta de um Pix saindo

Um Pix atravessa quatro fronteiras de confiança em segundos. Cada seta é um ponto onde idempotência, timeout e reconciliação precisam estar resolvidos, não há 'desfazer'.

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)

Fluxos

- pagador -> app: inicia
- app -> motorpix: idempotency key
- motorpix -> dict: resolve chave
- motorpix -> spi: ordem de pagamento
- spi -> motorD: liquida
- motorpix -> ledgerR: débito
- motorD -> ledgerD: crédito

  PLD/FT não é formulário, atravessa toda a arquitetura: Prevenção à Lavagem de Dinheiro e ao Financiamento do Terrorismo (PLD/FT) é frequentemente tratada como um módulo de compliance que o time jurídico cuida. Esse entendimento é perigoso e errado. PLD/FT é uma restrição arquitetural que atravessa toda a pilha: você precisa de trilha de auditoria imutável de cada transação (retenção mínima de 5 anos pela Resolução BCB nº 44), monitoramento em tempo real de padrões suspeitos (o que exige streaming de eventos, não batch), capacidade de bloqueio imediato de contas sob investigação (o que exige que seu modelo de dados suporte estados de bloqueio sem corrupção do histórico), e rastreabilidade completa de origem e destino de fundos. Em AWS, isso se traduz em escolhas concretas: CloudTrail com Object Lock para imutabilidade, Kinesis ou MSK para streaming de eventos de transação, e um modelo de dados que separa estado operacional de estado regulatório. Qualquer arquitetura que não projete PLD/FT desde o início vai ter que ser reescrita, e reescrever um ledger em produção é uma das operações mais arriscadas que existem.

Regulação como restrição de design: licenças, capital e o SCR

Quando sobe ao andar executivo, o arquiteto precisa traduzir regulação em linguagem de risco de negócio. Quando desce à sala de máquinas, precisa traduzir essa mesma regulação em restrições concretas de design. O elevador entre esses dois andares é onde a maioria dos projetos financeiros falha, ou por arquitetos que nunca subiram ao andar regulatório, ou por executivos que nunca desceram ao andar técnico.

O BACEN opera um sistema de licenças graduado. Um banco múltiplo pode captar depósitos, emprestar, emitir cartão, operar câmbio, mas exige capital mínimo regulatório que pode ser estimado na casa de dezenas a centenas de milhões de reais dependendo das carteiras ativas, além de estrutura de governança, auditoria independente e reporte contínuo ao BACEN. Uma Instituição de Pagamento (IP), regulada pela Resolução BCB nº 80, pode emitir instrumento de pagamento pré-pago, conta de pagamento ou credenciar estabelecimentos, com requisitos de capital e governança proporcionalmente menores. Uma SCD (Sociedade de Crédito Direto) pode conceder crédito com recursos próprios, sem captar depósitos do público.

Cada tipo de licença define o perímetro do que você pode fazer e, consequentemente, o que precisa estar no seu sistema. Uma IP que não pode captar depósitos precisa de um modelo de segregação de recursos de clientes (conta de pagamento não é conta corrente bancária, os recursos precisam ser mantidos em ativos seguros). Uma SCD que concede crédito precisa reportar ao SCR (Sistema de Informações de Crédito do BACEN): e o SCR é bidirecional: você reporta as operações de crédito que concede, e pode consultar o histórico de crédito de um tomador. Isso tem implicações de privacidade (LGPD), de latência (consulta SCR está no caminho de aprovação de crédito) e de integridade de dados (divergência entre seu ledger e o SCR é um problema regulatório grave).

O ponto que quero fixar: regulação não é uma lista de documentos para entregar ao BACEN. É um conjunto de invariantes que o seu sistema precisa manter em produção, continuamente, sob qualquer condição de carga ou falha. Projetar para isso desde o início é incomparavelmente mais barato do que remediar depois.

Banco versus fintech sobre BaaS: o teto de autonomia

Uma das decisões estratégicas mais consequentes para uma empresa que quer operar serviços financeiros é: construir sobre licença própria ou operar sobre Banking as a Service (BaaS) de um parceiro? A tabela a seguir: Banco × fintech: o que muda na prática: mapeia as dimensões dessa escolha. Aqui quero aprofundar o trade-off central que a tabela captura.

Operar sobre BaaS é a rota de menor atrito inicial. Você terceiriza a licença, o capital regulatório, a infraestrutura de conexão ao SPI/STR, e parte da responsabilidade regulatória para o banco parceiro. Em troca, você ganha velocidade de go-to-market que pode ser medida em meses versus anos. Para uma fintech em fase de validação de produto, esse trade-off frequentemente faz sentido.

Mas existe um teto de autonomia que precisa ser entendido antes de se comprometer com essa arquitetura: o ledger de verdade é do parceiro. Quando sua fintech processa uma transação via BaaS, o registro definitivo daquela transação vive no sistema do banco parceiro. Você tem uma visão derivada, um espelho, uma reconciliação, mas não o ledger primário. Isso tem consequências diretas: sua capacidade de inovar em produtos de crédito é limitada pelo que o parceiro expõe via API; sua capacidade de responder a auditorias regulatórias depende da cooperação do parceiro; e se o parceiro mudar sua política comercial ou encerrar o produto BaaS,

você tem um problema de continuidade de negócio que não está sob seu controle.

A maturidade arquitetural de uma fintech pode ser medida, em parte, por quão conscientemente ela gerencia esse teto. As mais sofisticadas constroem um ledger de sombra: uma representação interna de todas as posições financeiras, reconciliada continuamente contra o parceiro BaaS, que serve tanto para autonomia operacional quanto para o dia em que decidirem migrar para licença própria. Essa decisão de design, feita cedo, é o que separa fintechs que conseguem escalar de fintechs que ficam presas na dependência do parceiro.

O arquiteto que entende esse espectro, de fintech sobre BaaS até banco múltiplo com infraestrutura própria, consegue ter uma conversa muito mais honesta com o board sobre o que a empresa está comprando com cada escolha. Velocidade agora versus autonomia depois não é uma escolha óbvia; é uma aposta calculada que precisa ser registrada como decisão arquitetural explícita.

Banco × fintech: o que muda na prática

Critério Banco múltiplo Fintech IP Fintech sobre BaaS

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

Licença BACEN Completa (capta e empresta) Instituição de Pagamento Nenhuma própria, usa a do parceiro

Acesso ao SPB Participante direto Direto ou indireto Indireto, via banco liquidante

Pode manter ledger próprio? Sim, é o core Sim, da conta de pagamento Limitado, o ledger de verdade é do parceiro

Velocidade de lançar Lenta, muito controle Média Rápida, mas com teto de autonomia

Onde a arquitetura trava Peso regulatório e legado Capital e compliance Dependência e limites do BaaS

O que este capítulo fixou

- Autorização é uma promessa em milissegundos; liquidação é o cumprimento, e o Pix colapsa os dois, eliminando a janela de correção que outros instrumentos oferecem.
- Cada trilho de pagamento (Pix/SPI, TED/STR, boleto, cartão) tem física própria: disponibilidade, latência, modelo de liquidação e requisitos arquiteturais distintos.
- O EndToEndId do Pix é a chave de idempotência do sistema, tratá-lo como cidadão de primeira classe em toda a cadeia de processamento é não-negociável.
- PLD/FT é uma restrição arquitetural que atravessa toda a pilha, trilha imutável, monitoramento em tempo real, retenção por anos, não um módulo de compliance isolado.
- O tipo de licença (banco múltiplo, IP, SCD) define o perímetro do que o sistema precisa manter como invariante em produção, não apenas o que pode ser vendido como produto.
- Fintech sobre BaaS ganha velocidade, mas o ledger primário é do parceiro, o teto de autonomia precisa ser gerenciado conscientemente, idealmente com um ledger de sombra desde o início.

O que o arquiteto carrega deste capítulo

Os trilhos e as regras não são contexto de fundo, são o chão sobre o qual toda decisão técnica está apoiada. Um arquiteto que entende a diferença entre autorização e liquidação, que sabe onde o DICT termina e o SPI começa, que lê uma licença de Instituição de Pagamento como uma especificação de requisitos não-funcionais, e que consegue explicar o teto de autonomia do BaaS para um CEO em três minutos, esse arquiteto está operando no elevador, entre o penthouse e a sala de máquinas, que é exatamente onde o valor é criado. Os próximos capítulos vão aprofundar cada uma dessas camadas: o ledger e a idempotência no Capítulo 06, a arquitetura de referência

completa no Capítulo 07, e os eventos como tecido nervoso no Capítulo 08. Mas tudo isso só faz sentido sobre a base que acabamos de construir aqui.

06 - O ledger é o coração, e idempotência é o sangue

O ledger de dupla entrada é o Git do dinheiro: imutável, auditável, append-only. E idempotência não é detalhe de implementação, é o requisito funcional central de qualquer sistema que move dinheiro.

Todo sistema financeiro, por mais sofisticado que pareça na camada de produto, repousa sobre uma primitiva com cinco séculos de idade: o ledger de dupla entrada. Quando esse fundamento é mal implementado, saldos atualizados diretamente, idempotência tratada como detalhe de engenharia, conciliação relegada ao fim do mês, o banco não tem um problema técnico, tem um problema de integridade financeira. Este capítulo desce ao núcleo conceitual para mostrar que ledger e idempotência não são escolhas de design: são requisitos funcionais de primeira classe, e o arquiteto que não os defende no penthouse perde dinheiro na sala de máquinas.

O ledger é o Git do dinheiro

Pense no que o Git faz: ele nunca sobrescreve um commit. Cada mudança é um novo objeto imutável; o estado atual do repositório é uma projeção sobre o histórico. O ledger contábil funciona exatamente assim, e não por acidente, essa propriedade foi formalizada por Luca Pacioli no século XV precisamente porque dinheiro exige rastreabilidade absoluta.

A regra é simples e inegociável: nunca faça UPDATE em saldo. Sempre faça INSERT de lançamentos. O saldo de uma conta é a soma algébrica de todos os lançamentos associados a ela desde sua abertura. Isso significa que a pergunta "qual era o saldo desta conta às 14h32 de ontem?" tem resposta determinística e auditável, basta filtrar os lançamentos com timestamp <= '2024-01-15 14:32:00' e somar. Com um campo balance mutável, essa pergunta é impossível de responder com certeza, a menos que você tenha mantido um log separado, que é, ironicamente, um ledger.

A dupla entrada adiciona a segunda camada de garantia: todo débito tem um crédito igual e oposto. Quando o banco transfere R$ 1.000 da conta de Alice para a conta de Bob, dois lançamentos são criados atomicamente: um débito na conta de Alice e um crédito na conta de Bob, ambos com o mesmo valor, o mesmo transaction_id e o mesmo timestamp. A soma de todos os lançamentos no sistema, em qualquer momento, deve ser zero. Se não for zero, dinheiro foi criado ou destruído, e isso é um bug com consequências regulatórias, não uma inconsistência eventual a ser resolvida depois.

No contexto AWS, esse modelo se traduz em uma tabela de lançamentos (journal_entries) append-only no DynamoDB ou Aurora PostgreSQL, com uma view ou query materializada que projeta saldos. A tentação de manter um campo current_balance atualizado a cada transação é compreensível, parece mais performático, mas cria uma fonte de verdade duplicada que, sob falha parcial, diverge silenciosamente.

> A regra de ouro do ledger: Nunca faça UPDATE em saldo. O saldo é uma projeção, a soma dos lançamentos até um instante T. Essa propriedade é o que permite responder 'qual era o saldo às 14h32 de ontem?' com precisão cirúrgica e evidência auditável. Um campo balance mutável é uma mentira conveniente que o regulador, o auditor e o cliente vão cobrar com juros

quando a inconsistência aparecer. Se você precisar de performance de leitura, use uma projeção materializada e incremental, mas mantenha o ledger como única fonte de verdade.

Idempotência: o requisito funcional que a rede esconde

Existe um cenário que todo arquiteto de sistemas financeiros precisa ter gravado na memória muscular: a rede cai depois de o cliente enviar a requisição de pagamento, mas antes de o servidor confirmar o recebimento. O cliente não sabe se o pagamento foi processado. O comportamento correto é reenviar, mas se o sistema não for idempotente, o reenvio gera um segundo pagamento. O dinheiro já saiu da conta na primeira tentativa; agora sai de novo.

Esse não é um cenário de borda. É o comportamento normal de qualquer sistema distribuído sob carga real. Timeouts acontecem. Load balancers reiniciam. Filas de mensagens entregam eventos mais de uma vez, o SQS, por exemplo, garante entrega at-least-once, não exactly-once. Tratar duplicação como exceção é o erro de design mais caro que já vi em sistemas bancários.

Idempotência é um requisito funcional de primeira classe, não um detalhe de implementação. Isso significa que cada operação financeira deve carregar uma chave de idempotência (idempotency_key) gerada pelo cliente, um UUID v4, por exemplo, e o servidor deve garantir que a mesma chave processada duas vezes produz exatamente o mesmo resultado sem efeito colateral adicional. Na prática: a segunda chamada com a mesma chave retorna a resposta da primeira, sem criar um novo lançamento no ledger.

No lado dos consumers de evento, a mesma lógica se aplica: todo consumer que processa mensagens de uma fila SQS, um tópico SNS ou um stream Kinesis deve ser idempotente desde o dia zero. A pergunta de design não é "como evito duplicatas?", mas sim "o que acontece quando processo esta mensagem duas vezes?" Se a resposta for "cria dois lançamentos", o sistema está errado. Se a resposta for "detecta que já processou e retorna sem efeito", o sistema está correto.

A tabela a seguir compara o comportamento esperado de consistência quando o ativo é dinheiro versus outros tipos de dado, e por que as garantias que aceitamos em sistemas de conteúdo são inaceitáveis em sistemas financeiros.

> Replay e duplicação são comportamento normal, não exceção: SQS entrega at-least-once. Kinesis permite replay explícito. EventBridge pode reprocessar eventos em caso de falha de destino. Se o seu consumer não for idempotente, cada um desses mecanismos, que existem para aumentar a resiliência, se torna um vetor de duplicação de pagamentos. Não projete para o caminho feliz e depois tente adicionar idempotência como patch. Projete idempotência primeiro, como restrição de negócio.

Autorização, liquidação e o ponto de conciliação

Um erro arquitetural que vejo com frequência em sistemas bancários construídos às pressas é o acoplamento direto entre autorização e liquidação. O fluxo parece razoável na superfície: o cliente solicita uma transferência, o sistema autoriza, debita e credita na mesma transação de banco de dados, e retorna sucesso. Simples, atômico, correto, até o dia em que você precisa integrar com uma câmara de compensação externa, ou quando a liquidação precisa ocorrer em T+1, ou quando o regulador exige uma trilha de auditoria separada para cada fase.

Autorização e liquidação são eventos distintos no tempo e no significado regulatório. A autorização é a promessa: o banco verificou que os fundos existem, bloqueou o valor e comprometeu-se a liquidar. A liquidação é a transferência efetiva de propriedade. Entre os dois, existe um estado: "autorizado, pendente de liquidação", que precisa ser representado explicitamente no ledger como lançamentos provisionados, não como um flag em uma tabela de

transações.

O ponto de conciliação é o mecanismo que verifica, periodicamente ou em tempo real, que o somatório dos lançamentos autorizados e liquidados está consistente com as posições reportadas pelas contrapartes externas: SPB, câmaras de compensação, custódias. Esse ponto não é um relatório mensal. É um processo contínuo, automatizado, que deve levantar alertas em minutos quando uma divergência é detectada, não em dias.

No elevador da arquitetura, a conversa sobre autorização versus liquidação começa no penthouse: o CFO e o Chief Risk Officer precisam saber que o banco tem exposição durante a janela entre autorização e liquidação, e que essa exposição é mensurável e monitorada. O arquiteto que não sobe para ter essa conversa vai implementar um sistema que funciona em desenvolvimento e cria risco sistêmico em produção. A sala de máquinas, as filas SQS, os streams Kinesis, as lambdas de reconciliação, só faz sentido quando o andar de cima entendeu por que cada peça existe.

Antipadrões que custam dinheiro real

Depois de dezesseis anos trabalhando com sistemas financeiros, posso listar os antipadrões que mais aparecem, e que mais custam, seja em incidentes, em multas regulatórias ou em retrabalho arquitetural:

1. UPDATE direto em saldo. Já discutimos o porquê. O problema prático é que, além de perder auditabilidade, o UPDATE sob concorrência alta exige locks que degradam throughput. O ledger append-only escala horizontalmente com muito mais elegância.

2. Consumir eventos sem idempotência. O consumer processa a mensagem, chama o serviço de pagamento, o serviço retorna timeout, o consumer não faz commit do offset, a mensagem é reentregue, o pagamento é executado duas vezes. Esse bug existe em produção em mais sistemas do que qualquer um gostaria de admitir.

3. Acoplar autorização e liquidação sem ponto de conciliação. O sistema funciona perfeitamente em condições normais. Na primeira integração com uma câmara externa que reporta uma divergência, não existe mecanismo para identificar onde o valor se perdeu. A investigação leva dias; o regulador fica sabendo.

4. Tratar conciliação como relatório mensal. Conciliação é um processo de detecção de falhas. Rodá-la mensalmente é equivalente a verificar os logs de segurança uma vez por mês, quando você descobre o problema, o dano já está feito. Em sistemas financeiros modernos, conciliação deve ser contínua, automatizada e integrada ao pipeline de alertas operacionais.

5. Usar o mesmo banco de dados para ledger e para dados operacionais sem separação de modelo. O ledger tem um modelo de acesso fundamentalmente diferente, append-only, leituras analíticas sobre histórico, enquanto dados operacionais têm leituras e escritas transacionais frequentes. Misturar os dois no mesmo schema cria contenção e dificulta a evolução independente de cada domínio.

Cada um desses antipadrões tem uma versão que parece razoável durante o desenvolvimento e revela seu custo real apenas sob falha parcial, carga alta ou auditoria regulatória. O trabalho do arquiteto é tornar o custo visível antes que ele se materialize.

Princípios inegociáveis do ledger e da idempotência

- Ledger é append-only: INSERT de lançamentos, nunca UPDATE de saldo. O saldo é sempre uma projeção calculada sobre o histórico imutável.

• Dupla entrada garante conservação: a soma de todos os lançamentos deve ser zero. Se não for, dinheiro foi criado ou destruído, isso é um bug, não uma inconsistência aceitável.

• Idempotência é requisito funcional de negócio, não detalhe de engenharia. Toda operação financeira deve ter uma chave de idempotência e o sistema deve garantir que reprocessamento não gera efeito adicional.

• Consumers de evento devem ser idempotentes desde o dia zero. At-least-once delivery é o padrão de filas; duplicação é comportamento normal, não exceção.

• Autorização e liquidação são eventos distintos com representação explícita no ledger. Acoplá-los sem ponto de conciliação cria risco sistêmico invisível.

• Conciliação é detecção contínua de falhas, não relatório periódico. Divergências devem gerar alertas em minutos, não em dias.

Consistência: o que muda quando o ativo é dinheiro

Critério E-commerce / SaaS típico Sistema bancário Por que a diferença importa

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

Modelo de consistência Eventual costuma bastar Forte no core, eventual nas bordas Saldo errado por 2s já é dinheiro indevido

Operação de saldo UPDATE no registro INSERT no ledger (append-only) Auditoria e reconstrução histórica

Duplicar uma mensagem Geralmente tolerável Inaceitável, pagamento duplo Idempotência é requisito, não bônus

Perder uma mensagem Retry e segue Inaceitável, conciliação obrigatória Outbox, DLQ e reprocessamento seguro

Fonte da verdade O banco de dados da app O ledger + conciliação com o regulador Tem que bater com o BACEN, não só interno

O arquiteto entre o penthouse e a sala de máquinas

Existe uma tensão real entre o que o time de produto quer, velocidade, features, time-to-market, e o que o ledger exige, imutabilidade, idempotência, conciliação contínua. Essa tensão não se resolve na sala de máquinas. Ela precisa ser resolvida no penthouse, com a linguagem do risco de negócio.

Quando subo ao penthouse para conversar com o CTO ou com o Chief Risk Officer de um banco, não começo pela arquitetura técnica. Começo pela pergunta: "Se um pagamento for processado duas vezes por falha de rede, quanto tempo leva para detectar? Quem é notificado? Qual é o processo de estorno?" Se a resposta for vaga: "a gente vê no fechamento mensal", então o risco já existe, independentemente de como o sistema foi implementado.

A conversa técnica que se segue, sobre chaves de idempotência, sobre consumers idempotentes, sobre a separação entre autorização e liquidação, só tem peso quando o andar de cima entendeu o custo de não tê-las. O arquiteto que desce direto para a sala de máquinas sem fazer essa tradução vai implementar corretamente e ser ignorado na próxima decisão de priorização.

O ledger de dupla entrada e a idempotência não são abstrações acadêmicas. São os mecanismos pelos quais um banco prova, a qualquer instante, que não criou nem destruiu dinheiro, e que pode demonstrar isso para o BACEN, para o auditor externo e para o cliente que liga perguntando onde foi parar sua transferência. Essa capacidade de prova é o que separa um sistema financeiro de um sistema que processa pagamentos. E construí-la corretamente é, fundamentalmente, responsabilidade do arquiteto que sabe andar de elevador.

Minha opinião direta: ledger e idempotência não são negociáveis: Em dezesseis anos, nunca vi

um sistema financeiro que começou com saldos mutáveis e idempotência fraca e depois corrigiu isso de forma indolor. A migração de um modelo de saldo mutável para um ledger append-only em produção é uma das operações mais arriscadas e caras que um banco pode fazer, e ela sempre acontece depois de um incidente que expôs o problema ao regulador. Minha recomendação é direta: se você está desenhando um novo sistema financeiro, comece pelo ledger correto. Se você está evoluindo um sistema legado, trate a migração para ledger append-only como um projeto de risco regulatório, não como uma refatoração técnica. O custo de fazer certo no início é uma fração do custo de corrigir depois.

