archgents

GUIA PRÁTICO

Data Streaming e agentes de IA em produção

Como levar agentes que agem sobre dados em tempo real da arquitetura à implantação

Archgents

Edição 2026 · América Latina

archgents.com

INTRODUÇÃO

Como levar agentes que agem sobre dados em tempo real da arquitetura à implantação

Este guia reúne o que aprendemos implementando plataformas de dados em movimento e agentes de IA na América Latina. Não é uma introdução à inteligência artificial nem um catálogo de produtos: foi escrito para quem precisa decidir a arquitetura e responder por ela quando o sistema estiver em produção.

Os quatro capítulos seguem a ordem em que os problemas costumam aparecer. Primeiro, por que um agente ligado a dados que chegam tarde toma decisões erradas por mais que se ajuste o modelo. Depois, o que é de fato a plataforma que sustenta esses dados, agora que a IBM a comprou. Em seguida, o que muda quando o agente executa dentro do próprio fluxo de eventos. E por fim, como migrá-la para produção sem parar a operação.

Cada capítulo pode ser lido separadamente. No final há quatro perguntas de controle para revisar um projeto antes de levá-lo para produção.

CONTEÚDO

CAPÍTULO 01

O lote noturno já não é suficiente: por que seus agentes de IA precisam de dados em movimento

Um agente conectado a um data warehouse que se atualiza toda noite raciocina sobre um negócio que já mudou. O problema não está no modelo, está em como os dados chegam.

Quando um piloto de agentes de IA não chega a produção, a conversa quase sempre gira em torno do modelo. Que é preciso ajustar as instruções, que talvez valha a pena trocar de fornecedor, que falta um fine-tuning com dados próprios. Na nossa experiência implementando plataformas de dados na América Latina, a causa costuma estar em outro lugar, bem menos glamouroso: o agente está olhando para dados velhos.

O agente não sabe que está desatualizado

Um modelo de linguagem não tem como saber se o dado que você entregou é de três segundos atrás ou de quatorze horas atrás. Ele vai responder com a mesma confiança nos dois casos. É exatamente essa propriedade que o torna perigoso quando o contexto chega tarde.

Pense em um agente de atendimento ao cliente em um banco. Você pergunta pelo status de uma transferência. O agente consulta o data warehouse, alimentado por um processo noturno, e responde que a transferência está em processamento. Na verdade ela foi rejeitada duas horas atrás e o cliente já recebeu a notificação. O agente não mentiu: respondeu corretamente sobre um estado que deixou de ser verdadeiro.

Nenhuma quantidade de engenharia de prompts corrige isso.

Por que a arquitetura em lote sobreviveu tanto tempo

O modelo de mover dados copiando funcionou por décadas por um motivo simples: as decisões podiam esperar. O relatório de vendas do dia servia para a reunião da manhã seguinte. A conciliação bancária fechava à noite. Ninguém precisava saber o estado do estoque no segundo exato.

Essa premissa se rompeu por duas frentes ao mesmo tempo.

A primeira é o negócio. A fraude se decide dentro da transação, não na revisão do dia seguinte. O cliente que abandona o carrinho faz isso no minuto, não no trimestre. A queda de rede que vai gerar reclamações já está acontecendo enquanto o painel se atualiza.

A segunda é o agente. Um agente autônomo não consulta dados para informar uma pessoa que vai aplicar o próprio critério. Consulta dados para **agir**. E agir sobre um estado que já mudou não é um erro de relatório, é uma decisão errada executada de verdade.

A armadilha de ligar o agente direto ao banco

A reação natural de quem percebe isso é evidente: se o data warehouse está velho, que o agente consulte o banco transacional diretamente.

Isso funciona com um agente e um caso de uso. Para de funcionar rápido.

- Cada agente novo abre mais uma conexão contra o sistema que sustenta a operação. O core bancário, o sistema de estoque ou o de faturamento não foram desenhados para absorver consultas analíticas de agentes que raciocinam em voz alta.
- As consultas de um agente são imprevisíveis em forma e em volume. Um pico de conversas vira um pico de carga sobre o sistema crítico.
- Cada integração é ponto a ponto. Com dez sistemas e dez agentes você tem cem caminhos possíveis, e todos precisam de manutenção.
- Não fica registro do que o agente viu. Quando alguém perguntar por que ele decidiu o que decidiu, não há como reconstruir.

Esse último ponto é o que costuma travar o projeto em bancos e telecomunicações. Não basta o agente acertar: é preciso poder explicar depois.

Colocar os eventos no centro

A alternativa que implementamos é diferente em um ponto que parece pequeno e muda tudo: em vez de cada consumidor ir buscar o dado onde ele mora, cada sistema **publica o que acontece com ele** como um evento, e quem precisar consome dali.