Perguntas frequentes sobre ledger e idempotência

Se o saldo é calculado por soma, não fica lento para contas com muitos anos de histórico?

É uma preocupação legítima de performance, não de correção. A solução padrão é um checkpoint periódico: um saldo consolidado até uma data de corte, mais os lançamentos incrementais após esse corte. O checkpoint é um dado derivado, nunca a fonte de verdade, e pode ser invalidado e recalculado a qualquer momento. DynamoDB com streams e Lambda, ou Aurora com views materializadas, implementam esse padrão com eficiência.

Como implementar chaves de idempotência em uma API REST na AWS?

O padrão mais robusto é: o cliente gera um UUID v4 e envia no header Idempotency-Key. O servidor, antes de processar, verifica em uma tabela de idempotência (DynamoDB é ideal pela latência baixa e TTL nativo) se aquela chave já foi processada. Se sim, retorna a resposta armazenada. Se não, processa, armazena a resposta com a chave e retorna. O API Gateway com Lambda pode implementar esse padrão nativamente. O TTL da chave deve ser dimensionado para cobrir a janela máxima de retry do cliente, tipicamente 24 horas para operações financeiras.

A dupla entrada exige que débito e crédito sejam inseridos na mesma transação de banco de dados?

Em um sistema monolítico com um único banco de dados relacional, sim, e isso é o mais simples. Em um sistema distribuído com múltiplos serviços, a atomicidade é garantida por saga com compensação: se o crédito falhar após o débito ser inserido, uma entrada de compensação (estorno do débito) é inserida. O ponto crítico é que o ledger nunca fica em estado inconsistente, ele pode ficar em estado "pendente de compensação", que é explícito e monitorado, mas nunca em estado onde o débito existe sem o crédito correspondente sem registro.

O ledger correto é o fundamento de tudo que vem depois

Os capítulos seguintes vão construir sobre este fundamento: a arquitetura de referência bancária na AWS (Capítulo 07), o tecido nervoso event-driven (Capítulo 08) e os dados como produto com linhagem auditável (Capítulo 09) só fazem sentido completo quando o ledger subjacente é imutável, a idempotência está garantida e a conciliação é contínua. Um sistema financeiro sem esses fundamentos é um sistema que funciona até falhar, e quando falha, falha de formas que o regulador e o cliente não perdoam. O arquiteto que entende isso não está sendo perfeccionista: está sendo preciso sobre onde o risco realmente mora.

PARTE III

A Arquitetura Desce à Sala de Máquinas

A arquitetura de referência de um banco moderno na AWS: core banking, eventos, dados, plataforma de runtime e IA generativa com guardrails, cada peça conectada a uma dor de negócio.

07 - Arquitetura bancária de referência na AWS

Uma visão de referência para sistemas financeiros modernos na AWS: canais e BFFs, domínios em containers e serverless, eventos governados, dados como produto e IA com guardrails, desenhada para FinOps e resiliência.

Uma arquitetura de referência não é um catálogo de serviços AWS com setas entre caixas, é uma declaração de intenção sobre como zonas de responsabilidade se conectam, como o risco flui entre elas e onde o custo se justifica pelo valor entregue. Neste capítulo, percorro a visão de referência que uso como ponto de partida em engajamentos com bancos e fintechs, explicando não apenas o que está no diagrama, mas por que cada peça está lá e o que acontece quando ela não está.

> Por que eu uso uma referência, não uma receita: Depois de dezesseis anos trabalhando em sistemas financeiros, aprendi que o pior erro que um arquiteto pode cometer é entregar um diagrama como se fosse uma verdade universal. A referência que apresento aqui é deliberadamente opinativa: ela reflete decisões sobre onde serverless faz sentido, onde containers são necessários, e como eventos criam fronteiras reversíveis entre domínios. Você vai discordar de algumas escolhas, e esse é exatamente o ponto. Uma boa referência provoca a conversa certa, não elimina a necessidade de pensar.

As zonas do diagrama: borda, domínios, eventos, dados e operação

O diagrama que acompanha este capítulo divide a plataforma em cinco zonas funcionais. Não são camadas empilhadas, são regiões com responsabilidades distintas que se comunicam por contratos explícitos.

Zona de borda e identidade. Todo tráfego externo entra pelo CloudFront, que entrega proteção de borda, cache e terminação TLS antes de qualquer lógica de aplicação. O WAF senta imediatamente atrás, aplicando regras gerenciadas e customizadas que o time de segurança pode evoluir sem tocar no código de negócio. O Cognito e o IAM Identity Center resolvem dois problemas diferentes: o primeiro cuida de identidades de clientes finais (autenticação federada, MFA, tokens OIDC); o segundo gerencia identidades internas e acesso a consoles e APIs administrativas. O API Gateway é a fronteira entre o mundo externo e os BFFs: Backend for Frontend, que traduzem a linguagem de cada canal (mobile, internet banking, open finance) para os contratos internos dos domínios. Essa separação importa: quando o regulador exige um novo campo no extrato do open finance, a mudança fica contida no BFF de open finance, não se propaga para o domínio de conta.

Zona de domínios. Aqui vivem as capacidades de negócio descritas no Capítulo 04. Domínios com estado complexo, ciclos de vida longos ou requisitos de controle de runtime, como o motor de crédito ou o processador de pagamentos, rodam em EKS ou ECS. Domínios orientados a eventos discretos, sem estado persistente entre invocações, usam Lambda e Step Functions. Não é uma escolha religiosa: é uma decisão de modelo operacional que discuto em detalhe no Capítulo 10. O que importa aqui é que ambos os modelos coexistem na mesma plataforma e compartilham os mesmos contratos de evento.

Eventos, dados e IA: as zonas que sustentam a inteligência da plataforma

Zona de eventos. O MSK (Kafka gerenciado) é o backbone para fluxos de alta vazão e retenção, transações, posições, alertas de risco. O EventBridge complementa com roteamento baseado em schema para eventos de negócio de menor volume mas alta semântica, como aprovação de crédito ou onboarding concluído. O SQS aparece nos pontos onde a semântica de fila, entrega garantida, dead-letter, visibilidade controlada, é mais importante do que o modelo pub/sub. Esses três serviços não são intercambiáveis: cada um resolve uma classe de problema diferente, e misturá-los sem critério gera complexidade operacional sem benefício. No Capítulo 08, aprofundo o raciocínio sobre quando usar cada um.

Zona de dados. Aurora PostgreSQL serve os domínios transacionais que precisam de ACID e SQL expressivo, o ledger, a posição de carteira, o cadastro de clientes. DynamoDB serve os domínios que precisam de latência sub-milissegundo e escala horizontal previsível, sessões, cache de produto, preferências. O S3 com Glue e Lake Formation forma o data lake governado, onde os dados de produto (Capítulo 09) são catalogados, versionados e disponibilizados com controle de acesso por coluna e linha. Nenhum dado sai dessa zona sem passar pelo Lake Formation, essa é a fronteira de governança que o auditor vai perguntar sobre.

Zona de IA. O Bedrock com Guardrails não é um ornamento, é a camada que permite que modelos de linguagem sejam usados em fluxos regulados sem abrir mão de auditabilidade. O Capítulo 11 trata disso em profundidade. O que importa na visão de referência é que a IA está dentro da plataforma, não colada por fora: ela consome eventos, escreve em dados de produto e é governada pelos mesmos controles de identidade e acesso do restante da arquitetura.

Zona de operação. CloudWatch e OpenTelemetry formam o plano de observabilidade. Security Hub agrega findings. GuardDuty detecta comportamento anômalo. Config registra toda mudança de configuração de recurso. KMS gerencia chaves com rotação auditável. Essa zona não é opcional em ambiente regulado, ela é a evidência que o Capítulo 12 vai exigir.

Plataforma bancária moderna na AWS, visão de referência

Não é um catálogo de serviços: é um modelo de como as zonas se conectam. Canais na borda, domínios isolados, eventos como contrato entre eles, dados governados e IA tratada como capability com fronteiras.

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)

Fluxos

- cliente -> waf: HTTPS
- waf -> apigw
- apigw -> cognito: authz
- apigw -> eks: síncrono
- eks -> eb: publica eventos
- eb -> lambda: dispara
- eks -> msk: alto volume
- lambda -> sqs
- eks -> aurora
- lambda -> dynamo
- msk -> lake: ingestão
- lambda -> bedrock: RAG / agentes
- bedrock -> lake: Knowledge Base

O que o diagrama não mostra, e você precisa saber

- Fronteiras de conta AWS (multi-account) não aparecem no diagrama por clareza visual, mas são decisão de dia zero: cada domínio de negócio em conta separada é a posição defensiva padrão em ambientes regulados.
- O diagrama não mostra latências nem SLOs, esses números dependem do produto, não da plataforma, e devem ser definidos antes de qualquer escolha de serviço.
- Nenhuma seta no diagrama representa uma chamada síncrona entre domínios distintos. Se você precisar desenhar uma, é um sinal de que a fronteira de domínio está errada.
- O diagrama é agnóstico a região AWS, mas a escolha de região tem implicações de soberania de dados que o BACEN e a LGPD tornam não-triviais.
- A zona de operação não é um add-on de fase dois, ela deve ser provisionada antes do primeiro serviço de negócio, não depois.

FinOps como princípio de desenho, não como otimização posterior

O primeiro princípio que sustenta este desenho é FinOps, e preciso ser preciso sobre o que isso significa aqui. Não estou falando de dashboards de custo ou de Reserved Instances. Estou falando de uma decisão arquitetural: custo por transação é uma métrica de produto, não de infraestrutura.

Quando um domínio de pagamentos processa uma transferência, o custo daquela operação,

computação, armazenamento, transferência de dados, chamadas de API, deve ser mensurável e atribuível. Isso tem duas consequências práticas. Primeira: cada peça da arquitetura precisa justificar seu custo operacional em relação ao valor que entrega. Um cluster EKS rodando um serviço que processa duzentas requisições por dia é um problema de desenho, não de otimização. Segunda: a escolha entre serverless e container não é estética, é financeira e operacional.

A posição que adoto nesta referência é: serverless e pay-per-use onde o modelo de execução for compatível; container onde o domínio exige controle de runtime, estado persistente ou características de performance que o modelo serverless não consegue garantir de forma econômica. Lambda com arquitetura event-driven tem custo marginal próximo de zero em períodos de baixa demanda, algo que um cluster EKS nunca vai conseguir, mesmo com Karpenter e escalonamento agressivo. Por outro lado, um motor de precificação de derivativos que precisa de CPU dedicada, memória previsível e latência de cold-start zero não pertence ao Lambda.

A consequência prática para o arquiteto que sobe ao andar executivo: quando o CFO perguntar por que a conta AWS cresceu trinta por cento no último trimestre (e ele vai perguntar), você precisa conseguir responder em termos de volume de transações, não de horas de instância. Essa é a conversa que a referência torna possível, porque o custo está distribuído por domínio, e cada domínio mapeia para uma capacidade de negócio com receita associada.

Reversibilidade: a arquitetura que mantém opções abertas

O segundo princípio é reversibilidade, e aqui o elevador de arquitetura aparece de forma mais explícita. No andar executivo, o negócio precisa de agilidade: lançar novos produtos, responder a mudanças regulatórias, integrar parceiros. Na sala de máquinas, o time de engenharia precisa de autonomia: evoluir um domínio sem coordenar com todos os outros, trocar uma implementação sem reescrever integrações. Esses dois imperativos são o mesmo imperativo visto de andares diferentes, e eventos são o mecanismo que os reconcilia.

Quando dois domínios se comunicam por evento, o contrato entre eles é o schema do evento, não a implementação interna de nenhum dos dois. O domínio de conta publica um evento ContaAtualizada com um schema versionado. O domínio de notificação consome esse evento. Se amanhã o domínio de conta migrar de Aurora para DynamoDB, ou de ECS para Lambda, o domínio de notificação não sabe e não precisa saber. Essa é a reversibilidade que o diagrama materializa: eventos como contratos entre domínios permitem trocar a implementação de um domínio sem reescrever os outros.

A implicação prática é que a arquitetura boa não tenta prever o futuro, ela mantém o máximo de opções abertas pelo menor custo de manutenção. Isso tem um corolário importante: acoplamento síncrono entre domínios é uma dívida técnica com juros compostos. Cada chamada REST direta entre domínios distintos é uma dependência que vai custar caro quando um dos dois precisar evoluir. Não estou dizendo que chamadas síncronas são sempre erradas, estou dizendo que elas precisam ser uma decisão consciente, com o custo de reversibilidade explicitamente aceito.

No contexto bancário brasileiro, isso tem uma dimensão regulatória adicional. O BACEN evolui normas com frequência: PIX, open finance, DREX. Cada nova regulação é um vetor de mudança que vai atingir domínios específicos. Uma arquitetura com fronteiras reversíveis absorve essas mudanças de forma localizada. Uma arquitetura fortemente acoplada transforma cada nova norma em um projeto de seis meses com risco de regressão em produção. Como mostra o diagrama a seguir, as zonas de referência são desenhadas exatamente para que esse isolamento seja possível.

Como ler o diagrama de referência em um engajamento real

1. Identifique as capacidades de negócio presentes: Antes de olhar para qualquer serviço AWS, mapeie quais capacidades do Capítulo 04 o banco já tem e quais está construindo. Cada capacidade vai para uma zona de domínio, não para um serviço específico.

2. Marque os fluxos de risco regulatório: Identifique quais fluxos de dados cruzam fronteiras regulatórias, dados de cliente, dados de transação, dados de posição. Esses fluxos determinam onde Lake Formation, KMS e Config são obrigatórios, não opcionais.

3. Valide os contratos de evento entre domínios: Para cada seta no diagrama que cruza uma fronteira de domínio, pergunte: qual é o schema do evento? Quem é o produtor canônico? Qual é a política de versionamento? Se não houver resposta, a fronteira ainda não está pronta.

4. Aplique os pilares do Well-Architected como checklist: O bloco Well-Architected que acompanha este capítulo lê cada zona da referência pelos seis pilares. Use-o como roteiro de revisão antes de qualquer decisão de go-live.

> A referência não substitui o contexto do seu banco: Toda vez que apresento esta referência, alguém pergunta se pode simplesmente adotá-la como está. A resposta é não. A referência é um ponto de partida para a conversa certa, sobre fronteiras de domínio, sobre modelo operacional, sobre tolerância a risco. Um banco de atacado com dez mil transações por dia tem restrições de custo e complexidade diferentes de uma fintech de varejo com dez milhões. A referência funciona para os dois, mas as escolhas dentro dela serão diferentes. O trabalho do arquiteto é justamente fazer essas escolhas com evidência, não por analogia.

Perguntas frequentes sobre a arquitetura de referência

Por que MSK e não Kinesis para o backbone de eventos?

Kinesis é uma escolha válida para fluxos de menor complexidade operacional. MSK (Kafka) é preferível quando o banco já tem competência em Kafka, quando precisa de retenção longa com replay granular, ou quando quer portabilidade de workload entre cloud e on-premises. A decisão deve ser baseada no modelo operacional do time, não na familiaridade do arquiteto com um dos dois.

EKS ou ECS para os domínios containerizados?

ECS tem menor overhead operacional e é suficiente para a maioria dos casos. EKS faz sentido quando o banco precisa de portabilidade entre clouds, quando já tem investimento em ferramentas do ecossistema Kubernetes, ou quando os times de plataforma têm maturidade para operar o plano de controle. Não escolha EKS porque é mais sofisticado, escolha porque o modelo operacional do seu time justifica a complexidade adicional.

Como a referência se relaciona com os requisitos de resiliência do BACEN?

A Resolução BCB nº 85 e as normas de continuidade de negócio do BACEN exigem RTO e RPO documentados para sistemas críticos. A referência suporta isso através de multi-AZ nativo nos serviços gerenciados, replicação de eventos no MSK e estratégias de backup no Aurora e S3. Mas a referência não define os números, isso é responsabilidade do processo de gestão de risco do banco, não da arquitetura técnica.

Lendo a referência pelos pilares do Well-Architected

- security: Identidade contextual, least privilege, criptografia com KMS, trilha imutável com

CloudTrail e segregação por conta/OU. Segurança é evidência, não opinião.

• reliability: Domínios isolados, filas com DLQ, idempotência de ponta a ponta, multi-AZ por padrão e plano de recuperação testado por jornada, não genérico.

• cost: Serverless onde o tráfego é irregular, containers onde há escala constante, e custo por transação como métrica de produto observada continuamente.

• operational-excellence: Golden paths, observabilidade desde o dia um, runbooks acionáveis e mecanismos que tornam o bom caminho o caminho mais fácil.

• performance: Aurora para consistência relacional, DynamoDB para baixa latência, cache e CQRS onde leitura e escrita têm perfis diferentes.

• sustainability: Pay-per-use reduz capacidade ociosa; right-sizing contínuo e arquitetura event-driven evitam polling e desperdício de cômputo.

Lendo a referência pelos pilares: o que vem a seguir

Uma visão de referência percorrida zona por zona é necessária, mas não suficiente. O arquiteto que sobe ao andar do comitê de risco precisa conseguir responder perguntas que não estão no diagrama: o que acontece quando o MSK fica indisponível? Quem tem acesso aos dados de cliente no data lake e como isso é auditado? Qual é o custo de uma transação de PIX nesta arquitetura em escala de dez milhões de operações por dia?

Essas perguntas são respondidas quando você lê a referência pelos seis pilares do AWS Well-Architected Framework, excelência operacional, segurança, confiabilidade, eficiência de performance, otimização de custos e sustentabilidade. O bloco que segue este texto faz exatamente isso: percorre cada zona da referência e aponta onde cada pilar se manifesta, onde há tensões entre pilares e quais são as perguntas que o arquiteto deve ser capaz de responder antes de considerar a arquitetura pronta.

O que quero deixar claro antes de você ler aquele bloco é o seguinte: os pilares não são uma checklist de conformidade. Eles são um vocabulário compartilhado para ter conversas difíceis, sobre trade-offs entre custo e confiabilidade, entre velocidade de entrega e segurança, entre autonomia de domínio e governança centralizada. Usar o Well-Architected como ferramenta de revisão é uma das formas mais eficientes que conheço de subir do diagrama técnico para a conversa de risco de negócio sem perder precisão no caminho. Os capítulos seguintes desta Parte III vão descer ainda mais fundo em cada uma dessas dimensões, eventos no Capítulo 08, dados no Capítulo 09, plataforma no Capítulo 10. A referência apresentada aqui é o mapa; os próximos capítulos são o território.

O que esta referência entrega, e o que não entrega

Esta arquitetura de referência entrega um modelo de como zonas de responsabilidade se conectam em uma plataforma bancária moderna na AWS, dois princípios explícitos de desenho (FinOps e reversibilidade), e um vocabulário compartilhado para conversas entre o andar executivo e a sala de máquinas. Ela não entrega um design pronto para produção, não define SLOs, não substitui o processo de gestão de risco do banco e não é válida sem adaptação ao contexto específico de cada instituição. Use-a como ponto de partida para as perguntas certas, não como resposta para todas elas.

08 - Event-driven como tecido nervoso do banco

Bancos sempre foram event-driven, muito antes do termo virar moda. A questão é se os fatos de negócio ficam escondidos em sistemas acoplados ou viram contratos explícitos, com schema, versionamento e ownership.

Bancos sempre foram event-driven, muito antes do termo virar capa de conferência. Uma transação liquidada, uma proposta que muda de estado, um contrato assinado, um limite recalculado, uma fraude suspeita, um consentimento revogado: todos são fatos de negócio que ocorreram no mundo real e que o sistema precisa registrar, propagar e honrar. A pergunta de arquitetura nunca foi 'devemos usar eventos?', foi, e continua sendo, 'esses fatos são contratos explícitos com schema, dono e rastreabilidade, ou estão presos dentro de chamadas síncronas acopladas que ninguém mais consegue evoluir sem medo?'

> Minha visão depois de 16 anos em sistemas financeiros: Toda vez que entro em um banco e vejo um serviço de pagamento chamando seis outros serviços de forma síncrona, e cada um desses chamando mais três, reconheço o padrão imediatamente: alguém tentou 'desacoplar' sem entender que desacoplamento real exige que o fato de negócio seja um cidadão de primeira classe. O resultado é o pior dos mundos: a fragilidade de um sistema distribuído com o acoplamento temporal de um monólito. Event-driven feito certo não é sobre tecnologia de mensageria; é sobre tornar os fatos do domínio explícitos, versionados e auditáveis. Quando subo ao penthouse para conversar com o Chief Risk Officer sobre exposição a fraude em tempo real, e depois desço à sala de máquinas para revisar o design de um tópico Kafka, estou fazendo exatamente o mesmo trabalho, traduzindo o mesmo fato de negócio entre dois vocabulários. O elevador não para no meio.

Fatos de negócio são contratos, não notificações

Existe uma distinção que separa arquiteturas bancárias maduras das que viram passivo operacional em dois anos: a diferença entre um evento como notificação e um evento como contrato de domínio.

Uma notificação diz: "algo aconteceu, faça o que quiser com isso." Um contrato de domínio diz: "o fato TransacaoLiquidada ocorreu às 14:23:07 UTC, com estes campos obrigatórios, neste schema versionado v2.1, emitido pelo domínio de Liquidação, e qualquer consumidor pode depender desse contrato sem consultar o emissor." A diferença não é filosófica, é operacional. Quando o BACEN exige rastreabilidade de uma operação de câmbio, você não quer responder "o evento foi publicado, mas não temos schema registrado e o consumidor pode ter interpretado o campo valorBruto de forma diferente do emissor."

Um evento de domínio bancário bem formado carrega, no mínimo: identidade única e imutável do fato (eventId), timestamp de ocorrência no domínio (não de publicação na fila), identificador de correlação para rastreamento cross-sistema, versão do schema, e o dono do domínio emissor. Esses campos não são burocracia, são os atributos que permitem reprocessamento seguro, auditoria regulatória e evolução independente de consumidores.

A governança começa no schema. Um Schema Registry, seja o Confluent Schema Registry sobre MSK, seja o AWS Glue Schema Registry, não é uma ferramenta de infraestrutura; é onde o contrato entre domínios fica escrito e versionado. Quando o time de Crédito evolui o evento LimiteRecalculado para incluir o campo scoreModelVersion, o Schema Registry garante que consumidores antigos continuam funcionando (compatibilidade backward) enquanto novos consumidores podem optar pelo campo adicional. Sem isso, uma mudança de schema em produção é uma mudança de contrato não anunciada, e em um banco, contratos não anunciados viram incidentes.

Escolhendo o backbone certo: MSK, EventBridge ou SQS

A pergunta "qual serviço de mensageria devo usar?" é, na prática, a pergunta errada. A pergunta certa é: "qual é o modelo de consumo, o requisito de retenção, o padrão de acoplamento e o nível de governança que esse fato de negócio exige?"

No contexto bancário, três serviços da AWS dominam o backbone de eventos, e cada um tem um papel distinto. O Amazon MSK (Kafka gerenciado) é o tecido nervoso para eventos de domínio de alta frequência e alta criticidade, liquidações, movimentos de ledger, eventos de fraude em tempo real, onde retenção configurável, replay determinístico e consumer groups com offsets explícitos são requisitos, não diferenciais. O custo operacional é real: MSK exige sizing cuidadoso de brokers, gestão de partições e monitoramento de consumer lag. Esse custo se justifica quando o volume, a latência e a necessidade de replay fazem de qualquer alternativa uma concessão arquitetural.

O Amazon EventBridge brilha em integrações entre domínios e entre contas AWS, o barramento de eventos onde um domínio publica sem saber quem vai consumir, e onde regras de roteamento por schema permitem que novos consumidores se conectem sem tocar no emissor. Para um banco que está construindo uma arquitetura de plataforma com múltiplos times de produto, o EventBridge é o mecanismo de extensibilidade: o domínio de Onboarding publica ClienteAprovado e os domínios de Cartão, Conta e CRM se inscrevem de forma independente. A limitação é o throughput e a ausência de replay nativo com a semântica de offsets do Kafka, para eventos de altíssimo volume, EventBridge não é o lugar certo.

O Amazon SQS resolve um problema diferente: entrega confiável ponto-a-ponto com DLQ nativa, visibilidade de mensagem e integração trivial com Lambda. Para comandos assíncronos: "processar esta solicitação de portabilidade", "enviar este comprovante": SQS é frequentemente a escolha mais simples e mais operacionalmente segura. A tabela a seguir compara os três serviços nos critérios que mais importam em um banco.

Escolhendo o backbone de eventos na AWS

Critério Amazon MSK / Kafka EventBridge SQS

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

Melhor para Alto volume, ordenação, replay longo Eventos de domínio, fan-out, regras, SaaS Desacople ponto-a-ponto, absorção de picos

Ordenação Por partição Sem garantia forte FIFO opcional

Retenção / replay Dias a meses, replay nativo Curta (archive/replay limitado) Até processar (+ DLQ)

Custo operacional Maior, cluster para operar Baixo, serverless Muito baixo, serverless

Em banco, use para Stream de transações, ledger feeds Eventos entre domínios, orquestração Buffers de carga, integração com legado

> O acoplamento distribuído é pior que o monólito: Quando um serviço publica um evento sem schema registrado, sem DLQ configurada, sem idempotência no consumidor e sem dono declarado, o resultado não é um sistema desacoplado, é um monólito distribuído onde os contratos implícitos estão escondidos em código de consumidor que ninguém mais lembra quem escreveu. Em um banco, isso tem consequência regulatória direta: se você não consegue provar o que aconteceu com um fato de negócio específico, você não tem auditoria, você tem esperança.

O padrão Transactional Outbox, eliminando a dupla escrita

Este é o ponto onde a maioria dos times erra, e onde o erro tem consequência financeira direta. O

problema é simples de enunciar e difícil de resistir: quando um serviço precisa gravar estado no banco de dados e publicar um evento, a solução ingênua é fazer as duas operações em sequência, primeiro INSERT no banco, depois publish no Kafka. O problema é que entre essas duas operações existe uma janela de falha. Se o processo cair após o INSERT e antes do publish, o estado foi gravado mas o evento nunca foi emitido. Um pagamento aconteceu no ledger, mas nenhum consumidor downstream soube, o saldo foi debitado, a notificação nunca chegou, o sistema de conciliação nunca registrou. O inverso também é possível: o evento é publicado, mas a transação no banco falha ou é revertida. Agora consumidores downstream reagiram a um fato que não existe.

O padrão Transactional Outbox resolve isso com uma garantia fundamental: estado e evento são gravados na mesma transação local do banco de dados. O serviço não publica diretamente no broker, ele escreve em uma tabela outbox dentro da mesma transação ACID que grava o estado de negócio. Um processo separado (o relay ou CDC connector) lê essa tabela e publica no broker com semântica at-least-once. Se o relay falhar, ele simplesmente relê a tabela e republica, o evento pode chegar mais de uma vez, mas nunca deixa de chegar.

A consequência imediata é que consumidores precisam ser idempotentes: processar o mesmo evento duas vezes deve produzir o mesmo resultado que processá-lo uma vez. Isso não é uma limitação do padrão; é uma propriedade que todo consumidor de evento bancário deveria ter independentemente, porque redes falham, brokers reentregam e sistemas reiniciam. A combinação Outbox + consumidor idempotente entrega a garantia que sistemas financeiros exigem: nenhum fato se perde e nenhum fato se duplica em efeito.

Na AWS, o relay pode ser implementado com Debezium sobre Amazon MSK Connect lendo o binlog do RDS PostgreSQL via CDC, ou com um processo Lambda agendado que faz polling da tabela outbox, a segunda opção é mais simples operacionalmente mas introduz latência proporcional ao intervalo de polling. Para eventos de liquidação onde latência importa, CDC é a escolha certa. Para notificações assíncronas de menor criticidade, polling com Lambda é aceitável e mais fácil de operar.

Implementando o Transactional Outbox em um banco na AWS

1. Criar a tabela outbox na mesma instância RDS do domínio: A tabela domain_outbox deve ter: event_id UUID PRIMARY KEY, aggregate_id, event_type, schema_version, payload JSONB, created_at, published_at (nullable). O índice em published_at IS NULL é o que o relay usa para encontrar eventos pendentes.

2. Gravar estado de negócio e evento na mesma transação: No código do serviço, dentro do mesmo BEGIN/COMMIT: UPDATE accounts SET balance = ... e INSERT INTO domain_outbox (event_id, event_type, payload) VALUES (...). Se a transação fizer rollback por qualquer motivo, o evento também some, nunca há divergência.

3. Configurar o relay com CDC (Debezium + MSK Connect): O conector Debezium monitora o binlog do RDS PostgreSQL e publica cada INSERT na tabela outbox como mensagem no tópico MSK correspondente. A chave da mensagem deve ser o aggregate_id para garantir ordering por agregado. Habilitar logical replication no RDS e configurar o slot de replicação com retenção adequada.

4. Garantir idempotência no consumidor com deduplication store: O consumidor deve verificar o event_id em um store de deduplicação (DynamoDB com TTL de 7 dias é uma escolha comum) antes de processar. Se o event_id já existe, o evento é descartado silenciosamente e o offset é commitado. Se não existe, processa e grava o event_id no store na mesma operação lógica.

5. Monitorar consumer lag e DLQ como SLA operacional: Consumer lag no MSK é um indicador de saúde do fluxo de eventos, não apenas de performance. Um lag crescente em um tópico de liquidação pode significar que um serviço downstream está falhando silenciosamente. Alarmes no CloudWatch para lag > threshold e para qualquer mensagem na DLQ devem ser tratados com a mesma urgência que um alarme de erro 5xx.

Governança de eventos: dono, schema e ciclo de vida

Tecnologia sem governança é automação do caos. Em uma arquitetura event-driven bancária, governança significa responder a três perguntas para cada evento de domínio: quem é o dono, qual é o contrato e qual é o ciclo de vida.

O dono não é o time que criou o tópico Kafka, é o domínio de negócio responsável pela semântica do evento. O domínio de Crédito é dono de LimiteRecalculado; o domínio de Pagamentos é dono de TransacaoLiquidada. Essa distinção importa quando há um incidente: se o schema mudou de forma incompatível e consumidores quebraram, o dono é quem tem que responder, não a equipe de infraestrutura. Sem ownership declarado, toda mudança de schema vira uma negociação política entre times.

O contrato é o schema versionado registrado no Schema Registry, com política de compatibilidade explícita. Para eventos bancários, recomendo BACKWARD_TRANSITIVE como padrão: qualquer versão nova pode ser lida por consumidores de qualquer versão anterior. Isso permite que o emissor evolua sem coordenação com cada consumidor, que é exatamente o desacoplamento que justifica a arquitetura event-driven. Campos novos são opcionais; campos existentes nunca mudam de tipo; campos removidos passam por um período de deprecação com versão de sunset anunciada.

O ciclo de vida define por quanto tempo o evento existe, quem pode consumir em replay e o que acontece quando o contrato precisa ser quebrado (versão major). Em um banco, eventos de liquidação têm requisito de retenção regulatória, o BACEN pode exigir rastreabilidade de operações por anos. Isso significa que a retenção do tópico MSK precisa ser alinhada com a política de retenção de dados da instituição, e que replay de eventos históricos é um caso de uso legítimo que a arquitetura deve suportar, não um acidente que ela tolera.

Quando subo ao penthouse e o Chief Compliance Officer pergunta "conseguimos provar o que aconteceu com essa operação de câmbio em março do ano passado?", a resposta depende inteiramente de decisões que foram tomadas na sala de máquinas meses antes: o evento foi gravado com timestamp de ocorrência no domínio, o schema estava registrado, a retenção foi configurada corretamente, e o relay nunca perdeu uma mensagem. Governança de eventos não é uma preocupação de infraestrutura, é uma preocupação de compliance que se manifesta em infraestrutura.

Regras de ouro para eventos de domínio bancário

- Um evento de domínio é um fato imutável, ele descreve o que aconteceu, nunca uma instrução do que fazer. Nomeie no passado: TransacaoLiquidada, não LiquidarTransacao.
- Nunca faça dupla escrita (banco + broker em operações separadas). Use Transactional Outbox para garantir que estado e evento são atômicos na origem.
- Todo evento precisa de: eventId único, timestamp de ocorrência no domínio, correlationId, versão de schema e domínio emissor. Esses campos não são opcionais em sistemas financeiros.
- Todo consumidor de evento bancário deve ser idempotente. At-least-once delivery é a garantia

real de qualquer sistema distribuído, projete para isso, não contra isso.

• DLQ não é lixeira, é um contrato de observabilidade. Qualquer mensagem em DLQ é um fato de negócio que não foi processado. Trate com a mesma urgência de um erro de produção.

• Schema Registry com política BACKWARD_TRANSITIVE é o mecanismo que permite evolução independente de emissores e consumidores. Sem ele, cada mudança de schema é uma janela de coordenação, e coordenação em escala é o inimigo da velocidade.

Event-driven bancário: o veredicto do arquiteto

Event-driven em bancos não é uma escolha de modernização, é o reconhecimento tardio de que fatos de negócio sempre existiram e sempre precisaram ser propagados. A escolha real é entre propagar esses fatos de forma explícita, governada e auditável, ou continuar escondendo-os em chamadas síncronas que ninguém mais consegue evoluir. O Transactional Outbox, o Schema Registry, o ownership de domínio e a idempotência de consumidores não são detalhes de implementação, são os pilares que separam uma arquitetura event-driven que o regulador pode auditar de uma que o arquiteto precisa explicar por que falhou. Escolha os pilares antes de escolher o broker.

09 - Dados como produto, linhagem e prova

Em banco, dado não é subproduto: é evidência, obrigação regulatória, insumo de risco e base para IA. Arquitetura madura não pergunta só onde armazenar, pergunta quem é dono, qual a linhagem e qual a obrigação.

Em banco, dado não é subproduto de sistema: é evidência regulatória, insumo de risco, base de personalização e, quando mal governado, passivo que espera apenas o momento certo para se materializar em multa, fraude ou falha de modelo. A arquitetura madura de dados não começa escolhendo entre Redshift e Athena, começa numa pergunta de domínio que a maioria dos times nunca faz em voz alta: este dado representa que fato, para quem, com qual obrigação de retenção e qual risco se vazar?

> A pergunta que separa arquitetura de dados de engenharia de pipeline: Depois de dezesseis anos construindo plataformas em instituições financeiras, aprendi que o maior erro não é escolher a tecnologia errada, é começar pela tecnologia. Já vi lakes petabyte que ninguém confiava, feature stores sem dono e dashboards de risco que contradiziam o sistema de registro. O problema invariavelmente era o mesmo: os dados chegaram ao lake como despejo, não como produto. Ninguém havia respondido quem é o produtor responsável, qual o contrato de qualidade, qual a linhagem até a fonte e qual a obrigação regulatória que esse dado carrega. Quando o arquiteto sobe ao penthouse e ouve 'queremos IA para personalização', a resposta técnica correta não é provisionar um endpoint de SageMaker, é perguntar se o consentimento LGPD está segregado, se a linhagem do dado de comportamento está auditável e se a qualidade está contratualizada. Só depois disso a conversa sobre modelo faz sentido.

O dado bancário tem identidade antes de ter endereço

Considere três objetos de dado que coexistem em qualquer banco brasileiro de médio porte: uma transação Pix, um score de crédito e um registro de consentimento LGPD. Tecnicamente, os três são linhas em algum banco de dados. Arquiteturalmente, são objetos radicalmente diferentes.

A transação Pix é um fato imutável de negócio: ocorreu num instante, tem valor jurídico, precisa

ser retida por cinco anos conforme Resolução BCB nº 1 e suas derivações, e qualquer alteração posterior é fraude, não correção. O score de crédito é um dado derivado e temporal: é o resultado de um modelo aplicado sobre sinais em determinado momento, expira, e sua linhagem precisa ser auditável porque o BACEN pode perguntar, numa supervisão, por que aquele cliente foi negado naquela data. O consentimento LGPD é um dado de controle de acesso: ele não descreve o cliente, ele autoriza ou proíbe que outros dados do cliente sejam usados para determinada finalidade, e sua ausência deve bloquear pipelines inteiros.

Quando o arquiteto não distingue esses três objetos antes de projetar o lake, o resultado é previsível: a transação Pix acaba sendo sobrescrita por um job de ETL mal escrito, o score de crédito perde a linhagem depois de uma migração de schema, e o dado de comportamento de navegação flui para um modelo de personalização sem verificar se o consentimento para aquela finalidade existe. Cada um desses erros tem nome no vocabulário regulatório: adulteração de registro, impossibilidade de explicabilidade de decisão automatizada e uso indevido de dado pessoal.

A pergunta de domínio: que fato, para quem, com qual obrigação, não é filosófica. É o primeiro controle de risco da arquitetura de dados.

Dado como produto: do lake como lixão ao marketplace governado

O conceito de data product: dado como produto, resolve o problema de governança onde frameworks puramente técnicos falham: no incentivo. Quando um time de engenharia entrega dados ao lake sem contrato, sem SLA e sem dono declarado, o consumidor downstream assume o risco de qualidade silenciosamente. Ninguém é responsabilizado quando o dado chega corrompido, atrasado ou sem documentação. O lake vira depósito de despejo porque o modelo de responsabilidade permite isso.

Aplicar a lógica de produto inverte esse incentivo. Cada domínio, crédito, pagamentos, onboarding, antifraude: publica seus dados como um produto com contrato explícito: schema versionado, SLA de atualização, definição de qualidade mensurável, dono nomeado e política de acesso. O consumidor subscreve ao produto, não ao bucket. Se a qualidade cai abaixo do contrato, o produtor é notificado e responsabilizado, exatamente como acontece com uma API de serviço.

Na prática, isso muda três coisas na arquitetura AWS. Primeiro, o AWS Glue Data Catalog deixa de ser apenas um catálogo técnico de schemas e passa a ser o registro de produtos de dados: cada tabela tem metadados de negócio, dono, classificação de sensibilidade e SLA. Segundo, o Lake Formation implementa o controle de acesso baseado em atributo (TBAC/ABAC) que respeita os contratos de acesso, o time de marketing não acessa dados de score de crédito sem aprovação explícita do domínio de risco. Terceiro, os jobs de AWS Glue que alimentam as camadas do medalhão (Bronze -> Silver -> Gold) passam a incluir validações de qualidade como etapa obrigatória, não opcional, um dado que não passa na validação não avança de camada.

O diagrama a seguir mostra como essas peças se articulam na plataforma de dados governada que uso como referência para instituições financeiras na AWS.

O que define um data product bancário maduro

- Schema versionado e publicado no Glue Catalog com metadados de negócio, não apenas técnicos
- Dono de domínio nomeado, pessoa física responsável pela qualidade e pelo SLA, não um time genérico
- Classificação de sensibilidade declarada: dado pessoal, dado financeiro, dado operacional,

dado público

- Política de acesso baseada em finalidade (purpose-based), não apenas em perfil técnico
- Linhagem rastreável da fonte até o consumidor final, auditável por regulador
- SLA de qualidade mensurável com alertas automáticos, não promessa verbal de 'dados confiáveis'

Linhagem e prova: o dado que não tem história não tem valor em auditoria

Linhagem de dados é o registro de onde um dado veio, por quais transformações passou e quem o consumiu. Em qualquer setor, isso é boa prática. Em banco, é obrigação. Quando o BACEN questiona uma decisão de crédito automatizada, a resposta não pode ser 'o modelo disse não', precisa ser 'o modelo versão 2.3, treinado com dados do período X a Y, recebeu estes sinais de entrada, produziu este score, e a decisão foi tomada com base nesta política de corte vigente naquela data'. Cada elo dessa cadeia precisa ser rastreável.

Na AWS, a linhagem pode ser implementada em camadas. No nível de infraestrutura, o AWS Glue registra automaticamente as transformações de job e o Amazon S3 com versionamento ativado preserva cada estado dos objetos. No nível de catálogo, o Glue Data Catalog com integração ao Apache Atlas ou soluções como o Amazon DataZone permite mapear dependências entre tabelas e jobs. No nível de modelo, o Amazon SageMaker Experiments e o SageMaker Model Registry registram qual dataset treinou qual versão de modelo, o que é indispensável para o artigo 20 da LGPD (direito à explicação de decisão automatizada).

Mas linhagem técnica sem governança de processo é incompleta. Já participei de auditorias onde a linhagem técnica estava perfeita no papel, os jobs estavam documentados, os schemas versionados, mas ninguém sabia dizer se o dado de entrada havia passado por um processo de anonimização antes de alimentar o modelo, porque esse passo estava num script manual fora do pipeline oficial. A linhagem precisa capturar todo o caminho, incluindo os atalhos que os times criam sob pressão de entrega.

O arquiteto que sobe ao penthouse e ouve 'precisamos explicar nossas decisões de crédito ao regulador' precisa descer à sala de máquinas e perguntar: existe um registro auditável de cada transformação entre o dado bruto e a decisão? Se a resposta for não, a conversa sobre modelos de IA precisa pausar.

Plataforma de dados governada na AWS

Da ingestão à decisão, com governança atravessando tudo. O ponto não são os serviços, é que cada zona tem dono, linhagem e política de acesso explícita.

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)

Fluxos

- stream -> bronze
- batch -> bronze
- bronze -> silver: qualidade
- silver -> gold: data products
- lf -> gold: governa acesso
- catalog -> silver: linhagem
- gold -> redshift
- gold -> feature: alimenta modelos

  Dado mal governado é risco regulatório com data de vencimento: A LGPD, o BACEN e o CMN não exigem perfeição imediata, exigem evidência de controle e trajetória de melhoria. Mas dado pessoal fluindo para modelo de IA sem consentimento verificado, score de crédito sem linhagem auditável e transações sem retenção adequada são passivos que se acumulam silenciosamente. Quando o incidente ocorre, vazamento, questionamento de supervisão ou reclamação de titular, a ausência de governança não é atenuante: é agravante. A multa da LGPD pode chegar a 2% do faturamento. O risco reputacional não tem teto.

Da plataforma ao modelo: o que a arquitetura de dados responde antes de qualquer IA

Quando a liderança sobe ao penthouse com dois objetivos, personalização com IA e redução de fraude, e pede ao arquiteto uma solução, a tentação é ir direto para o catálogo de serviços de machine learning. Resisto sempre a essa tentação. Minha resposta começa quatro andares abaixo, na sala de máquinas dos dados.

Para personalização com IA, as perguntas que a arquitetura de dados precisa responder primeiro são: o consentimento LGPD para a finalidade de personalização está capturado, segregado e verificável em tempo de execução do pipeline? Os dados de comportamento estão separados dos dados de transação financeira, ou estão misturados num schema que torna impossível aplicar controles diferentes? A qualidade do dado de comportamento é suficiente, qual a taxa de eventos perdidos, qual o atraso médio de ingestão? O custo de manter esses dados quentes para feature serving é compatível com o valor esperado do modelo? Nenhuma dessas perguntas é respondida pelo time de ciência de dados, são perguntas de arquitetura de dados com consequências de negócio e regulatórias.