Um pagamento aprovado é um evento. Uma mudança de endereço é um evento. Uma célula de rede degradada é um evento. Um produto que sai do depósito é um evento.

Sobre esse fluxo se resolvem as três coisas que a arquitetura anterior não conseguia dar:

O contexto se constrói enquanto os dados se movem. O perfil de comportamento de um cliente não é recalculado toda noite: ele se mantém atualizado à medida que as transações chegam. Quando o agente pergunta, a resposta já está pronta e já está fresca.

A complexidade deixa de crescer ao quadrado. Um sistema novo publica seus eventos uma vez e qualquer consumidor os toma. Não é preciso construir uma integração para cada par.

Cada decisão fica registrada. Se a decisão do agente também é publicada como evento, o registro completo de quais dados existiam e o que foi decidido sobre eles fica no mesmo lugar. Dá para auditar e dá para reproduzir.

Onde isso começa na prática

Não começa comprando uma licença. Começa escolhendo um caso de uso em que o tempo real muda um resultado mensurável, e trazendo para o fluxo apenas os eventos de que esse caso precisa.

Já vimos projetos que começam querendo publicar o core inteiro como eventos e ficam seis meses empacados na modelagem. E vimos projetos que começam com três tipos de evento, resolvem um caso concreto de fraude e a partir dali crescem por demanda real, não por planejamento especulativo.

O segundo caminho chega a produção. O primeiro costuma terminar como uma apresentação.

O que vale revisar antes de somar mais um agente

Se você está avaliando levar um agente a produção, estas perguntas separam os pilotos que sobrevivem dos que não:

1. De onde sai o dado que o agente vai ver, e com que idade ele chega?
2. Se esse dado muda enquanto o agente raciocina, o que acontece?
3. Você consegue reconstruir, seis meses depois, que informação estava disponível quando ele decidiu?
4. Quando você adicionar o quinto agente, quantas integrações novas serão necessárias?

Se alguma das quatro não tem resposta clara, o problema não está no modelo.

CAPÍTULO 02

O que a IBM comprou quando comprou a Confluent

Quase todo mundo equipara Confluent a Kafka. O Kafka é uma de sete peças, e as outras seis explicam por que a IBM pagou onze bilhões de dólares.

Em 17 de março de 2026 a IBM fechou a compra da Confluent a 31 dólares por ação, um valor de empresa de cerca de onze bilhões de dólares. A Confluent saiu da Nasdaq e passou a ser uma subsidiária da IBM.

A pergunta que nos fazem desde então, em reuniões e no café da manhã executivo que realizamos há pouco, é quase sempre a mesma: se o Kafka é open source e gratuito, o que exatamente a IBM comprou?

A resposta curta é que o Kafka é uma de sete peças. A resposta longa é este artigo.

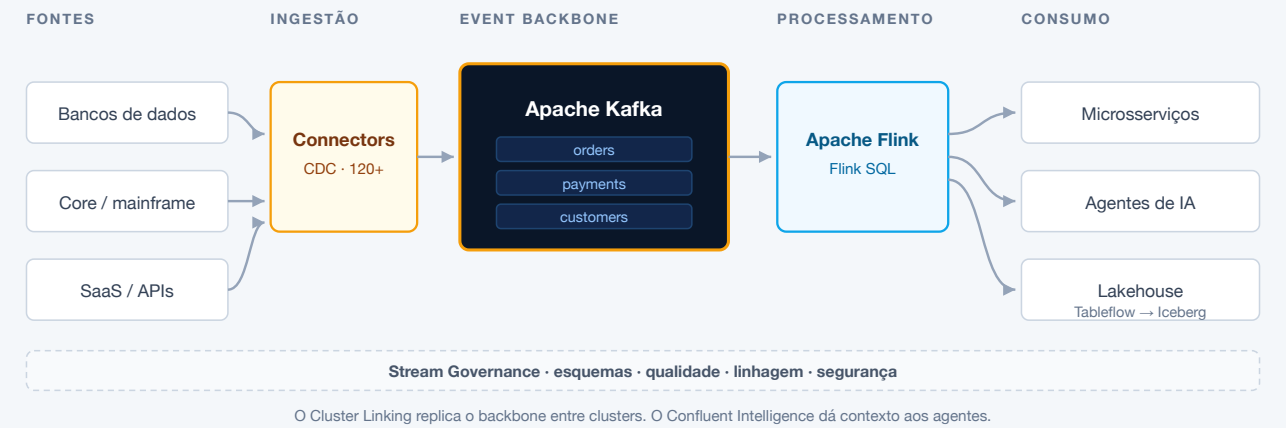
Confluent não é Kafka com suporte

É a confusão mais comum e a mais cara, porque leva a comparar uma licença contra zero e a concluir o óbvio.