Para redução de fraude em tempo real, o problema é diferente mas igualmente estrutural. Fraude exige sinais em milissegundos, não em horas de batch. Isso significa que a Feature Store (Amazon SageMaker Feature Store com online store ativado) precisa ser alimentada por streams de eventos, não por jobs noturnos. Significa que o pipeline do Capítulo 8, o tecido nervoso event-driven, precisa estar integrado à camada de dados de forma que um evento de transação Pix dispare atualização de features antes que a decisão de aprovação seja tomada. E significa que a intervenção humana precisa estar desenhada no fluxo: quando o modelo sinaliza fraude com baixa

confiança, existe um processo operacional para revisão? Quem decide? Em quanto tempo? Esse processo está auditável?

A plataforma de dados governada, com suas camadas Bronze, Silver e Gold, com Lake Formation controlando acesso, com Glue Catalog registrando linhagem e com Redshift e Athena servindo camadas diferentes de consumo, não é o destino. É a infraestrutura que torna possível responder essas perguntas com evidência, não com esperança.

Como iniciar uma arquitetura de dados governada em banco, sequência pragmática

1. Inventariar domínios e classificar dados por obrigação: Antes de qualquer infraestrutura, mapeie os domínios de negócio e, para cada tipo de dado relevante, responda: é dado pessoal (LGPD), dado financeiro (BACEN), dado operacional ou dado público? Qual a retenção mínima e máxima? Quem é o produtor responsável? Esse inventário é o insumo para todas as decisões de arquitetura seguintes.

2. Definir o modelo de ownership e o contrato de produto: Cada domínio nomeia um dono de dado responsável. Esse dono assina um contrato de produto: schema versionado, SLA de atualização, critérios de qualidade mensuráveis e política de acesso por finalidade. Sem contrato, não há produto, há apenas dado solto.

3. Implementar o Lake Formation com controle de acesso baseado em atributo: Configure o AWS Lake Formation como ponto central de controle de acesso ao lake. Use tags de sensibilidade (PII, financeiro, operacional) e políticas de finalidade para garantir que o acesso seja concedido com base em quem precisa do dado e para quê, não apenas em qual perfil técnico o usuário tem.

4. Instrumentar linhagem no Glue Catalog e nos jobs de transformação: Ative o rastreamento de linhagem no AWS Glue e enriqueça o Glue Catalog com metadados de negócio. Para cada job de transformação, registre explicitamente as fontes, as regras aplicadas e o destino. Considere Amazon DataZone para governança de catálogo em escala com múltiplos domínios.

5. Validar qualidade como portão de camada, não como relatório posterior: Implemente validações de qualidade (completude, unicidade, consistência de schema, regras de negócio) como etapa obrigatória nos jobs Glue que promovem dados de Bronze para Silver e de Silver para Gold. Dado que falha na validação não avança, é roteado para quarentena com alerta ao dono do produto.

Perguntas frequentes sobre arquitetura de dados em bancos

Redshift ou Athena, qual usar para analytics em banco?

Essa é a pergunta errada para começar. A resposta depende do padrão de acesso: Redshift para queries complexas e frequentes sobre dados estruturados com latência previsível (relatórios regulatórios, dashboards operacionais); Athena para exploração ad-hoc, auditoria forense e queries sobre dados semi-estruturados no S3. Em plataformas maduras, os dois coexistem: Redshift serve a camada Gold para consumidores recorrentes, Athena serve a camada Silver para analistas explorando dados. O que define a escolha é o SLA do consumidor e o padrão de acesso, não a preferência do time de dados.

Data mesh é adequado para bancos brasileiros?

Data mesh é um modelo organizacional, não uma tecnologia. Seus princípios, ownership por domínio, dado como produto, plataforma self-service, governança federada, são altamente compatíveis com bancos que já têm domínios de negócio bem definidos. O desafio em bancos

brasileiros é a governança federada: a LGPD e as normas do BACEN exigem controles centralizados de acesso e retenção que precisam ser implementados de forma consistente mesmo quando a ownership é distribuída. A solução é uma camada de governança centralizada (Lake Formation + políticas corporativas) sobre uma ownership de dados distribuída, não um ou outro.

Como tratar dados de terceiros (bureaus de crédito, Open Finance) na plataforma?

Dados de terceiros têm obrigações contratuais e regulatórias próprias que precisam ser refletidas na classificação do produto de dados. Dados de Open Finance, por exemplo, têm consentimento do titular com escopo e prazo definidos, o pipeline que os ingere precisa verificar se o consentimento ainda é válido antes de cada uso, não apenas na ingestão. Dados de bureaus de crédito têm restrições de redistribuição que precisam estar na política de acesso do produto. A regra prática é: dado de terceiro entra na camada Bronze com metadados de origem, contrato e restrição de uso explícitos, e nunca avança para Gold sem que essas restrições sejam verificadas.

O que separa uma plataforma de dados bancária madura de uma imatura

Uma plataforma imatura armazena dados e espera que alguém os use bem. Uma plataforma madura publica produtos de dados com contratos, linhagem auditável, qualidade mensurável e controle de acesso baseado em finalidade, e trata dado mal governado como o risco regulatório que ele é. A diferença não está na tecnologia escolhida: está em quem é responsável, pelo quê, com qual evidência. Quando o arquiteto consegue subir ao penthouse e dizer 'nossa decisão de crédito de ontem é explicável, auditável e defensável perante o BACEN', é porque a sala de máquinas foi construída com essa exigência em mente desde o primeiro dia.

10 - Plataforma e runtime: escolha pelo modelo operacional

EKS, ECS, Lambda ou EC2? A decisão madura não começa pela tecnologia favorita, começa pelo workload e pela maturidade operacional do time. E termina em golden paths, não em preferências.

A pergunta 'EKS ou Lambda?' aparece em quase toda discussão de plataforma que já participei, e quase sempre é a pergunta errada. O que realmente importa é quem vai operar isso às três da manhã, como o time faz deploy sem medo, e como a organização recupera um incidente antes que o regulador perceba. Runtime é consequência dessas respostas, não ponto de partida.

O andar errado para começar a decisão

Quando sobe ao penthouse, o CTO de um banco não pensa em pods Kubernetes nem em funções Lambda. Ele pensa em risco operacional, em janelas de manutenção que o Banco Central exige documentar, em custo de falha de um sistema de pagamentos às 23h59 de uma sexta-feira. Quando desce à sala de máquinas, o engenheiro pensa em imagens Docker, em cold starts, em limites de concorrência. O arquiteto que anda de elevador precisa traduzir essas duas linguagens sem perder nenhuma das duas.

O erro clássico é começar a decisão de plataforma no meio do prédio, na camada de preferência tecnológica. Um time que cresceu com Kubernetes vai querer EKS para tudo. Um time formado por ex-startups serverless vai querer Lambda para tudo. Nenhuma das duas posições é errada em si; ambas são perigosas quando se tornam universais.

A decisão madura começa com três perguntas de negócio que descem direto do penthouse: Qual é a tolerância a falha deste domínio? Um domínio de onboarding de clientes pode tolerar degradação por minutos; um domínio de liquidação interbancária, não. Qual é o padrão de carga? Picos previsíveis e curtos favorecem serverless; carga sustentada e uniforme favorece containers. Qual é a maturidade operacional do time responsável? Um time sem experiência em Kubernetes que herda um cluster EKS em produção não ganhou poder, ganhou responsabilidade sem preparo.

Só depois dessas respostas faz sentido abrir a matriz de decisão e mapear o workload ao runtime adequado. Como mostra a tabela a seguir, cada combinação de domínio bancário e runtime carrega um conjunto distinto de vereditos operacionais, e ignorar esses vereditos é a forma mais cara de aprender arquitetura.

> Minha posição depois de 16 anos em sistemas financeiros: Nunca vi um banco de porte médio ou grande que funcionasse bem com um único runtime. Toda arquitetura bancária real que já auditei ou desenhei é híbrida: canais digitais em BFFs containerizados, domínios críticos de liquidação e contabilidade em ECS ou EKS com deploy progressivo e controle fino de rollback, processos event-driven assíncronos em Lambda e Step Functions, dados quentes em Aurora e DynamoDB, dados frios em S3. A questão nunca foi 'qual runtime vence o debate', foi 'qual runtime serve melhor este workload específico, operado por este time específico, dentro deste envelope de risco'. Quem insiste em uniformidade está otimizando para conforto do arquiteto, não para resiliência do banco.

A matriz de decisão e o que ela realmente mede

A matriz de decisão que acompanha este capítulo: [DECISION MATRIX], organiza os principais runtimes disponíveis na AWS (EKS, ECS/Fargate, Lambda/Step Functions) contra as dimensões que mais importam em contexto bancário: modelo de responsabilidade operacional, superfície de ataque e postura de segurança, latência e comportamento sob carga, custo de falha e estratégia de recuperação, e velocidade de entrega de mudanças.

O que a matriz não faz, e é importante ser explícito, é declarar um vencedor absoluto. Ela mapeia vereditos condicionais. EKS vence quando o time tem maturidade de plataforma, o workload exige controle fino de scheduling, e a organização já opera Kubernetes em outros contextos. ECS/Fargate vence quando o time quer a semântica de containers sem o overhead de gerenciar o control plane, um trade-off que faz sentido para domínios de negócio que não são o core da plataforma de engenharia. Lambda/Step Functions vencem quando o padrão de invocação é esporádico ou orientado a eventos, quando o time quer pagar por execução e não por capacidade reservada, e quando a lógica de orquestração de processos longos precisa de durabilidade sem gerenciar filas manualmente.

No contexto de BACEN e CMN, há uma dimensão adicional que a maioria dos frameworks de decisão ignora: auditabilidade do deploy. Regulação brasileira exige que mudanças em sistemas críticos sejam rastreáveis, com evidência de aprovação e rollback documentado. EKS e ECS com GitOps (ArgoCD, Flux) entregam isso naturalmente via histórico de manifests. Lambda com SAM ou CDK entrega via CloudFormation change sets. A diferença não está no runtime, está em como o pipeline de CI/CD é construído sobre ele. Isso significa que a escolha de runtime e a escolha de estratégia de deploy são decisões acopladas, e tratá-las separadamente é um erro de design.

Um detalhe prático que aprendi da forma difícil: cold start em Lambda não é só latência, é risco de SLA. Em domínios de pagamento onde o SLA de resposta é de 500ms ponta a ponta, um cold start de 800ms em uma função Java não inicializada quebra o contrato. A solução não é

abandonar Lambda, é usar Provisioned Concurrency, SnapStart para JVM, ou simplesmente escolher um runtime de execução mais leve. Mas isso precisa estar na decisão, não ser descoberto em produção.

Escolhendo o runtime de um domínio bancário

Amazon EKS (Kubernetes)

Pros

- Padronização multi-time e portabilidade
- Controle fino de runtime, service mesh, operadores
- Forte para escala de plataforma constante

Cons

- Alta carga operacional e maturidade exigida
- Custo de cluster mesmo ocioso
- Complexidade que vira risco se o time é pequeno

Verdict: Quando há escala de plataforma, vários times e capacidade operacional real para operar Kubernetes.

Amazon ECS / Fargate

Pros

- Containers com muito menos carga que Kubernetes
- Integração nativa com o ecossistema AWS
- Bom meio-termo para times sem SRE dedicado

Cons

- Menos flexível que EKS para casos avançados
- Ecossistema de terceiros menor

Verdict: Quando o time quer containers e previsibilidade, sem assumir a complexidade do Kubernetes.

Lambda + Step Functions

Pros

- Operação mínima, escala automática
- Ideal para fluxos event-driven e integração
- Custo zero em ociosidade

Cons

- Exige disciplina em idempotência e limites
- Cold start e tetos de execução
- Observabilidade e IAM precisam de rigor

Verdict: Quando o fluxo é orientado a eventos, intermitente, e a baixa operação vale mais que o controle fino.

Golden paths: quando o caminho certo é o mais fácil

Existe uma tensão permanente entre autonomia dos times de produto e governança da plataforma. Times de produto querem mover rápido, escolher ferramentas, experimentar. A plataforma quer

padronizar, auditar, controlar custos. Em bancos, essa tensão é amplificada pelo regulador: qualquer desvio de padrão pode virar achado de auditoria.

A solução que funciona, e que vi funcionar em organizações financeiras que conseguiram escalar engenharia sem perder controle, é o conceito de golden paths: caminhos pavimentados que tornam a escolha correta a escolha mais fácil. Um golden path não proíbe alternativas; ele as torna mais custosas em esforço. Se o template de serviço já vem com observabilidade configurada, SAST no pipeline, política de IAM mínima, e integração com o service mesh, o time que decide sair desse template precisa justificar e arcar com o custo de manutenção do desvio.

Na prática, um golden path bancário na AWS tem pelo menos cinco componentes: template de serviço (repositório com estrutura de projeto, Dockerfile ou SAM template, pipeline de CI/CD pré-configurado); observabilidade por padrão (CloudWatch structured logs, X-Ray tracing, métricas de negócio via EMF já configuradas no template); segurança por padrão (IAM roles com least privilege, secrets via Secrets Manager, scanning de imagem no ECR, checagem de dependências no pipeline); política como código (SCPs no AWS Organizations, Config Rules, OPA/Rego para validação de manifests Kubernetes se EKS for o runtime escolhido); e runbook de incidente (documentação de como escalar, como fazer rollback, quem acionar, integrado ao sistema de on-call).

O efeito colateral mais valioso de golden paths é cultural: governança para de ser briga. Quando o caminho seguro é também o caminho rápido, o time de produto não sente a plataforma como obstáculo, sente como acelerador. Isso muda a dinâmica de toda a organização de engenharia, e é especialmente crítico em bancos onde a pressão regulatória já cria atrito suficiente.

> Como começar um golden path sem uma plataforma madura: Não espere ter uma Internal Developer Platform completa para começar. Comece com um único template de repositório no GitHub/CodeCommit que já resolve os cinco pontos mais dolorosos do time: estrutura de projeto, pipeline básico com SAST, variáveis de ambiente via Secrets Manager, logging estruturado, e um README de runbook. Esse template simples já é um golden path. Itere sobre ele a cada sprint com base no que os times reclamam. Plataforma não é um projeto, é um produto interno com usuários reais.

O modelo operacional é a decisão real

Voltando ao elevador: quando sobe ao penthouse com a decisão de runtime, o que o board e o CRO de um banco precisam entender não é qual serviço AWS foi escolhido. Eles precisam entender quem é responsável pelo quê quando algo falha. Essa é a tradução que o arquiteto precisa fazer.

Em EKS, o time de plataforma é responsável pelo control plane gerenciado pela AWS, pelos node groups, pelo CNI, pelo ingress controller, pelo service mesh, pela política de rede. O time de produto é responsável pela aplicação, pelo Dockerfile, pelo manifesto Kubernetes, pela lógica de negócio. Quando um pod crashea às 3h, quem acorda? Quando o cluster precisa de upgrade de versão do Kubernetes, quem planeja a janela de manutenção e comunica ao regulador?

Em ECS/Fargate, o modelo muda: a AWS gerencia o plano de controle e o provisionamento de infraestrutura de compute. O time de plataforma ainda é responsável pela definição de task, pela rede VPC, pelas políticas de IAM, pela estratégia de deploy. O time de produto é responsável pela imagem e pela lógica. O on-call de infraestrutura é mais simples, mas o custo por unidade de compute é geralmente maior que EC2 equivalente, e o controle de scheduling é menor.

Em Lambda, o modelo vai mais longe: a AWS gerencia execução, scaling, disponibilidade da

função. O time é responsável pelo código, pelo timeout, pela memória configurada, pela política de retry, pelo dead-letter queue. O on-call de infraestrutura praticamente desaparece, mas o on-call de lógica de negócio aumenta, porque falhas silenciosas em processamento assíncrono são mais difíceis de detectar sem observabilidade bem configurada.

A escolha de runtime, portanto, é uma escolha de onde a responsabilidade operacional se concentra. Bancos com times de plataforma maduros e grandes podem absorver EKS. Bancos digitais menores com times de produto ágeis mas sem engenharia de plataforma dedicada se beneficiam de ECS/Fargate e Lambda. E a maioria dos bancos reais vive no meio: uma combinação deliberada que mapeia responsabilidade ao nível de maturidade de cada time, domínio por domínio. Essa combinação deliberada, e não a uniformidade, é o sinal de uma arquitetura madura.

O que levar deste capítulo

- A escolha de runtime começa nas perguntas de negócio, tolerância a falha, padrão de carga, maturidade operacional do time, não na preferência tecnológica.
- Arquiteturas bancárias reais são híbridas: BFFs em containers, domínios críticos em EKS/ECS, processos event-driven em Lambda/Step Functions, dados em Aurora/DynamoDB/S3.
- A escolha de runtime e a estratégia de deploy são decisões acopladas; auditabilidade de mudanças é requisito regulatório no Brasil, não opcional.
- Golden paths tornam a escolha correta a mais fácil: template, observabilidade, segurança, policy-as-code e runbook por padrão, não por esforço adicional.
- O modelo operacional, quem opera às 3h, como se faz deploy, como se recupera, é a decisão real; o runtime é a resposta técnica a esse modelo.
- Uniformidade de runtime é otimização para conforto do arquiteto; combinação deliberada por domínio é otimização para resiliência do banco.

Veredicto do capítulo

Não existe resposta universal para 'qual runtime usar'. Existe uma pergunta universal que precisa ser respondida antes: qual é o modelo operacional que este domínio exige e que este time consegue sustentar? Quando essa pergunta é respondida com honestidade, considerando maturidade real, não maturidade aspiracional, a escolha de runtime se torna consequência natural, não campo de batalha. Golden paths transformam essa consequência em padrão reproduzível. E padrões reproduzíveis são o que separa uma plataforma bancária que escala de uma coleção de heróis individuais que não dormem.

11 - IA generativa com guardrails: valor sem caixa-preta

A IA generativa aproxima conhecimento, automação e decisão, mas em banco só cria valor sustentável com segurança, avaliação, rastreabilidade e limites. Bedrock ajuda; a arquitetura é que define as fronteiras.

IA generativa em banco não é uma funcionalidade, é uma capacidade com dono, política de risco, SLO e plano de fallback, ou não é nada além de um piloto eterno que nunca chega à produção. O modelo de linguagem é o componente mais fácil de trocar; as fronteiras que o cercam são o que protege o banco, o cliente e a licença de operação. Neste capítulo fecho a Parte III mostrando

como a arquitetura desce até embeddings, ferramentas e logs de autorização, e sobe de volta ao penthouse como produtividade mensurável, escala de atendimento e evidência regulatória.

> Minha posição depois de 16 anos em sistemas financeiros: Toda vez que apresento IA generativa para um comitê de risco bancário, a primeira pergunta não é 'qual modelo?', é 'quem responde quando der errado?'. Essa pergunta é exatamente a certa. Em todos os projetos que arquitetei com Bedrock, o diferencial não foi a escolha do LLM; foi a disciplina de tratar o sistema de IA como qualquer outro serviço crítico: backlog priorizado, SLO definido, telemetria desde o dia zero, versionamento de prompt como código e avaliação contínua com datasets reais. Bancos que pulam essa disciplina e entram direto na demo do copiloto estão construindo dívida técnica e regulatória ao mesmo tempo. Minha tese é simples: guardrail sem telemetria é promessa sem prova; RAG sem curadoria é busca cara com aparência inteligente; agente sem autorização contextual é risco operacional esperando um incidente para se materializar.

O elevador sobe com valor, desce com responsabilidade

No penthouse, o executivo vê três promessas concretas da IA generativa: copiloto de atendimento que reduz tempo médio de resolução; analisador de contratos que acelera onboarding jurídico; assistente interno que democratiza o conhecimento de compliance. Essas promessas são legítimas. O problema começa quando o arquiteto não desce no elevador para mostrar o que sustenta cada uma delas na sala de máquinas.

Quando desço, encontro quatro camadas que precisam existir antes de qualquer modelo ser chamado em produção. A primeira é dados governados: o RAG só é confiável se a base de conhecimento tem curadoria, versionamento e linhagem rastreável, o que conecta diretamente ao Capítulo 9, onde tratei dados como produto. A segunda é autorização contextual: cada ação que um agente executa precisa carregar a identidade do usuário, o escopo de permissão e o contexto da sessão; sem isso, o agente age como um usuário com privilégios excessivos. A terceira é telemetria de qualidade: cada inferência precisa ser logada com prompt versionado, resposta, latência, tokens consumidos e, quando possível, feedback explícito ou implícito do usuário. A quarta é política de risco operacionalizada: não um documento de governança na gaveta, mas guardrails configurados, testados, monitorados e com alertas ativos.

Essa descida não é burocracia, é o que permite ao arquiteto subir de volta ao penthouse com evidência, não com esperança. O comitê de risco não quer saber se o modelo é bom; quer saber se o sistema é auditável, se o custo é previsível e se existe um humano no laço quando a decisão tem consequência regulatória.

Como o Amazon Bedrock estrutura as fronteiras, e por que o modelo é o componente mais intercambiável

Como mostra o diagrama a seguir, a arquitetura de referência que uso em projetos bancários com Bedrock não é organizada em torno do modelo, é organizada em torno das fronteiras. Guardrails na entrada e na saída, RAG sobre dados governados com Knowledge Bases, ações de agentes restritas por autorização contextual, e telemetria capturada em cada camada.

Os Guardrails do Bedrock operacionalizam políticas que de outra forma existiriam apenas em documentos: filtros de conteúdo configuráveis por categoria e intensidade, detecção e mascaramento de PII antes que qualquer dado sensível chegue ao modelo ou retorne ao usuário, bloqueio de tópicos proibidos (como aconselhamento de investimento não regulado), e proteção contra prompt injection, um vetor de ataque subestimado em ambientes bancários onde o agente

tem acesso a ferramentas reais. Cada decisão do guardrail é logada, o que transforma a política em evidência auditável.

As Knowledge Bases resolvem o problema de RAG sem curadoria. Em vez de indexar documentos brutos e torcer para que a recuperação seja relevante, a arquitetura exige que cada fonte de conhecimento passe por um pipeline de ingestão com validação, versionamento e metadados de linhagem. O modelo não acessa dados; acessa fragmentos curados com proveniência rastreável. Isso é a diferença entre um sistema que o auditor consegue inspecionar e um que ele precisa confiar cegamente.

Os Agents e o AgentCore fecham o ciclo de automação, mas com uma restrição arquitetural inegociável: nenhuma ferramenta é chamada sem que o contexto de autorização do usuário seja validado no momento da execução. O agente não herda permissões do sistema, ele carrega o contexto do usuário e o escopo da sessão, e cada chamada de ferramenta é registrada com esses atributos. Isso é o que separa um agente bancário de um script com LLM na frente.

Os três pecados da IA generativa em banco

Em projetos de arquitetura, antipadrões ensinam mais do que padrões porque mostram onde a pressão por velocidade vence a disciplina de engenharia. Na IA generativa bancária, identifico três pecados recorrentes.

Primeiro pecado: RAG sem curadoria. A equipe indexa documentos de regulação, manuais de produto e políticas internas sem pipeline de qualidade. O resultado é uma base de conhecimento que mistura versões desatualizadas de normas com documentos válidos, sem metadados de vigência. O modelo recupera fragmentos plausíveis mas incorretos, e o usuário recebe respostas com aparência de autoridade sobre informações obsoletas. O custo de inferência é real; a qualidade da resposta é ilusória. Curadoria não é opcional, é o SLO do RAG.

Segundo pecado: agente sem autorização contextual. O agente é configurado com credenciais de serviço que têm acesso amplo às APIs do banco, e a autorização do usuário é verificada apenas na camada de apresentação. Quando o agente encadeia chamadas de ferramentas, consultar saldo, verificar limite, iniciar transferência, ele opera com privilégios que o usuário autenticado não teria se acessasse as APIs diretamente. Isso não é só um risco de segurança; é uma violação do princípio de menor privilégio que qualquer auditoria de segurança vai identificar. O contexto de autorização precisa descer junto com a requisição, até a última ferramenta chamada.

Terceiro pecado: guardrail sem telemetria. O time configura filtros de conteúdo e apresenta ao comitê de risco como evidência de controle. Mas sem logging estruturado das decisões do guardrail, quantas requisições foram bloqueadas, por qual categoria, com qual conteúdo mascarado, a política existe apenas como configuração, não como prova. Na primeira auditoria do BACEN sobre uso de IA em atendimento, a pergunta será: 'mostre-me os logs de controle dos últimos 90 dias'. Sem telemetria, a resposta é silêncio.

RAG e agentes no banco com Amazon Bedrock

Toda interação atravessa guardrails na entrada e na saída, recupera contexto governado e só executa ações com autorização contextual e telemetria. O modelo é o componente mais fácil de trocar; as fronteiras é que protegem o banco.

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)

Fluxos

- user -> guard: pergunta
- guard -> budget
- guard -> agent: se aprovado
- agent -> model: infere
- agent -> kb: recupera contexto
- agent -> tools: ação com authz
- agent -> telemetry: registra tudo
- model -> guard: filtra saída

IA generativa como capacidade bancária: os pilares inegociáveis

- Dono de capacidade com backlog, SLO e política de risco documentada, não um projeto de inovação sem responsável.
- Versionamento de prompt como código: cada versão rastreada, testada contra datasets de avaliação e promovida via pipeline.
- Orçamento diário de tokens com alertas e circuit breaker: custo de IA é custo operacional, não surpresa no fim do mês.
- Blacklist de termos e tópicos proibidos configurada e testada, não assumida como coberta pelo modelo base.
- Plano de fallback explícito: o que o sistema faz quando o modelo está indisponível, quando o guardrail bloqueia, quando a confiança da resposta está abaixo do limiar.
- Avaliação contínua com datasets rotulados: qualidade não é percepção subjetiva, é métrica com baseline e tendência.

Controle de custo e avaliação contínua: a disciplina que separa piloto de produção

Um dos sinais mais claros de que uma implementação de IA generativa não está pronta para produção bancária é a ausência de orçamento de tokens como controle operacional. Token é unidade de custo e de risco: um agente mal configurado pode consumir em minutos o que estava orçado para um dia, e um prompt injection bem-sucedido pode forçar o modelo a gerar respostas longas e custosas em loop. O orçamento diário de tokens com alertas progressivos e circuit breaker automático não é uma otimização de custo, é um controle de risco.

Na arquitetura que uso, o orçamento é implementado em duas camadas. A primeira é no nível da conta AWS, com alertas de billing configurados para disparar antes do limite, não depois. A

segunda é no nível da aplicação, com um contador de tokens por sessão e por usuário que interrompe a interação quando o limiar é atingido e registra o evento como anomalia a ser investigada. Isso cobre tanto o risco financeiro quanto o vetor de abuso.

A avaliação contínua é o mecanismo que fecha o ciclo de qualidade. Em vez de confiar na percepção qualitativa da equipe de produto, a arquitetura mantém um conjunto de datasets de avaliação com casos representativos do domínio bancário, perguntas sobre produtos, cenários de compliance, consultas de saldo, situações de atendimento, com respostas esperadas anotadas por especialistas. A cada nova versão de prompt, o pipeline de avaliação executa automaticamente, calcula métricas de qualidade (precisão factual, taxa de alucinação estimada, aderência à política) e bloqueia a promoção se os limiares não forem atingidos. Isso é o que transforma versionamento de prompt de prática de engenharia em evidência regulatória: cada versão em produção tem um scorecard de qualidade associado, com data, dataset e resultado.

Como tratar IA generativa como capacidade bancária, sequência de implantação

1. Definir a capacidade, não a feature: Antes de escolher modelo ou framework, documente: qual problema de negócio esta capacidade resolve, quem é o dono, qual é o SLO de qualidade e disponibilidade, e qual é a política de risco associada. Sem isso, o projeto nasce como piloto e morre como piloto.

2. Construir a base de dados governada antes do RAG: Defina o pipeline de ingestão com validação de qualidade, versionamento de documentos, metadados de vigência e linhagem rastreável. A Knowledge Base do Bedrock é o destino; a curadoria dos dados é o pré-requisito.

3. Configurar e testar guardrails antes de qualquer demo: Implemente filtros de conteúdo, detecção de PII, blacklist de tópicos e proteção contra prompt injection. Teste com casos adversariais reais. Ative logging estruturado de todas as decisões do guardrail desde o primeiro dia.

4. Implementar autorização contextual nos agentes: Cada ferramenta chamada pelo agente deve receber e validar o contexto de autorização do usuário. Nunca use credenciais de serviço com escopo amplo como substituto para autorização por usuário. Registre identidade, escopo e timestamp em cada chamada.

5. Estabelecer orçamento de tokens e avaliação contínua: Configure orçamento diário com alertas e circuit breaker. Crie o dataset de avaliação inicial com especialistas do domínio. Integre avaliação automática ao pipeline de promoção de versões de prompt.

O que o comitê de risco sempre pergunta sobre IA generativa

Como garantimos que dados de clientes (PII) não vazam para o modelo ou para outros usuários?

Os Guardrails do Bedrock detectam e mascaram PII na entrada antes que o dado chegue ao modelo, e na saída antes que a resposta chegue ao usuário. Além disso, a arquitetura de RAG não armazena dados de clientes na Knowledge Base, armazena apenas documentos de política e produto. Dados transacionais são acessados via ferramentas de agente com autorização contextual, nunca injetados diretamente no prompt. Cada operação de mascaramento é logada com timestamp e categoria, gerando trilha auditável.

Se a IA errar em uma análise de crédito, quem é responsável? Podemos usar IA em decisões reguladas?

Em fluxos regulados, crédito, KYC, prevenção à lavagem de dinheiro, a IA generativa atua como assistente, não como decisor. O modelo pode sintetizar informações, destacar inconsistências e sugerir perguntas de due diligence, mas a decisão final é sempre de um humano identificado e registrado no sistema. Isso não é limitação de tecnologia; é requisito de governança que o BACEN e a LGPD já sinalizam como expectativa. O arquiteto precisa garantir que o fluxo técnico reflita essa distinção: a saída do modelo é uma recomendação com nível de confiança, não uma instrução executável.

Como controlamos o custo? IA generativa pode gerar surpresas no billing.

O controle de custo é arquitetural, não apenas financeiro. Implementamos orçamento diário de tokens com três camadas: alertas de billing na conta AWS (disparando a 70% e 90% do limite), contador de tokens por sessão na aplicação com interrupção automática ao atingir o limiar, e revisão semanal de padrões de consumo para identificar sessões anômalas. Adicionalmente, o versionamento de prompt com avaliação de custo por versão permite identificar se uma mudança de prompt aumentou o consumo médio de tokens sem ganho proporcional de qualidade.

Como provamos que a qualidade do sistema de IA é adequada e se mantém ao longo do tempo?

Qualidade em IA generativa não é percepção, é métrica com baseline, tendência e threshold de promoção. A arquitetura mantém datasets de avaliação com casos anotados por especialistas do domínio bancário, organizados por categoria (produto, compliance, atendimento). A cada nova versão de prompt, o pipeline de CI/CD executa avaliação automática e calcula métricas de precisão factual, aderência à política e taxa de respostas fora do escopo. A versão só é promovida para produção se atingir os thresholds definidos. O histórico de avaliações é retido como evidência regulatória, associado à versão do prompt e ao dataset usado.

O modelo é o componente mais fácil de trocar, as fronteiras são o ativo

Encerro este capítulo, e a Parte III, com a afirmação que mais incomoda equipes apaixonadas por modelos de linguagem: o LLM é o componente com menor custo de substituição em toda a arquitetura. Modelos evoluem, novos são lançados, benchmarks mudam, preços caem. O que não é fácil de trocar é a base de conhecimento curada, o pipeline de avaliação com datasets proprietários, os guardrails calibrados para o contexto regulatório do banco, a telemetria estruturada que acumula meses de evidência auditável, e a confiança do comitê de risco construída ao longo de incidentes não acontecidos.

Essa inversão de perspectiva é o que separa uma arquitetura de IA generativa bancária de um wrapper de API com interface bonita. O wrapper pode ser construído em dias; a arquitetura leva meses porque as fronteiras precisam ser projetadas, testadas, operacionalizadas e provadas. Mas quando o próximo modelo com melhor desempenho for lançado, a troca leva horas, porque as fronteiras já estão lá.

No elevador de Hohpe, a IA generativa sobe ao penthouse como produtividade de atendimento, escala sem contratação proporcional, aceleração de processos jurídicos e democratização de conhecimento de compliance. Ela desce à sala de máquinas como embeddings indexados sobre dados governados, ferramentas com autorização contextual, guardrails com telemetria estruturada, orçamento de tokens com circuit breaker e pipeline de avaliação contínua com datasets proprietários. O arquiteto que consegue fazer essa tradução nos dois sentidos, e documentar cada decisão com o raciocínio de trade-off, é o que o banco precisa. Não o que sabe qual modelo tem o

melhor benchmark esta semana.

IA generativa em banco: quando vira arquitetura de verdade

IA generativa vira arquitetura bancária de verdade quando tem dono, SLO, política de risco operacionalizada, telemetria desde o dia zero, versionamento de prompt como código, avaliação contínua com datasets proprietários e plano de fallback testado. O Amazon Bedrock oferece os blocos certos: Guardrails, Knowledge Bases, Agents, mas é a disciplina arquitetural que transforma esses blocos em um sistema auditável, previsível e evolutivo. O modelo é o componente mais fácil de trocar; as fronteiras são o ativo que o banco constrói ao longo do tempo. Qualquer implementação que não consiga responder às quatro perguntas do comitê de risco deste capítulo ainda é um piloto, independentemente de quantos usuários já estejam usando.

PARTE IV

Segurança, Regulação e Operação

O que separa um diagrama bonito de um sistema bancário real: segurança como evidência, conformidade como restrição de design e operação como o único lugar onde a arquitetura existe de verdade.

12 - Segurança como evidência, não como opinião

Bancos precisam provar: quem acessou, quando, por quê, com qual privilégio, em qual dado. Segurança em arquitetura financeira é menos checklist isolado e mais evidência contínua, embutida no desenho.

Segurança em arquitetura bancária não é uma camada que você adiciona ao final, é uma restrição de design que atravessa identidade, criptografia, logging, retenção, rede e resposta a incidente desde o primeiro diagrama. A pergunta que o regulador faz não é 'vocês parecem seguros?', mas sim 'vocês conseguem provar quem acessou esse dado, quando, com qual privilégio, por qual canal, e com qual controle compensatório ativo naquele momento?', e a resposta precisa existir em forma de evidência imutável, correlacionável e retida pelo prazo exigido.

> Minha visão: segurança como evidência é uma postura de design, não de compliance: Depois de mais de dezesseis anos trabalhando com sistemas financeiros, a distinção que mais separa arquiteturas que sobrevivem a auditorias das que desmoronam sob pressão regulatória é simples: as primeiras foram desenhadas para produzir evidência, as segundas foram desenhadas para funcionar e depois tentaram encaixar evidência. Quando um banco sofre um incidente e o BACEN ou o DREX-audit pede a trilha completa de uma transação, quem iniciou, quem aprovou, qual chave criptografou, qual regra de PLD foi avaliada, qual dado pessoal foi acessado e por quem, essa trilha ou existe de forma estruturada e imutável, ou não existe. Não há meio-termo. Tratar conformidade como burocracia final é a decisão arquitetural mais cara que um banco pode tomar, porque o custo de reescrever logging, retenção e controle de acesso sob pressão de um incidente real é uma ordem de magnitude maior do que tê-los embutido desde o início.

Conformidade é restrição de design, não checklist de entrega

Quando o arquiteto sobe ao penthouse e conversa com o Chief Risk Officer sobre obrigações regulatórias, ouve frases como 'reter evidência de PLD por cinco anos', 'segregar dados por jurisdição', 'provar quem aprovou cada operação acima de determinado valor'. Essas frases soam como requisitos de compliance. Mas quando o arquiteto desce à sala de máquinas e começa a desenhar, percebe que cada uma delas é, na verdade, um requisito não-funcional que muda decisões concretas de arquitetura.

'Reter evidência de PLD por cinco anos' implica: escolha de storage com política de retenção imutável (S3 Object Lock com modo COMPLIANCE), definição de quais eventos constituem evidência de PLD, esquema de indexação que permita recuperação por cliente, período e tipo de operação em tempo hábil para resposta regulatória, e controle de acesso que impeça até o administrador de deletar registros antes do prazo. Isso atravessa storage, IAM, schema de eventos e processo operacional.

'Segregar dados por jurisdição' implica: múltiplas contas AWS organizadas via AWS Organizations com SCPs que proíbem replicação cross-region de dados sensíveis, KMS com chaves por região sem permissão de exportação, e pipelines de dados que respeitam fronteiras de jurisdição como invariante, não como configuração opcional.

'Provar quem aprovou' implica: cada aprovação é um evento assinado, com identidade federada rastreável até o usuário humano via IAM Identity Center, timestamp auditável no CloudTrail, e correlação com o evento de negócio que ela autorizou. Não basta ter um log de aprovação separado, ele precisa ser correlacionável com a transação real.

O arquiteto que trata essas restrições como camada final descobre, invariavelmente, que elas atravessam cada componente do sistema. Reescrever sob pressão de auditoria é o pior momento possível para aprender isso.

> Segurança como opinião: o banco perde: A frase mais perigosa que já ouvi em revisão de arquitetura bancária é 'eu acho que é seguro o suficiente'. A pergunta certa nunca é se parece seguro, é se você consegue provar. Um controle que você não consegue auditar é uma promessa, não uma proteção. Quando o regulador chega, quando o incidente acontece, quando o auditor externo pede evidência, 'eu acho' não é uma resposta aceitável. A diferença entre um banco que passa por uma auditoria regulatória com confiança e um que entra em modo de pânico é exatamente essa: o primeiro tem evidência estruturada e imutável; o segundo tem intenção não documentada.

HSM na prática: o sistema pede assinaturas, nunca a chave

O princípio mais importante da gestão de chaves criptográficas em sistemas financeiros é tão simples que é fácil de ignorar: a chave nunca sai do hardware. Chaves que assinam transações, que geram criptogramas de cartão, que protegem a comunicação com o BACEN, essas chaves nascem dentro do hardware, vivem dentro do hardware e morrem dentro do hardware. O sistema nunca vê o material da chave; ele envia dados ao HSM e recebe de volta uma assinatura, um criptograma ou um dado cifrado.

No contexto AWS, isso se materializa de duas formas complementares. O AWS KMS oferece um modelo gerenciado onde as chaves residem em HSMs operados pela AWS, com suporte a CMKs (Customer Managed Keys) que permitem ao banco controlar política de uso, rotação e auditoria via CloudTrail, cada chamada a kms:Decrypt, kms:GenerateDataKey ou kms:Sign gera um registro com identidade do chamador, recurso acessado e resultado. O AWS CloudHSM oferece HSMs dedicados, single-tenant, onde o banco detém controle exclusivo do material de chave, a AWS não tem acesso. Para operações que exigem certificação FIPS 140-2 Level 3 com custódia exclusiva

do banco, CloudHSM é o caminho; para a grande maioria das operações de envelope encryption e assinatura de tokens, KMS com CMKs é suficiente e operacionalmente mais simples.

Um padrão que uso consistentemente em sistemas de pagamento: envelope encryption com hierarquia de chaves. A Data Encryption Key (DEK) cifra o dado; a Key Encryption Key (KEK) cifra a DEK; a KEK vive no KMS ou CloudHSM. O dado em repouso nunca está exposto sem uma chamada explícita e auditada ao KMS. Quando o regulador pergunta 'quem decifrou esse dado de cliente em 14 de março às 03h47?', a resposta está no CloudTrail: identidade, recurso, timestamp, resultado, imutável, porque o bucket de logs do CloudTrail está protegido com S3 Object Lock e a chave de criptografia desse bucket tem política que proíbe deleção.

Para geração de criptogramas de cartão (EMV, tokenização), o padrão é ainda mais restritivo: o HSM executa o algoritmo internamente, e o sistema de autorização envia apenas os dados de entrada e recebe o criptograma de saída. Nenhum componente de software da aplicação toca material criptográfico sensível. Isso não é paranoia, é o modelo que os esquemas de bandeira (Visa, Mastercard) e o BACEN exigem.

Zero trust e evidência na AWS

Cada acesso é verificado, cada chave é gerenciada, cada ação é registrada de forma imutável. A segurança não é um muro na borda, é uma propriedade de cada camada.

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)

Fluxos

- idc -> kms: acesso a chave
- scp -> idc: limita
- kms -> trail: uso registrado
- config -> hub: achados
- macie -> hub: dados sensíveis
- hub -> trail: correlaciona

Zero trust como arquitetura de evidência contínua

O diagrama a seguir: Zero Trust e Evidência na AWS, materializa o modelo que descrevo neste capítulo. Mas antes de analisá-lo, é importante entender que zero trust em bancos não é um produto que você compra; é uma postura arquitetural que se traduz em decisões concretas em cada camada.

No nível de identidade, o ponto de partida é IAM Identity Center com federação para o diretório corporativo do banco. Nenhum usuário humano tem acesso direto via IAM users com chaves de longa duração, toda sessão é federada, com duração limitada, e cada ação é rastreável até o indivíduo. SCPs (Service Control Policies) no AWS Organizations funcionam como guardrails inegociáveis: mesmo que uma política IAM permita uma ação, a SCP pode bloqueá-la em nível organizacional. Uso SCPs para proibir, por exemplo, desabilitar CloudTrail, criar usuários IAM com console access sem MFA, ou replicar dados de produção para contas de desenvolvimento.

No nível de dados, Amazon Macie escaneia continuamente buckets S3 em busca de dados pessoais e financeiros não catalogados, não como auditoria periódica, mas como detector contínuo. Quando Macie encontra um CPF ou número de cartão em um bucket que não deveria contê-los, isso é um sinal de que um pipeline de dados vazou informação para o lugar errado. Integrado ao Security Hub, esse achado vira um finding com severidade, owner e prazo de remediação.

AWS Config registra o estado de cada recurso e suas mudanças ao longo do tempo. Quando o regulador pergunta 'qual era a configuração do security group do servidor de pagamentos em 14 de março?', Config tem a resposta, não como memória humana, mas como registro imutável. Regras do Config detectam desvios de conformidade em tempo real: um bucket S3 sem criptografia, um security group com porta 22 aberta para 0.0.0.0/0, uma instância EC2 sem tag de classificação de dados.

GuardDuty analisa VPC Flow Logs, DNS logs e CloudTrail para detectar comportamentos anômalos, uma instância fazendo chamadas a IPs de C2 conhecidos, um usuário IAM fazendo ListBuckets em todas as regiões às 3h da manhã, um padrão de acesso a dados que diverge do baseline histórico. Isso não substitui controles preventivos, mas fecha o ciclo: prevenção + detecção + evidência = postura de segurança auditável.

O ponto central que o diagrama ilustra é que cada componente: IAM Identity Center, Organizations/SCPs, KMS, Secrets Manager, Macie, CloudTrail, Config, Security Hub, GuardDuty, não é uma ferramenta isolada de segurança. É um produtor de evidência que alimenta um registro contínuo e correlacionável do estado de segurança do ambiente. A integração entre eles é o que transforma ferramentas em arquitetura.