O Kafka resolve um problema: guardar eventos em um log distribuído, ordenado e persistente, do qual muitos consumidores podem ler no seu próprio ritmo e reler o passado quando precisarem. É a peça central e é excelente no que faz.

Mas um backbone de eventos em produção precisa de mais seis coisas, e são justamente essas que uma equipe acaba construindo à mão quando não as compra.

As sete peças



As sete peças em um só fluxo: a mesma cor é o mesmo componente em cada desenho.

Apache Kafka é o log de eventos. Ordenado, persistente, particionado para escalar na horizontal, com retenção que permite reler o histórico. É a fonte única da verdade sobre o que está acontecendo agora.

Connectors resolve a conexão com o mundo real. Ligar o core, dezenas de bancos de dados e o SaaS da vez ao backbone costuma significar código sob medida, frágil e caro de manter. Aqui são mais de cento e vinte conectores prontos, com captura de mudanças nativa para ler do core sem tocá-lo. É configuração, não desenvolvimento.

Apache Flink processa em tempo real. Os eventos crus raramente servem sozinhos: é preciso uni-los, enriquecê-los e calcular sobre eles no momento, não em uma consulta batch posterior. O Flink SQL torna essa lógica declarativa e versionável, com resultados em milissegundos.

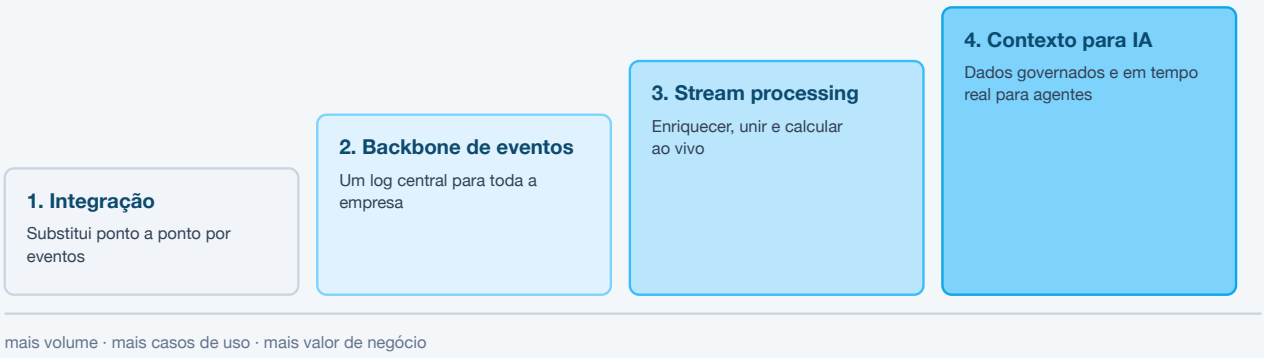
Stream Governance é a peça que a maioria subestima e a que decide se o projeto chega à produção em uma instituição regulada. Sem controle, um backbone de eventos vira um caos de formatos em que ninguém confia. Aqui há um registro de esquemas com contratos versionados, regras de qualidade, linhagem de ponta a ponta, criptografia e controle de acesso.

Tableflow transforma os streams em tabelas Iceberg ou Delta de forma automática. Analítica e IA precisam dos mesmos dados que a operação, e duplicá-los com um ETL noturno é caro, lento e deixa as duas visões fora de sincronia. Um único dado serve às duas.

Cluster Linking replica topics entre clusters, ao vivo. Um único cluster é um ponto único de falha, e cenários híbridos ou multirregião precisam dos mesmos dados disponíveis em vários lugares ao mesmo tempo. Habilita ativo-ativo, recuperação de desastres e a ponte entre on-premise e nuvem sem um ETL no meio.

Confluent Intelligence é a peça mais nova e a que explica o interesse da IBM. Um agente alucina quando lhe falta contexto fresco e confiável do negócio, e não consegue agir sobre o que está acontecendo agora. Essa camada dá contexto ao vivo para o agente e permite que ele execute dentro do próprio fluxo de eventos, sobre dados governados e rastreáveis.

Como uma plataforma assim amadurece



O mesmo padrão amadurece em camadas, e cada nível habilita mais casos de uso sobre a mesma infraestrutura.

Ninguém começa pela última camada, e tentar isso é o erro mais comum.

1. **Integração.** As conexões ponto a ponto são substituídas por eventos. O objetivo é parar de ter N sistemas falando com N sistemas.
2. **Backbone de eventos.** O log deixa de ser de um projeto e passa a ser da empresa.
3. **Stream processing.** Enriquecer, unir e calcular ao vivo sobre esse backbone.
4. **Contexto para IA.** Os agentes consomem dados governados e em tempo real, e agem sobre eles.