Os princípios de segurança como evidência

- Conformidade é restrição de design desde o diagrama zero, tratar como camada final garante reescrita sob pressão de auditoria.
- A chave criptográfica nunca sai do hardware: o sistema pede assinaturas e criptogramas ao HSM, nunca o material da chave.
- Todo acesso relevante gera registro imutável, correlacionável e retido pelo prazo regulatório: CloudTrail com S3 Object Lock é o piso mínimo.
- SCPs no AWS Organizations são guardrails inegociáveis: operam acima de qualquer política IAM e não podem ser contornados por conta individual.
- Um controle que você não consegue auditar é uma promessa, não uma proteção, se não há evidência, o controle não existe para o regulador.
- Zero trust não é produto, é postura: cada componente de segurança deve ser também um produtor de evidência integrado ao ciclo de detecção e resposta.

Auditabilidade como capacidade de primeira classe

O arquiteto que sobe ao penthouse para discutir risco regulatório e desce à sala de máquinas para

desenhar pipelines de dados precisa manter um fio condutor claro: toda ação relevante para o negócio precisa gerar um registro que sobreviva ao pior cenário. Não apenas o pior cenário técnico, falha de componente, corrupção de dados, mas o pior cenário regulatório: um incidente de segurança investigado pelo BACEN, uma auditoria de PLD, uma ação judicial que exige reconstrução da cadeia de custódia de uma transação.

Auditabilidade em sistemas bancários tem três dimensões que precisam ser endereçadas explicitamente no design:

Completude: o que precisa ser registrado? Não apenas erros e exceções, cada operação de leitura em dados de cliente, cada aprovação de crédito, cada mudança de configuração em sistema de produção, cada acesso a chave criptográfica. O escopo do que constitui 'ação relevante' precisa ser definido com o time de compliance antes do primeiro sprint de desenvolvimento, não depois.

Imutabilidade: o registro precisa ser à prova de adulteração, inclusive por administradores internos. S3 Object Lock com modo COMPLIANCE impede deleção ou modificação mesmo por usuários com permissões administrativas, a única forma de remover o lock é via processo formal com aprovação da AWS. CloudTrail com validação de integridade de arquivo detecta qualquer tentativa de modificação retroativa. Logs enviados para uma conta AWS separada, com acesso restrito e SCP que proíbe deleção, criam uma segunda linha de defesa.

Correlacionabilidade: um log isolado não conta uma história. O regulador não quer saber apenas que 'usuário X acessou sistema Y às 14h32', quer saber qual transação foi processada, qual dado de cliente foi lido, qual regra de negócio foi aplicada, qual controle compensatório estava ativo. Isso exige que eventos de diferentes sistemas, aplicação, banco de dados, KMS, rede, compartilhem um identificador de correlação comum. O design do esquema de eventos (como discutido no Capítulo 8 sobre event-driven) e o design da auditoria são, na prática, o mesmo problema visto de ângulos diferentes.

Quando o regulador pergunta 'quem moveu esse dinheiro e com qual autorização?', a resposta precisa existir em forma estruturada, recuperável em minutos, e verificável como autêntica. Não como reconstrução post-hoc a partir de logs fragmentados, como consulta a um registro que foi desenhado para existir desde o início.

Como embutir evidência desde o diagrama zero

1. Mapeie restrições regulatórias como requisitos não-funcionais explícitos: Antes do primeiro diagrama de arquitetura, liste cada obrigação regulatória relevante (PLD, LGPD, Resolução CMN 4.658, PCI-DSS) e converta em requisito concreto: prazo de retenção, escopo de dado, controle de acesso, evidência exigida. Esses requisitos entram no backlog com a mesma prioridade que requisitos funcionais.

2. Defina o esquema de eventos de auditoria junto com o esquema de domínio: Cada evento de domínio (transação iniciada, crédito aprovado, dado de cliente atualizado) deve ter um evento de auditoria correspondente com campos obrigatórios: identidade do ator, timestamp com fuso horário, recurso afetado, ação executada, resultado, identificador de correlação. Esse esquema é definido antes da implementação, não depois.

3. Configure CloudTrail multi-região com validação de integridade e destino imutável: CloudTrail habilitado em todas as regiões, com log file integrity validation ativado, enviando para bucket S3 em conta de log archive separada com Object Lock em modo COMPLIANCE e política de bucket que nega deleção mesmo a administradores. A chave KMS que cifra os logs deve ter política que

proíbe deleção da chave antes do prazo de retenção.

4. Implemente hierarquia de chaves com envelope encryption e auditoria de uso: Dados sensíveis cifrados com DEK gerada pelo KMS; DEK cifrada pela CMK; cada chamada ao KMS auditada no CloudTrail. Para operações de alta criticidade (assinatura de transações, criptogramas de cartão), use CloudHSM com custódia exclusiva do banco. Rotação automática de chaves habilitada com período definido pela política de segurança.

5. Integre Security Hub, GuardDuty e Config em ciclo de detecção e resposta: Findings do Security Hub, alertas do GuardDuty e desvios do Config devem alimentar um processo de resposta com SLA definido por severidade. Não basta detectar, cada finding precisa de owner, prazo e evidência de remediação. Isso fecha o ciclo: prevenção, detecção, evidência de resposta.

Perguntas frequentes sobre segurança como evidência

KMS ou CloudHSM? Como decidir?

KMS com CMKs é suficiente para a grande maioria dos casos: envelope encryption de dados em repouso, assinatura de tokens, proteção de secrets. CloudHSM é necessário quando o banco precisa de custódia exclusiva do material de chave (a AWS não pode ter acesso nem em hipótese), quando a operação exige certificação FIPS 140-2 Level 3 com HSM dedicado, ou quando o esquema de bandeira ou o BACEN exige HSM próprio para geração de criptogramas. O custo operacional do CloudHSM é significativamente maior, use onde a regulação ou o modelo de ameaça justifica.

Por quanto tempo preciso reter logs de CloudTrail?

Depende da jurisdição e do tipo de dado. Para evidência de PLD no Brasil, a Circular BACEN 3.978/2020 exige retenção de registros por no mínimo cinco anos. Para dados pessoais sob LGPD, o prazo varia conforme a base legal do tratamento. Para operações de cartão sob PCI-DSS, o mínimo é um ano com três meses online. A recomendação prática é definir classes de retenção por tipo de evento, operacional (1 ano), regulatório (5 anos), jurídico (prazo prescricional aplicável), e configurar lifecycle policies no S3 para transição automática entre classes de storage, mantendo imutabilidade em todas.

Como provar que um log não foi adulterado?

CloudTrail com log file integrity validation gera um arquivo de digest a cada hora, assinado com chave RSA gerenciada pela AWS, que encadeia todos os arquivos de log do período. Para verificar integridade, você executa aws cloudtrail validate-logs e o serviço reconstrói a cadeia de hashes. Se qualquer arquivo foi modificado ou deletado, a validação falha. Combinado com S3 Object Lock (que impede modificação física) e destino em conta separada (que impede acesso por administradores da conta de produção), você tem três camadas independentes de garantia de integridade.

Segurança como evidência: o padrão mínimo para arquitetura bancária séria

A distinção entre segurança como opinião e segurança como evidência é, no fundo, uma distinção de maturidade arquitetural. Bancos que tratam conformidade como restrição de design desde o primeiro diagrama, embutindo auditabilidade, imutabilidade e correlacionabilidade em cada camada, constroem sistemas que sobrevivem a auditorias, a incidentes e à passagem do tempo. Bancos que tratam segurança como checklist final descobrem, invariavelmente, que o custo de

reescrever sob pressão é ordens de magnitude maior do que tê-la embutido desde o início. O arquiteto que consegue subir ao penthouse e traduzir 'provar quem aprovou' em decisões concretas de IAM, KMS, CloudTrail e Config, e descer à sala de máquinas e implementá-las de forma que o regulador possa verificar, é o arquiteto que o banco precisa.

13 - A arquitetura só existe de verdade em produção

Uma arquitetura que não se observa, não se opera e não se recupera é uma hipótese. Em sistemas financeiros, produção é onde decisões viram consequência, e SLOs, resiliência e FinOps fazem parte do desenho.

Uma arquitetura existe no papel, no diagrama, na apresentação para o comitê, mas só se torna real no momento em que um cliente tenta fazer um Pix às 23h47 de uma sexta-feira e o sistema decide, em milissegundos, se aquela transação completa ou falha. Produção não é o destino final do design: é onde cada decisão arquitetural vira consequência mensurável, regulatória e financeira. Neste capítulo fecho a Parte IV com a tese que orienta tudo o que veio antes: arquitetura que não se observa, não se opera e não se recupera é hipótese, e hipóteses não pagam salário de ninguém.

O SLO que o cliente experimenta não é o SLO do serviço

Existe um erro de raciocínio que encontro repetidamente em revisões de arquitetura: equipes exibem dashboards com 99,9% de disponibilidade por serviço e interpretam isso como evidência de saúde do sistema. Não é. Um Pix bem-sucedido atravessa, em média, entre oito e quinze componentes internos: API gateway, serviço de autenticação, validador de limite, motor de fraude, integrador com o SPI do Banco Central, ledger, notificador, reconciliador. Se cada um desses componentes tem 99,9% de disponibilidade independente e são estatisticamente independentes entre si (o que já é uma hipótese otimista), a disponibilidade composta da jornada é, na melhor das hipóteses, algo próximo de 99,9%^n, e com dez componentes isso cai para aproximadamente 99,0%. Esse número ainda parece alto até você calcular que 1% de falha em um banco com um milhão de transações diárias representa dez mil clientes impactados por dia.

A mudança que proponho é estrutural: defina SLOs de jornada, não de serviço. O SLO relevante é: "99,5% dos pagamentos Pix iniciados completam com sucesso em menos de 8 segundos, medidos do lado do cliente, em janelas de 5 minutos." Esse SLO atravessa domínios, inclui dependências externas (bandeira, parceiro, BACEN) e é exatamente o que o regulador e o cliente medem, mesmo que nunca usem esse vocabulário.

Isso muda profundamente como você instrumenta o sistema. Em vez de alertas por serviço, você precisa de traces distribuídos por jornada, com spans nomeados por etapa de negócio, não por nome de microsserviço. No AWS, isso significa X-Ray ou OpenTelemetry com atributos de domínio propagados via context propagation, correlacionados no CloudWatch ou em uma plataforma de observabilidade dedicada. O erro de budget de SLO precisa ser calculado sobre a jornada, não sobre o componente. Quando o error budget da jornada Pix começa a queimar mais rápido do que o esperado numa terça-feira de manhã, o arquiteto precisa saber disso antes do cliente reclamar, e antes do Banco Central abrir um chamado.

Perguntas que o arquiteto faz antes do incidente

- Parceiro, BaaS ou bandeira cai: meu sistema degrada graciosamente ou para junto? Qual

jornada continua operando em modo degradado e qual recusa com mensagem clara?

• Um evento chega duplicado ou fora de ordem: o ledger mantém a consistência? O processador de eventos é idempotente por contrato, não por esperança?

• Pico no dia de pagamento de salários: o que enfileira, o que rejeita com backpressure explícito e o que nunca pode ser descartado? Há filas por criticidade de jornada?

• Deploy falha em produção: qual é o tempo de volta? O rollback é automático ou depende de alguém acordar? O blast radius está contido a um domínio?

• O modelo de IA que avalia crédito ou fraude degrada ou fica indisponível: existe fallback determinístico documentado, testado e acionável sem aprovação manual?

• O runbook do incidente mais provável foi executado em simulação nos últimos 90 dias? Alguém da equipe atual o executou, não apenas o autor original?

Resiliência é o que sobra depois que o diagrama encontra a realidade

Quando subo ao andar executivo para discutir continuidade de negócio, uso a linguagem de risco: probabilidade de impacto, janela de exposição, custo de indisponibilidade por hora. Quando desço à sala de máquinas para implementar essa mesma conversa, a linguagem muda para circuit breakers, dead-letter queues, retry com exponential backoff e jitter, bulkheads por domínio, e chaos engineering agendado. A habilidade do arquiteto está em manter essas duas conversas conectadas, e em garantir que a decisão técnica de "usar SQS com DLQ e visibilidade timeout de 30 segundos" seja a implementação direta da decisão de negócio de "nenhuma transação de pagamento pode ser silenciosamente descartada".

Resiliência em sistemas financeiros tem três dimensões que precisam ser projetadas explicitamente. A primeira é contenção de blast radius: quando um componente falha, o dano não se propaga para domínios adjacentes. No AWS, isso se traduz em contas separadas por domínio (ou pelo menos por criticidade), VPCs com peering controlado, e filas SQS ou tópicos SNS como fronteiras de isolamento entre serviços. A segunda dimensão é degradação graciosa com contrato explícito: cada jornada precisa ter um modo degradado documentado, não improvisado durante o incidente. Uma proposta de crédito pode ser enfileirada para processamento assíncrono se o motor de decisão estiver lento; um Pix não pode ser silenciosamente atrasado sem notificação ao cliente e ao sistema de monitoramento. A terceira dimensão é recuperação verificável: não basta ter um plano de recuperação; ele precisa ser executado em simulação regularmente, com métricas de tempo de recuperação registradas e comparadas ao RTO contratado.

O AWS Fault Injection Simulator (FIS) é a ferramenta que uso para tornar o chaos engineering parte do ciclo de desenvolvimento, não um evento especial. Injetar latência no serviço de autenticação numa tarde de quarta-feira, em ambiente de staging com tráfego espelhado, revela dependências que nenhum diagrama captura. Quando o resultado do experimento contradiz o diagrama, o diagrama está errado, e é melhor descobrir isso na quarta do que na sexta às 23h47.

> A arquitetura mais bonita é a que o time opera às 3h: Depois de dezesseis anos, aprendi a avaliar uma arquitetura com uma pergunta simples: se o engenheiro de plantão for acordado às 3h da manhã com um alerta, sem contexto, com sono e pressão, ele consegue diagnosticar e mitigar o problema em menos de quinze minutos usando apenas o runbook e os dashboards disponíveis? Se a resposta for não, se o diagnóstico exigir conhecimento tácito de quem desenhou o sistema, ou se o runbook assumir que a pessoa está descansada e com tempo para pensar, então a arquitetura não está pronta para produção, independentemente de quantos noves de disponibilidade o design promete. Resiliência não é diagrama. É runbook que funciona com sono e pressão, executado por alguém que entrou na equipe há três meses.

Esse é o teste real.

Observabilidade como instrumento de decisão, não gráfico bonito

Observabilidade em sistemas financeiros tem uma exigência que vai além do que a maioria das plataformas de monitoramento endereça por padrão: ela precisa ser auditável e correlacionável com eventos de negócio. Quando o Banco Central ou um auditor externo questiona por que uma determinada transação foi processada com 4,2 segundos de latência numa janela específica, a resposta não pode ser "não temos granularidade suficiente nos logs". Cada evento de negócio relevante, início de jornada, decisão de crédito, liquidação, reversão, precisa ter um trace ID correlacionado, um timestamp com precisão de milissegundos, e um contexto de negócio (ID da transação, domínio, tipo de jornada) que permita reconstrução post-mortem.

No AWS, a combinação que uso como ponto de partida é: CloudWatch Logs Insights para correlação ad-hoc, X-Ray ou OpenTelemetry com ADOT para traces distribuídos, CloudWatch Metrics com dimensões de negócio para SLOs de jornada, e EventBridge como barramento de auditoria para eventos de domínio. Para bancos com volume alto, a ingestão no Amazon OpenSearch ou em uma plataforma dedicada como Datadog ou Grafana Cloud frequentemente se justifica pelo custo de operação reduzido e pela capacidade de correlação em tempo real.

O ponto que mais enfatizo com equipes: observabilidade precisa ser instrumentada no design, não adicionada depois. Quando um serviço é desenhado sem considerar quais métricas de negócio ele precisa emitir, o resultado é um sistema que monitora a si mesmo, latência de endpoint, uso de CPU, erros HTTP, mas não monitora o que o negócio precisa saber. A pergunta que faço em cada revisão de design de serviço é: "se este serviço começar a produzir resultados incorretos sem falhar tecnicamente, como saberemos?" Essa pergunta, quando respondida honestamente, define a instrumentação necessária, e frequentemente revela que o serviço precisa emitir eventos de negócio com semântica explícita, não apenas logs técnicos.

FinOps: custo por transação como métrica de produto

Existe uma conversa que raramente acontece em revisões de arquitetura de bancos e que deveria ser obrigatória: qual é o custo unitário de cada jornada que o sistema executa? Não o custo total da infraestrutura, esse número é importante para o CFO mas não orienta decisão de design. O número que importa para o arquiteto é: quanto custa processar um Pix, aprovar uma proposta de crédito, ou gerar um extrato? Quando esse número é visível em tempo real, ele se torna uma métrica de produto, e começa a orientar decisões de design da mesma forma que latência e disponibilidade.

A tabela a seguir apresenta o framework FinOps que aplico em contextos bancários na AWS, organizando custo por domínio de negócio, com granularidade de transação e atribuição de responsabilidade por equipe. O princípio central é que cada domínio é dono do seu custo: não existe "custo de infraestrutura" como categoria opaca gerenciada centralmente. Quando a equipe de pagamentos vê que o custo por Pix aumentou 15% depois de um deploy, essa informação é tão relevante quanto um aumento de latência.

Pay-per-use e right-sizing são os dois alavancas principais. No AWS, isso significa preferir Lambda e Fargate para workloads com picos imprevisíveis (como processamento de boletos em datas de vencimento), e EC2 com Savings Plans ou Reserved Instances para workloads com baseline estável e previsível (como o ledger de posições). O erro que vejo com frequência é o inverso: instâncias reservadas para workloads que variam 10x entre pico e vale, e Lambda para processamento batch que seria mais barato em EC2 Spot. A decisão de compute não é apenas

técnica, ela tem impacto direto no custo unitário da jornada, e portanto no modelo de negócio.

FinOps em banco não é sobre gastar menos. É sobre gastar com visibilidade suficiente para saber se o modelo de negócio é sustentável, e para identificar, antes do produto escalar, se o custo unitário está na trajetória certa.

Como estruturar SLOs de jornada na prática

1. Mapeie as jornadas críticas, não os serviços: Liste as cinco a dez jornadas que, se degradadas, causam impacto regulatório, financeiro ou de reputação imediato. Pix, TED, proposta de crédito, autenticação, extrato são candidatos típicos. Cada jornada terá seu próprio SLO, disponibilidade, latência e correção.

2. Instrumente o trace de ponta a ponta com contexto de negócio: Propague trace ID e atributos de negócio (tipo de jornada, domínio, criticidade) via headers HTTP e metadados de mensagem SQS/EventBridge. O span raiz deve ser nomeado pela jornada de negócio, não pelo serviço técnico.

3. Calcule error budget por jornada e torne-o visível: Em CloudWatch ou na plataforma de observabilidade escolhida, crie um dashboard por jornada com error budget atual, taxa de queima (burn rate) e projeção para o fim da janela. Esse dashboard é o instrumento de decisão, não o dashboard de serviço.

4. Defina alertas de burn rate, não de threshold absoluto: Um alerta que dispara quando a latência passa de 500ms é reativo. Um alerta que dispara quando o error budget da jornada Pix está queimando a 14x a taxa normal é preditivo, ele avisa que, se nada mudar, o SLO será violado em menos de uma hora.

5. Valide o runbook de cada jornada crítica em simulação trimestral: Agende game days trimestrais com injeção de falha via AWS FIS. O critério de sucesso não é "o sistema sobreviveu", é "o engenheiro de plantão diagnosticou e mitigou dentro do RTO usando apenas o runbook, sem ajuda de quem desenhou o sistema".

FinOps em banco: custo como métrica de produto

Critério Abordagem ingênua Abordagem FinOps madura
--- --- ---
Visão de custo Fatura total no fim do mês Custo por transação e por jornada, em tempo real
Capacidade Provisionar para o pico e esquecer Pay-per-use + right-sizing contínuo
Decisão de runtime Tudo no mesmo cluster sempre ligado Serverless no irregular, container no constante
Dono do custo Infra, no fim da fila Cada domínio dono do seu custo, observável

Perguntas frequentes sobre operação de arquiteturas financeiras

SLO de jornada é diferente de SLA contratual com o cliente?

Sim, e a distinção importa. O SLA é o compromisso externo, frequentemente conservador e com penalidades contratuais. O SLO é o objetivo interno, mais rigoroso, que serve como sinal de alerta antes que o SLA seja violado. O SLO deve ser mais exigente que o SLA por uma margem que dê tempo de reação, tipicamente, se o SLA é 99,5%, o SLO interno é 99,7% ou 99,8%.

Como atribuir custo por jornada quando a infraestrutura é compartilhada?

Use tags de custo (Cost Allocation Tags no AWS) com dimensões de domínio e tipo de jornada propagadas até o nível de recurso. Para infraestrutura genuinamente compartilhada (como um cluster EKS multi-tenant), use métricas de utilização proporcional para distribuir o custo, não é

perfeito, mas é suficientemente preciso para orientar decisões de design e identificar jornadas com custo unitário fora de controle.

Chaos engineering é viável em bancos com regulação rígida?

Sim, com escopo e governança adequados. Comece em staging com tráfego espelhado (shadow traffic), documente cada experimento como um caso de teste de resiliência, e mantenha o histórico de execução como evidência de due diligence operacional. Reguladores como o Banco Central valorizam evidência de que o banco testa ativamente sua capacidade de recuperação, o oposto de chaos engineering não regulado é um banco que descobre suas fragilidades durante um incidente real.

Produção é onde a arquitetura prova seu valor

Ao longo deste capítulo, o elevador subiu e desceu várias vezes: do risco regulatório de indisponibilidade do Pix até a configuração de visibility timeout no SQS; da conversa com o CFO sobre custo unitário de transação até a tag de alocação de custo no recurso AWS; da definição de RTO no contrato de continuidade até o game day trimestral com AWS FIS. Cada descida à sala de máquinas foi motivada por uma decisão do andar executivo, e cada subida levou evidência técnica para uma conversa de negócio. Esse é o trabalho do arquiteto em produção: não apenas desenhar sistemas que funcionam, mas garantir que funcionam de forma observável, recuperável e economicamente sustentável, e que o time que os opera às 3h da manhã tem as ferramentas e os runbooks para provar isso.

PARTE V

Subindo o Elevador: Decisão e Transformação

De volta ao penthouse: como vender opções, registrar decisões, criar mecanismos que sobrevivem à reunião e liderar transformação sem transformar tudo em PowerPoint.

14 - Vender opções e registrar decisões

Arquitetura cria opções, e opções têm valor sob incerteza, essa é a tese mais subestimada de Hohpe. Subir o elevador é vender esse valor; descer é registrar a decisão em ADR e transformá-la em consequência.

Toda decisão de arquitetura é, no fundo, uma aposta sobre o futuro, e em um banco, onde o regulador muda as regras, o mercado lança novos concorrentes e a tecnologia se reinventa a cada ciclo, apostar sem hedge é imprudência profissional. Este capítulo abre a Parte V com a tese que considero a mais subestimada de Gregor Hohpe: arquitetura não entrega software, entrega opções, e opções têm valor financeiro real, mensurável, especialmente sob incerteza alta. Subir o elevador é saber vender esse valor ao penthouse; descer é transformar a decisão em ADR e em consequência concreta.

Arquitetura como portfólio de opções

Quem trabalhou com derivativos sabe que uma opção de compra (call) dá ao titular o direito, não a obrigação, de comprar um ativo a um preço fixo no futuro. Você paga um prêmio hoje por esse direito. Se o ativo subir além do strike, a opção vira lucro; se não subir, você perde apenas o

prêmio, não o capital total. A lógica é exatamente a mesma quando decidimos desacoplar dois domínios via eventos em vez de chamada síncrona direta.

O prêmio é real: mais infraestrutura, mais contratos de evento para manter, mais superfície de observabilidade. Ninguém deveria fingir que esse custo não existe. Mas o que você compra com esse prêmio é o direito de trocar a implementação de um domínio sem reescrever os outros: o direito de escalar o processamento de transações de forma independente do motor de notificações, o direito de migrar o core de pagamentos para um novo provedor sem interromper o fluxo de antifraude.

Em um ambiente de baixa incerteza, digamos, um sistema interno de folha de pagamento de uma empresa madura, requisitos estáveis há dez anos, essa opção vale pouco. O prêmio provavelmente não se justifica. Otimize custo, simplifique, acople. Mas em um banco brasileiro operando sob BACEN, LGPD, Pix, Open Finance, pressão de fintechs e ciclos regulatórios imprevisíveis, a incerteza é estrutural. Aqui, pagar pelo opcional não é ouro de tolo, é gestão de risco aplicada à engenharia.

O problema é que esse argumento raramente aparece assim formulado nas discussões técnicas. Ele aparece como preferência pessoal: 'eu prefiro eventos', 'microserviços são mais modernos'. Quando o debate chega ao penthouse como gosto técnico, ele perde. Quando chega como gestão de portfólio de opções sob incerteza, ele encontra interlocutores naturais, porque essa é a língua nativa de quem decide alocação de capital.

> A tese central deste capítulo: Arquitetura vende opções. Em baixa incerteza, opções valem pouco, otimize custo e simplifique. Em banco (alta incerteza regulatória, competitiva e tecnológica), opções valem muito, e saber quando pagar pelo opcional e quando não pagar é metade do trabalho do arquiteto. O outro metade é registrar essa decisão de forma que ela sobreviva à rotatividade de pessoas e à pressão do próximo sprint.

Subindo o elevador com a linguagem certa

Hohpe descreve o arquiteto como alguém que transita entre o penthouse, onde vivem estratégia, risco e capital, e a sala de máquinas, onde vivem código, latência e pipelines. A maioria dos arquitetos que conheço é tecnicamente sólida na sala de máquinas, mas perde contexto ao subir. Chegam ao penthouse falando em throughput, eventual consistency e idempotency keys, e saem sem budget, sem prioridade e sem patrocínio.

A virada acontece quando o arquiteto aprende a traduzir decisão técnica em consequência de negócio: e, mais especificamente, quando aprende a enquadrar essa consequência em termos de risco e opcionalidade. Considere dois enquadramentos para a mesma decisão de desacoplar o domínio de câmbio do domínio de compliance via EventBridge:

Enquadramento técnico: 'Vamos usar eventos assíncronos para reduzir acoplamento temporal entre os serviços.'

Enquadramento de opção: 'Estamos pagando um prêmio estimado de X semanas de esforço adicional para comprar o direito de substituir o motor de compliance, por exemplo, ao onboarding de um novo parceiro regulatório, sem precisar reescrever o fluxo de câmbio. Dado que temos pelo menos dois processos regulatórios abertos que podem exigir essa troca nos próximos dezoito meses, o prêmio parece justificado.'

O segundo enquadramento não é mais longo por acidente. Ele contém o prêmio (custo), o direito comprado (o que a opção permite), o gatilho (quando a opção seria exercida) e a probabilidade qualitativa de exercício. É uma linguagem que o CFO e o CRO reconhecem imediatamente.

Isso não significa que o arquiteto deve aprender a fazer planilhas de Black-Scholes. Significa que ele precisa desenvolver o hábito de perguntar, para cada decisão de desacoplamento: qual é o prêmio real? qual é o direito que estou comprando? em que cenários esse direito seria exercido? qual é a minha estimativa qualitativa de que esses cenários ocorram? Se não conseguir responder essas quatro perguntas, a decisão ainda não está madura para o penthouse.

> Quando a opção vale pouco, e quando vale muito: Em sistemas com requisitos estáveis e baixo risco regulatório, pagar o prêmio de desacoplamento é desperdício, simplifique e acople. Em bancos brasileiros, a combinação de BACEN com poder normativo amplo, ciclos de Open Finance ainda em evolução, pressão competitiva de fintechs e incerteza tecnológica (IA generativa redefinindo produtos em tempo real) cria uma das maiores densidades de incerteza que já encontrei em qualquer setor. Aqui, opções arquiteturais são hedge genuíno, não ouro de tolo.

O ADR como mecanismo, não cerimônia

Depois de subir o elevador e vender a opção, o arquiteto precisa descer e registrá-la. É aqui que o Architecture Decision Record (ADR) entra, não como burocracia, não como documentação para auditoria, mas como mecanismo que impede o sistema de perder memória.

Um ADR bem escrito tem cinco elementos obrigatórios. O contexto descreve o estado do mundo no momento da decisão: quais eram as restrições regulatórias vigentes, qual era a carga esperada, quais eram as dependências existentes. Sem contexto, a decisão parece arbitrária para quem chega depois. As opções consideradas listam as alternativas reais que foram avaliadas, e aqui está o elemento mais negligenciado: o que foi rejeitado e por quê. A opção descartada hoje é a tentação de amanhã. Quando um engenheiro novo chega e propõe exatamente o que foi rejeitado há dezoito meses, o ADR é o que evita refazer o debate do zero, ou pior, refazer o erro.

A decisão em si deve ser enunciada de forma inequívoca: não 'vamos considerar Aurora', mas 'usaremos Aurora PostgreSQL como ledger primário do core bancário'. E as consequências: o campo mais importante e o mais frequentemente omitido, devem listar o que muda no sistema como resultado direto dessa decisão: quais capacidades são habilitadas, quais são inibidas, quais débitos técnicos são aceitos conscientemente, quais revisões futuras são antecipadas.

O quinto elemento, que adiciono por experiência própria em sistemas financeiros, é o gatilho de revisão: em que condições esta decisão deve ser reaberta? Uma mudança de regulação? Um crescimento de volume acima de determinado threshold? A disponibilidade de um novo serviço gerenciado? Sem esse gatilho, o ADR vira arqueologia, encontrado por acidente, nunca atualizado, irrelevante. Com ele, o ADR vira parte do processo de governança viva da plataforma.

Como mostra a matriz de decisão a seguir, um ADR real para a escolha do ledger no core bancário: Aurora versus DynamoDB, não é um documento simples. Ele carrega trade-offs de consistência, modelo de custo, capacidade operacional e opcionalidade futura que só fazem sentido quando registrados juntos, no momento em que foram pesados.

Como estruturar um ADR em sistemas financeiros

1. 1. Contexto: Descreva o estado do mundo no momento da decisão: restrições regulatórias (BACEN, LGPD), volume esperado, dependências existentes, pressões de prazo. Sem contexto, a decisão parece arbitrária para quem chega depois.

2. 2. Opções consideradas (incluindo as rejeitadas): Liste todas as alternativas reais avaliadas. Registre explicitamente o que foi rejeitado e por quê, a opção descartada hoje é a tentação de amanhã e o ADR é o que evita refazer o debate do zero.

3. 3. Decisão: Enuncie a decisão de forma inequívoca e no presente: não 'vamos considerar X', mas 'usaremos X para Y'. Ambiguidade aqui gera reinterpretação divergente ao longo do tempo.

4. 4. Consequências: Liste o que muda no sistema: capacidades habilitadas, capacidades inibidas, débitos técnicos aceitos conscientemente, revisões futuras antecipadas. Este é o campo mais importante e o mais frequentemente omitido.

5. 5. Gatilho de revisão: Defina explicitamente em que condições esta decisão deve ser reaberta: mudança regulatória específica, crescimento de volume acima de um threshold, disponibilidade de novo serviço gerenciado. Sem gatilho, o ADR vira arqueologia.

O ADR como artefato de governança viva

Há uma objeção que ouço frequentemente: 'ADRs ficam desatualizados e ninguém lê'. É verdade, quando tratados como documentação. Quando tratados como mecanismo de governança, o comportamento muda.

A diferença está em três práticas operacionais. Primeira: o ADR vive no repositório de código, não em uma wiki separada. Ele é versionado junto com o sistema que ele governa. Quando o código muda de forma que contradiz o ADR, isso é visível, e deve ser um sinal de que ou o ADR precisa ser revisado ou a mudança de código precisa de justificativa explícita. Segunda: o ADR tem um status explícito: proposto, aceito, substituído, obsoleto. Um ADR substituído não é deletado; ele é marcado como substituído e aponta para o ADR sucessor, preservando a cadeia de raciocínio. Terceira: o gatilho de revisão é monitorado. Se o gatilho é 'volume de transações acima de 50 mil TPS' (estimativa ilustrativa), esse threshold deve estar em um dashboard, quando é atingido, o ADR entra automaticamente na agenda de revisão de arquitetura.

Em bancos, essa disciplina tem uma dimensão adicional: auditabilidade. O BACEN e auditores externos não perguntam apenas o que o sistema faz, perguntam por que foi construído assim, quais alternativas foram consideradas e quais riscos foram conscientemente aceitos. Um portfólio de ADRs bem mantido é a resposta mais honesta e mais defensável a essa pergunta. Já vi equipes gastarem semanas reconstruindo a justificativa de uma decisão de arquitetura para uma auditoria, tempo que seria zero se o ADR existisse.

A matriz de decisão que apresento a seguir: Aurora versus DynamoDB para o ledger do core bancário, é um exemplo real do tipo de trade-off que merece esse nível de registro. Não é uma decisão trivial: ela carrega implicações de consistência transacional, modelo de custo em escala, capacidade operacional da equipe e, crucialmente, opcionalidade futura para migração ou expansão.

[DECISION MATRIX] Exemplo de ADR: ledger no core: Aurora vs. DynamoDB

A matriz a seguir materializa os princípios discutidos neste capítulo em um caso concreto: a escolha do banco de dados para o ledger primário do core bancário. Esta é precisamente a categoria de decisão que não pode ser tomada por preferência técnica, ela precisa ser tomada como gestão de portfólio de opções, com prêmio, direito e gatilho de revisão explícitos.

Observe, ao ler a matriz, como cada opção carrega não apenas características técnicas, mas consequências de negócio: o que cada escolha habilita e o que inibe no horizonte de dois a cinco anos. Observe também o que foi rejeitado e por quê: a tentação de usar DynamoDB pela escalabilidade horizontal é real, mas as consequências para consistência transacional e auditabilidade do ledger são consequências que o penthouse precisa entender antes de aprovar.

Este é o elevador em operação: a decisão técnica sobre banco de dados carrega implicações de risco operacional, custo regulatório de conformidade e opcionalidade estratégica que só fazem sentido quando apresentadas juntas, no mesmo artefato, para audiências nos dois andares.

Exemplo de ADR: ledger no core: Aurora vs. DynamoDB

Aurora PostgreSQL (relacional)

Pros

- Consistência forte e transações ACID nativas
- SQL maduro para conciliação e auditoria
- Modelo natural para dupla entrada

Cons

- Escala de escrita exige sharding/Limitless
- Custo cresce com volume sustentado

Verdict: Padrão para o ledger central, onde consistência forte e auditabilidade são inegociáveis.

DynamoDB (chave-valor)

Pros

- Escala horizontal e latência previsível
- Operação mínima, pay-per-use
- Ótimo para projeções de saldo de baixa latência

Cons

- Transações limitadas; modelar dupla entrada é trabalhoso
- Conciliação e queries analíticas mais difíceis

Verdict: Forte para projeções e leituras de alta escala, não para a escrituração contábil central.

> Make impact, not PowerPoint: Hohpe: Hohpe é direto: o objetivo do arquiteto é causar impacto, não produzir slides. Meça-se pelo que mudou no sistema, uma decisão registrada em ADR que foi efetivamente implementada, um debate que não precisou ser refeito, uma opção exercida no momento certo porque estava documentada. Um deck bonito que não muda nada é ruído. Um ADR de duas páginas que alinha dez engenheiros e um CTO por dezoito meses é alavancagem real.

Fechar o ciclo: da opção à consequência

O ciclo completo do arquiteto neste capítulo tem três movimentos. No primeiro, ele identifica a decisão como uma opção, com prêmio, direito e gatilho. No segundo, ele sobe o elevador e vende essa opção ao penthouse na linguagem de risco e capital que o penthouse entende. No terceiro, ele desce e registra a decisão em ADR com contexto, alternativas rejeitadas, consequências e gatilho de revisão.

O que fecha o ciclo é o acompanhamento. Uma opção vendida e não monitorada é uma promessa não cumprida. O arquiteto precisa garantir que os gatilhos de revisão estejam instrumentados, em dashboards, em alertas, em processos de governança, e que quando um gatilho for acionado, o ADR seja revisado com a mesma seriedade com que foi criado.

Em sistemas financeiros, esse fechamento de ciclo tem uma dimensão adicional que não posso deixar de mencionar: a reversibilidade. Algumas decisões de arquitetura são facilmente reversíveis, trocar uma biblioteca de serialização, mudar um parâmetro de configuração. Outras

são praticamente irreversíveis no horizonte relevante, escolher o modelo de dados do ledger, definir a topologia de eventos do core. O ADR deve registrar explicitamente a reversibilidade estimada da decisão, porque isso afeta diretamente quanto prêmio vale pagar pela opção de não cometê-la.

Quando o arquiteto domina esse ciclo, identificar opção, vender ao penthouse, registrar em ADR, monitorar gatilhos, revisar quando acionado, ele para de ser o técnico que o negócio tolera e começa a ser o parceiro que o negócio procura. Essa é a promessa do elevador: não que você saiba tudo sobre todos os andares, mas que você seja capaz de traduzir consequência entre eles com precisão e com responsabilidade.

Pontos-chave do capítulo

- Arquitetura entrega opções, não apenas software, e opções têm valor financeiro real, especialmente sob incerteza alta como a de bancos brasileiros.
- Toda decisão de desacoplamento tem um prêmio (custo hoje) e um direito (capacidade futura); articular ambos é o que torna o argumento técnico compreensível no penthouse.
- O ADR não é documentação, é o mecanismo que impede o sistema de perder memória e que evita refazer debates ou reverter decisões sem entender suas consequências.
- O campo mais importante do ADR são as consequências, e o mais negligenciado. O segundo mais importante é o que foi rejeitado e por quê.
- O gatilho de revisão transforma o ADR de arqueologia em governança viva, ele deve ser instrumentado e monitorado, não apenas escrito.
- Meça-se pelo impacto no sistema, não pelos slides produzidos. Um ADR que alinha uma equipe por dezoito meses é mais valioso do que qualquer apresentação executiva que não muda nada.

O arquiteto que vende opções e registra decisões

A maturidade de um arquiteto de sistemas financeiros não se mede pela complexidade das soluções que propõe, mas pela qualidade das opções que preserva e pela clareza com que registra as que descarta. Subir o elevador com a linguagem de opções e descer com ADRs bem escritos é o que separa o arquiteto que deixa um legado do arquiteto que deixa uma dívida.

15 - Mecanismos e a liderança da mudança

Decisão sem mecanismo evapora. Transformar um banco não é desenhar o estado final, é criar os mecanismos que fazem a organização decidir melhor de forma sustentável, andar após andar.

Toda transformação arquitetural em banco começa com uma reunião entusiasmada e termina, na maioria das vezes, exatamente onde começou, porque entusiasmo não é mecanismo. O arquiteto sênior que entende isso para de perguntar 'qual é a decisão certa?' e começa a perguntar 'qual é o mecanismo que faz a decisão certa acontecer sozinha, depois que eu sair da sala?' Essa mudança de pergunta é a diferença entre escrever documentos e mudar organizações.

Decisão sem mecanismo é intenção disfarçada de arquitetura

Existe uma ilusão recorrente nos programas de transformação bancária: a de que decidir é suficiente. A liderança aprova o roadmap, o arquiteto apresenta o diagrama de referência, o comitê

assina a ata, e todos saem da sala convencidos de que algo mudou. Não mudou. O que mudou foi o registro de uma intenção.

Gregor Hohpe tem uma frase que uso como calibrador toda vez que avalio um programa de transformação: slow chaos is not order. Processo lento aplicado sobre caos não produz ordem, produz caos lento, que é ainda mais difícil de diagnosticar porque parece organizado. Quando um banco decide que 'todo domínio vai publicar eventos com schema versionado e idempotência garantida' sem criar nenhum mecanismo de adoção, o que acontece nos próximos seis meses é exatamente isso: cada squad interpreta a decisão de um jeito, alguns publicam eventos sem schema, outros criam schemas incompatíveis, e o catálogo de eventos vira um documento desatualizado que ninguém confia. A decisão existiu. A ordem não chegou.

Mecanismo é aquilo que continua funcionando depois que a reunião acabou e o entusiasmo passou. É o template de serviço que já vem com observabilidade embutida, então o caminho certo é também o caminho mais fácil. É a policy-as-code no pipeline de CI que bloqueia o deploy de um evento sem schema registrado, então a conformidade não depende de disciplina individual, depende de física. É o dashboard de DLQ por domínio com dono identificado, então o problema de um consumidor que não processa mensagens fica visível antes de virar incidente. É a revisão trimestral de incidentes que retroalimenta o template, então o aprendizado coletivo se acumula em vez de evaporar.

A distinção que faço para times com quem trabalho é direta: documento convence na reunião; mecanismo convence todo dia. E em banco, onde a rotatividade de liderança é alta e os ciclos de prioridade mudam a cada trimestre, só sobrevive o que foi institucionalizado em mecanismo.

> A transformação que durou: Em todos os programas de modernização que acompanhei de perto, a variável que melhor prediz se a transformação vai durar não é o orçamento, não é o patrocinador executivo e não é a qualidade do roadmap. É se o programa criou mecanismos que sobrevivem à troca de liderança. Vi iniciativas excelentes morrerem em seis meses porque dependiam do entusiasmo de um VP que saiu. Vi iniciativas modestas durarem anos porque alguém teve o cuidado de embutir as decisões em templates, pipelines e políticas automatizadas. O documento convence quem está na sala. O mecanismo convence quem nunca esteve.

De decisão a mecanismo: o ciclo completo

1. Decisão arquitetural: Todo domínio publica eventos com schema versionado, contrato de idempotência explícito e chave de correlação obrigatória. Essa decisão fica registrada como ADR (Architecture Decision Record) com contexto, consequências e data de revisão, não apenas como slide de apresentação.

2. Mecanismo de adoção: Um template de serviço no repositório interno já inclui o padrão Outbox implementado, o registro de schema no AWS Glue Schema Registry configurado, e um consumidor idempotente com tabela de deduplicação no DynamoDB pronto para uso. O time não precisa saber a teoria, precisa clonar o template.

3. Mecanismo de governança: Uma policy-as-code no pipeline de CI, implementada com AWS Config Rules e checagens no GitHub Actions, bloqueia o deploy de qualquer evento cujo schema não esteja registrado no catálogo central. O bloqueio é automático: não há comitê de aprovação, não há exceção manual sem registro de justificativa auditável.

4. Mecanismo de visibilidade: Um catálogo de eventos, alimentado automaticamente pelo Schema Registry, expõe produtor, consumidores, versão ativa e histórico de versões. Um dashboard no

Amazon CloudWatch mostra, por domínio, a taxa de mensagens em DLQ com o nome do time responsável. Problema visível tem dono. Problema invisível vira dívida silenciosa.

5. Mecanismo de aprendizado: Uma revisão trimestral de incidentes, com pauta estruturada e participação dos domain leads, analisa os incidentes do período e identifica padrões que devem retroalimentar o template. Se três incidentes no trimestre envolveram consumidores que não tratavam mensagens poison-pill, o template é atualizado com dead-letter handling explícito antes do próximo ciclo. O aprendizado se acumula na ferramenta, não na cabeça de uma pessoa.

O elevador entre o penthouse e a sala de máquinas: dois andares que precisam avançar juntos

A tensão que o arquiteto bancário enfrenta todo dia tem uma geometria específica: o penthouse quer velocidade, inovação e time-to-market; a sala de máquinas carrega sistemas críticos que processam liquidações, calculam reservas e reportam ao BACEN, sistemas que não podem parar, não podem perder dados e não podem introduzir inconsistência contábil. Quando o arquiteto não cria mecanismos que deixam os dois andares avançarem juntos, o que acontece é previsível: ou o penthouse atropela a sala de máquinas com mudanças que geram incidentes, ou a sala de máquinas paralisa o penthouse com processos de aprovação que levam semanas.

A saída não é escolher um andar. A saída é criar mecanismos que reduzem o atrito do caminho certo em ambos os andares simultaneamente. No penthouse, isso significa que o arquiteto traduz risco técnico em linguagem de negócio, não 'temos acoplamento síncrono excessivo', mas 'cada nova integração aumenta a probabilidade de indisponibilidade do canal digital em X pontos percentuais' (estimativa, não medição). Na sala de máquinas, isso significa que o arquiteto cria abstrações que permitem ao time de legado evoluir sem reescrever tudo, strangler fig sobre o core bancário, eventos como camada de desacoplamento, APIs versionadas que isolam o contrato do consumidor da implementação interna.

O mecanismo que conecta os dois andares é o que chamo de trilha de confiança: um conjunto de práticas automatizadas que permite ao time de inovação mover rápido porque o time de plataforma garantiu que o piso é sólido. Testes de contrato automatizados que validam que a nova feature não quebra o consumidor legado. Feature flags com rollback automático baseado em métricas de negócio. Ambientes de staging com dados sintéticos que respeitam a LGPD e ao mesmo tempo são representativos o suficiente para validar comportamento financeiro. Esses mecanismos não eliminam a tensão entre os andares, a tensão é legítima e produtiva. Eles eliminam o atrito desnecessário que transforma tensão em paralisia.

Rido internamente quando ouço 'precisamos de um comitê de arquitetura mais ágil'. Comitê mais ágil é mais reunião. O que o banco precisa é de menos decisão centralizada e mais mecanismo distribuído, guardrails que permitem autonomia dentro de limites seguros, em vez de aprovação centralizada que cria gargalo.

Ownership como mecanismo: quem é dono do problema é dono da solução

Um dos mecanismos mais subestimados, e mais baratos de implementar, é a atribuição explícita e pública de ownership. Não ownership no sentido burocrático de 'responsável formal', mas no sentido operacional de 'quem acorda às 2h quando isso quebra e tem autonomia para corrigir'. A diferença é enorme.

Em sistemas financeiros distribuídos, o problema de ownership difuso é especialmente grave. Um

evento que percorre quatro domínios antes de atualizar o saldo do cliente tem quatro times que podem dizer 'não é comigo' quando a mensagem some na DLQ. Sem um mecanismo de ownership explícito, o incidente vira uma reunião de finger-pointing. Com um mecanismo de ownership explícito, o catálogo de eventos com dono identificado, o dashboard de DLQ que envia alerta direto para o canal do time responsável, o runbook que define o protocolo de escalação, o problema tem um endereço antes de acontecer.

No contexto da AWS, isso se traduz em práticas concretas: tags obrigatórias em todos os recursos com owner, domain e criticality, validadas por AWS Config; alarmes do CloudWatch configurados no template de serviço com roteamento automático para o canal do time no Slack ou no PagerDuty; runbooks armazenados no Systems Manager e linkados diretamente no alarme, quando o alerta dispara, o link para o runbook já está no corpo da notificação. O oncall não precisa procurar o que fazer: o mecanismo entrega o contexto junto com o problema.

Ownership sem autonomia, porém, é punição disfarçada de responsabilidade. O mecanismo só funciona se o time que é dono do evento também tem poder para alterar o schema, ajustar o consumidor, e fazer rollback sem precisar de aprovação de três comitês. Isso tem implicação direta na forma como o arquiteto desenha os limites de domínio: fronteiras de domínio devem coincidir com fronteiras de autonomia operacional. Quando não coincidem, o ownership é nominal e o mecanismo falha na primeira crise real.

Mecanismos que fazem a transformação durar

- Template de serviço com observabilidade, idempotência e schema embutidos: o caminho certo é o caminho mais fácil, não o mais disciplinado.
- Policy-as-code no CI/CD: conformidade depende de física, não de revisão manual. Bloqueio automático sem exceção não auditada.
- Catálogo de eventos com dono, versão e consumidores visíveis: problema sem endereço não tem solução, só tem reunião.
- Dashboard de DLQ por domínio com alerta roteado ao time responsável: visibilidade operacional não é opcional em sistema financeiro.
- Revisão trimestral de incidentes que retroalimenta o template: aprendizado coletivo acumulado na ferramenta, não na cabeça de uma pessoa.
- ADRs com data de revisão: decisão arquitetural que não tem prazo de revisão é dogma, não arquitetura.

Liderar sem PowerPoint: o arquiteto como criador de condições

Existe uma versão do arquiteto sênior que passa a maior parte do tempo produzindo apresentações para convencer stakeholders. Essa versão é necessária em doses certas, o penthouse precisa de contexto e narrativa. Mas quando o arquiteto passa mais tempo convencendo do que construindo mecanismos, ele se torna um gargalo de persuasão: nada avança sem sua presença, e quando ele sai da empresa, a transformação para.

A versão que eu defendo é diferente: o arquiteto sênior lidera criando condições para que as decisões certas aconteçam sem ele. Isso significa investir tempo em três atividades que raramente aparecem no job description mas que têm o maior retorno de longo prazo. Primeiro, construir e manter os templates: não delegar a construção do template para um time júnior e assinar embaixo, mas sujar a mão na implementação do padrão Outbox, entender onde o DynamoDB Streams cria complexidade inesperada, descobrir que o Schema Registry do Glue tem comportamento específico com schemas Avro que contêm campos opcionais. Segundo, instrumentar a

visibilidade: garantir que os dashboards existem, que os alarmes estão calibrados para o contexto financeiro (um alarme de latência que dispara para toda operação acima de 200ms em batch noturno é ruído; o mesmo alarme em transação PIX é crítico), e que o runbook está linkado onde o oncall vai procurar. Terceiro, facilitar o aprendizado coletivo: a revisão trimestral de incidentes não é uma reunião de post-mortem; é o mecanismo pelo qual o conhecimento tácito do time mais experiente se torna conhecimento explícito no template.

Essa abordagem tem um custo que preciso nomear: ela é mais lenta no curto prazo e menos visível para quem mede output por número de apresentações entregues. O arquiteto que passa três semanas refinando um template de serviço não tem slide novo para mostrar na reunião de steering. Mas seis meses depois, quando doze times onboardaram usando o template sem incidente de schema, o valor está lá, distribuído, silencioso e duradouro. Essa é a assinatura do trabalho de arquitetura que realmente muda organizações: não é barulhento, mas é permanente.

O elevador, nesse contexto, tem uma função específica: o arquiteto sobe ao penthouse para entender qual problema de negócio o mecanismo precisa resolver, qual é o risco regulatório que a policy-as-code está mitigando, qual é o custo operacional que o dashboard de DLQ está evitando, e desce à sala de máquinas para garantir que o mecanismo está implementado com a precisão técnica que o contexto financeiro exige. A viagem de ida e volta não é cerimônia; é o que garante que o mecanismo resolve o problema certo da forma certa.

Perguntas frequentes sobre mecanismos e transformação

Como convencer a liderança a investir em mecanismos quando o que ela quer ver é feature entregue?

Não venda mecanismo, venda o que o mecanismo evita. 'Vamos criar um template de serviço' não convence ninguém no penthouse. 'Cada onboarding de novo domínio sem template custa em média X semanas de retrabalho e foi a causa raiz de dois dos três incidentes do último trimestre' (estimativa baseada em dados reais do seu contexto) convence. Traduza mecanismo em risco evitado e velocidade recuperada.

E quando o time resiste ao template porque 'cada domínio é diferente'?

Resistência a template geralmente é resistência a perda de autonomia, não objeção técnica legítima. A resposta é design do template: ele deve ser opinativo nas partes que importam para segurança e conformidade (schema registry, idempotência, observabilidade) e extensível nas partes que variam por domínio. Se o template não tem pontos de extensão claros, o problema é do template, não do time. Reescreva o template antes de forçar adoção.

Como manter os mecanismos atualizados sem virar um gargalo de manutenção?

O mecanismo de aprendizado, a revisão trimestral que retroalimenta o template, precisa ter um dono de produto, não um dono técnico individual. O arquiteto facilita o processo e toma as decisões de design; o time contribui com os casos de uso reais. Versione o template como você versiona uma API: mudanças breaking têm ciclo de migração explícito, mudanças aditivas são automaticamente disponíveis. E aceite que um template levemente desatualizado ainda é melhor que nenhum template.

O arquiteto que muda organizações

A diferença entre o arquiteto que escreve documentos e o arquiteto que muda organizações cabe numa palavra: mecanismo. Não o mecanismo como burocracia, mais processo, mais comitê, mais

aprovação. O mecanismo como física: aquilo que torna o caminho certo o caminho de menor resistência, que funciona depois que o entusiasmo passou, que sobrevive à troca de liderança, que acumula aprendizado coletivo em vez de deixá-lo evaporar. Em banco, onde o custo de inconsistência é regulatório e o custo de indisponibilidade é reputacional, construir esses mecanismos não é trabalho de suporte à arquitetura, é o trabalho central do arquiteto sênior.

16 - O arquiteto como tradutor de consequência

Fechando o elevador: arquitetura é conversa, decisão e consequência. Em banco, essa consequência aparece em confiança, risco, disponibilidade, auditoria, custo, experiência e velocidade de mudança.

Chegamos ao último andar. Ao longo deste livro percorremos o prédio inteiro, do penthouse onde executivos tomam decisões de risco à sala de máquinas onde idempotência e dupla entrada determinam se o dinheiro fecha, e o fio que costurou cada capítulo foi sempre o mesmo: o arquiteto existe para mover contexto entre esses andares sem perdê-lo no caminho. Este capítulo fecha o elevador e entrega o que o livro prometeu: uma síntese operacional de como arquitetura bancária na AWS conecta estratégia a consequência.

> Minha visão depois de dezesseis anos neste prédio: A maior falha que vejo em arquitetos sênior não é técnica. É a incapacidade de traduzir consequência: de transformar uma decisão de infraestrutura em linguagem de risco para o CFO, ou de transformar uma diretriz regulatória do BACEN em restrição de design para o time de engenharia. Tecnologia excelente escolhida sem essa tradução vira custo sem retorno. Regulação excelente entendida sem essa tradução vira compliance de papel. O arquiteto que sobe e desce o elevador com fluência, sem perder o rigor técnico no penthouse e sem perder a visão de negócio na sala de máquinas, é o ativo mais escasso e mais valioso que um banco pode ter.

O prédio que percorremos, e o que cada andar ensinou

Começamos estabelecendo por que o arquiteto precisa andar de elevador (Capítulo 1) e mapeamos a anatomia dos andares de um banco (Capítulo 2): o penthouse de estratégia e risco, os andares intermediários de produto, operação e conformidade, e a sala de máquinas de runtime e dados. O Capítulo 3 tratou da habilidade mais difícil, subir e descer sem perder contexto, e os Capítulos 4 e 5 nos deram a linguagem correta: bancos são conjuntos de capacidades, não telas, e operam dentro de trilhos regulatórios que não são obstáculos mas restrições de design com consequências jurídicas e reputacionais reais.

O Capítulo 6 foi o coração técnico do livro: o ledger como invariante de negócio, idempotência como propriedade de sobrevivência, e dupla entrada como a prova matemática de que o sistema está consistente. Sem entender esse capítulo, nenhuma decisão de arquitetura em core bancário tem base sólida. O Capítulo 7 materializou tudo isso em uma arquitetura de referência na AWS, não como receita a copiar, mas como mapa de decisões com trade-offs explícitos.

Os Capítulos 8 a 12 construíram as camadas que sustentam o core: eventos como tecido nervoso (substituindo integração ponto a ponto por contratos assíncronos auditáveis), dados como produto com linhagem rastreável (porque em banco dado sem proveniência é dado sem valor regulatório), plataforma e runtime escolhidos pelo modelo operacional e não pelo hype, IA generativa com guardrails que preservam a explicabilidade exigida pelo regulador, e segurança tratada como evidência: não como checklist. O Capítulo 13 foi o mais honesto: a arquitetura só existe em

produção, e projetar para operar às três da manhã é tão importante quanto projetar para escalar. Os Capítulos 14 e 15 fecharam o ciclo de decisão: vender opções e registrar em ADRs, transformar decisões em mecanismos que sobrevivem à rotatividade de pessoas.

Tecnologia importa muito, mas só vira arquitetura quando conectada ao problema certo

Ao longo do livro nomeei tecnologias com precisão intencional: Amazon EventBridge e MSK como espinha dorsal de eventos, Amazon Aurora com Multi-AZ e write-forwarding para consistência forte no core transacional, Amazon S3 e Lake Formation como fundação de dados com controle de acesso por coluna, Amazon Bedrock com guardrails configuráveis para IA generativa, Amazon EKS com Karpenter para workloads que exigem controle de runtime, AWS CloudTrail e Security Hub como camada de evidência auditável. Nomeei essas tecnologias não para fazer propaganda, mas porque a escolha de serviço é uma decisão de trade-off, e trade-offs sem nomes são conversas sem objeto.

Mas a armadilha mais comum que vejo em times de arquitetura bancária é inverter a ordem: escolher a tecnologia primeiro e depois procurar o problema que ela resolve. Isso produz arquiteturas que são tecnicamente impressionantes e operacionalmente frágeis, sistemas que ninguém consegue operar às três da manhã porque foram desenhados para uma apresentação de arquitetura, não para um incidente de produção.

A ordem correta é: entender a capacidade de negócio, identificar o risco dominante nessa capacidade (risco de consistência? de latência? de conformidade? de disponibilidade?), definir os trade-offs aceitáveis com os stakeholders corretos, e só então escolher o serviço que melhor endereça esse risco dentro das restrições operacionais do time. Um banco que escolhe Kafka sem ter um time capaz de operar Kafka em produção não comprou resiliência; comprou complexidade. Um banco que escolhe Lambda para o core transacional sem modelar os limites de idempotência em ambiente serverless não comprou agilidade; comprou inconsistência futura.

Tecnologia sem contexto de problema é ruído. Tecnologia conectada ao risco certo, com trade-offs explícitos e mecanismos de operação definidos, é arquitetura.

O arquiteto como tradutor de consequência, o trabalho real

O título deste capítulo não é metáfora. Em banco, a consequência de uma decisão arquitetural aparece em lugares muito concretos: na disponibilidade do PIX às três da manhã de um sábado, na capacidade de produzir um relatório de linhagem de dados para uma auditoria do BACEN em quarenta e oito horas, no tempo de resposta de um sistema de prevenção a fraude que precisa decidir em menos de duzentos milissegundos, na capacidade de reverter uma migração de dados sem perda de consistência contábil, no custo de conformidade que aparece na linha de resultado do CFO.

Traduzir consequência significa ser capaz de fazer o caminho nos dois sentidos com igual fluência. Subindo: pegar uma decisão técnica, por exemplo, adotar consistência eventual em um serviço de saldo, e traduzi-la em linguagem de risco para o comitê executivo: "isso significa que em janelas de até X milissegundos um cliente pode ver um saldo desatualizado; o risco de reclamação regulatória é Y; a contrapartida é Z de redução de latência e W de redução de custo operacional". Descendo: pegar uma diretriz do penthouse, por exemplo, "precisamos reduzir o tempo de onboarding de cliente para dois dias", e traduzi-la em restrições de design para engenharia: quais APIs do SERPRO precisam ser integradas de forma assíncrona, qual é o modelo de idempotência para reprocessamento de documentos, como o evento de cliente.aprovado propaga para

downstream sem criar acoplamento síncrono.

Essa tradução não é automática. Ela exige que o arquiteto mantenha simultaneamente dois vocabulários, dois modelos mentais e duas escalas de tempo, a escala de trimestre do executivo e a escala de milissegundo do sistema. A maioria dos problemas graves que vi em projetos bancários não foi causada por escolha técnica errada. Foi causada por perda de contexto na transição entre andares: um requisito de negócio que chegou à engenharia sem a restrição regulatória embutida, ou uma limitação técnica que nunca subiu ao penthouse e por isso nunca foi considerada na decisão de prazo.

O que levo daqui, e o que espero que você leve também

Escrever este livro foi um exercício de subir e descer o elevador em texto. Cada capítulo exigiu que eu encontrasse o nível de abstração correto, alto o suficiente para ser útil a um arquiteto sênior que precisa convencer um comitê executivo, técnico o suficiente para ser útil a um engenheiro que vai implementar o mecanismo de idempotência em produção. Não sei se acertei em todos os capítulos, mas sei que tentei honestamente.

O que levo como convicção central, depois de dezesseis anos construindo sistemas financeiros: arquitetura bancária não é sobre tecnologia, é sobre confiança. Confiança do cliente de que seu dinheiro está correto. Confiança do regulador de que o banco pode provar o que fez e quando fez. Confiança do time de engenharia de que o sistema pode ser mudado sem medo. Confiança do executivo de que a arquitetura suporta a estratégia e não a bloqueia.

A AWS oferece os blocos de construção certos para essa confiança, serviços gerenciados que reduzem o risco operacional, primitivas de segurança que produzem evidência auditável, plataformas de dados que permitem rastrear linhagem, modelos de IA que podem ser configurados com guardrails. Mas os blocos não se montam sozinhos. Eles precisam de um arquiteto que entenda o problema antes de escolher a solução, que registre a decisão antes de esquecer o contexto, que construa mecanismos que sobrevivam à sua própria saída do projeto.

Se este livro contribuiu para que você suba e desça o elevador com mais fluência, que você fale com executivos sem perder rigor técnico e com engenharia sem perder a visão de negócio, então cumpriu seu propósito. O trabalho continua. O elevador está esperando.

Os 6 princípios para levar deste livro

- A pergunta não é 'qual tecnologia usar', é 'que risco essa tecnologia reduz e que opção de futuro ela cria ou fecha'. Tecnologia sem essa pergunta é custo sem arquitetura.
- Modele capacidades, domínios e eventos antes de escolher serviço. O modelo de domínio revela os contratos; os contratos revelam os trade-offs; os trade-offs revelam a escolha de serviço correta.
- No core transacional, consistência forte e idempotência não são opções de design, são requisitos de sobrevivência. Consistência eventual no ledger é risco regulatório e reputacional, não trade-off de performance.
- Segurança é evidência, não opinião. Conformidade é restrição de design, não camada adicionada no final. Ambas precisam estar presentes no primeiro ADR, não na última sprint antes do go-live.
- A arquitetura só existe em produção. Desenhe para operar às três da manhã: runbooks, observabilidade com contexto de negócio, alertas com severidade calibrada ao impacto financeiro e regulatório.

• Venda opções, registre em ADRs, transforme em mecanismos. Uma decisão sem registro é uma decisão que será refeita. Um mecanismo sem dono é um mecanismo que será ignorado. Decisões que sobrevivem são as que viraram processo.

Perguntas que ouço frequentemente ao fechar este livro

Este livro é um manual de implementação ou um livro de conceitos?

É deliberadamente os dois, em níveis diferentes. Cada capítulo tem uma camada conceitual (o porquê e o trade-off) e uma camada técnica (o como e o serviço). Um livro só conceitual não ajuda o engenheiro a implementar. Um livro só técnico não ajuda o arquiteto a justificar a decisão para o comitê executivo. O elevador precisa dos dois andares.

Os padrões descritos aqui se aplicam a fintechs pequenas ou apenas a bancos grandes?

Os princípios se aplicam a qualquer instituição que opere sob regulação financeira, o que inclui fintechs com licença de pagamento, IPs e SCDs. A escala de implementação varia: uma fintech pequena pode começar com um subconjunto dos padrões e evoluir. O que não escala para baixo é a exigência de idempotência no core e de evidência em segurança, essas são restrições regulatórias, não escolhas de porte.

Como convencer um CTO ou CFO que ainda vê arquitetura como custo, não como investimento?

Fale a língua deles: risco e opção. Não diga 'precisamos de event sourcing'. Diga 'sem rastreabilidade de eventos, uma auditoria do BACEN pode exigir reconstrução manual de transações, o custo estimado de um incidente desse tipo é X; o custo de implementar o padrão correto agora é Y'. Quando arquitetura vira linguagem de risco quantificado, ela deixa de ser custo e vira seguro. O Capítulo 14 trata disso em detalhe.

Veredicto: o que define o arquiteto que faz diferença em banco

O arquiteto moderno em banco não é o especialista técnico mais profundo da sala, há engenheiros melhores em Kafka, em SQL, em segurança. Também não é o estrategista mais visionário do penthouse, há executivos com mais contexto de negócio e regulação. O valor único do arquiteto está na tradução de consequência entre os andares: falar com executivos sem perder rigor técnico, falar com engenharia sem perder a visão de negócio, e registrar essas traduções em decisões que sobrevivem à rotatividade de pessoas e à pressão de prazo.

Em banco, essa competência determina se a organização decide com clareza ou no escuro. Uma decisão de consistência eventual no ledger tomada sem a tradução correta para o penthouse é um risco regulatório não precificado. Uma diretriz de onboarding em dois dias tomada sem a tradução correta para a sala de máquinas é um prazo impossível disfarçado de meta. O arquiteto que faz essa tradução com honestidade, incluindo os trade-offs desconfortáveis, os riscos que ninguém quer nomear, as limitações técnicas que contradizem o roadmap, é o arquiteto que constrói sistemas que duram.

Este livro foi escrito para esse arquiteto. O elevador está esperando. Suba.

> Referências e leituras complementares: As referências completas deste capítulo e do livro estão consolidadas no bloco [REFERENCES] a seguir, organizado por tema: arquitetura de software e elevador, sistemas bancários e ledger, AWS e serviços de referência, regulação financeira brasileira (BACEN, CMN, LGPD), e leituras recomendadas para cada camada do

prédio.

Para ir além

- 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)