Cada camada habilita mais casos de uso e mais volume sobre a mesma infraestrutura. Microsserviços desacoplados, fraude ao vivo, cliente 360, modernização do core com captura de mudanças, risco e exposição calculados na hora, personalização no momento em que o cliente age. Todos rodam em paralelo sobre o mesmo backbone, sem duplicar a plataforma.

O que a plataforma ganha com a IBM

O comunicado da IBM é claro na sua intenção: **a Confluent transmite eventos operacionais ao vivo diretamente para o watsonx.data**, para que cada modelo, agente e fluxo de trabalho rode sobre dados corporativos continuamente atualizados. Essa é a tese da compra, e é coerente com a quarta camada acima.

Para quem já opera Confluent, ou está avaliando entrar, é isto que muda na prática.

A plataforma passa a ter a IBM por trás. Onze bilhões de dólares é uma aposta grande e é um sinal. A Confluent deixa de responder a um mercado trimestral e passa a fazer parte do portfólio de dados de uma empresa com décadas de presença corporativa, contratos globais e suporte na região.

O caminho até os agentes fica mais curto. A integração com o watsonx.data é justamente a peça que antes era preciso construir à mão: levar o contexto vivo do negócio até o modelo. Agora vem de fábrica e vem governada.

A base continua open source. Kafka e Flink são projetos da Apache Software Foundation. Isso não mudou com a aquisição e não vai mudar. A arquitetura de eventos que você desenhar é sua e continua portável, e nenhuma decisão corporativa prende você.

Os contratos vigentes seguem o seu curso. Os acordos plurianuais correm até o seu término. A renovação é o momento natural para revisar o roadmap conjunto, e é aí que vale sentar com quem conhece as duas casas.

A liderança de produto subiu de nível. Em agosto de 2026 Jay Kreps, cofundador e criador do Kafka, entregou a direção a Shaun Clowes, até então diretor de produto, hoje gerente geral de Confluent e Dados na IBM. É um bom sinal: o data streaming não virou mais uma linha do catálogo, ganhou cadeira própria na estratégia de dados da IBM.

Por onde começar

A aquisição não muda a análise técnica. As sete peças são as mesmas, o padrão de arquitetura é o mesmo e os casos de uso que ele habilita também. O que muda é que agora há um respaldo maior por trás e um caminho mais curto até os agentes.

É assim que fazemos, e funciona:

1. **Assessment.** Escolhemos um caso de alto impacto, um só. Não um plano de três anos.
2. **Desenho de arquitetura.** Modelamos a solução sobre as peças que o seu caso realmente precisa, nem uma a mais.
3. **Prova de conceito governada.** Na Confluent Cloud, em semanas, com esquemas e linhagem desde o primeiro dia.

Passamos sete anos implementando Confluent na América Latina e hoje somos parceiros IBM. É a mesma arquitetura que vínhamos construindo antes da compra, agora com um parceiro maior por trás.

Fontes: o comunicado de fechamento da operação está no newsroom da IBM, e o anúncio original da aquisição no blog da Confluent.

CAPÍTULO 03

Streaming Agents: agentes que executam dentro do fluxo de dados

Quando o agente roda como um job do Flink em vez de um serviço à parte, ele deixa de pedir contexto e passa a tê-lo. O que muda e quando vale a pena.

A forma habitual de construir um agente é como um serviço: ele recebe uma requisição, consulta o que precisa, raciocina e responde. É o modelo que herdamos das aplicações web e funciona bem quando alguém faz uma pergunta.

Existe uma segunda forma, menos conhecida, que muda onde o agente vive: em vez de ser um serviço a quem se pergunta, o agente é um **processo que executa dentro do fluxo de eventos**. Ninguém o invoca. Ele é ativado quando ocorre o evento que lhe cabe atender.

A diferença não é de infraestrutura

À primeira vista parece uma decisão de deploy. Na prática muda três coisas de fundo.

O agente deixa de pedir contexto. Um agente-serviço, quando precisa saber o histórico de um cliente, faz uma consulta e espera. Um agente que roda dentro do fluxo já tem esse estado à mão, porque o motor de processamento vem mantendo esse estado a cada evento que passou. A diferença entre consultar e ter costuma ser de centenas de milissegundos por decisão, o que em volume de produção não é um detalhe.

O agente reage em vez de esperar. Um serviço não faz nada até alguém chamá-lo. Isso significa que o processo só começa quando um humano ou um sistema decide iniciá-lo. Quando o agente vive no fluxo, o gatilho é o próprio fato. A degradação de rede não espera o cliente ligar para que alguém olhe para ela.

O agente escala como o fluxo escala. O motor já sabe repartir o trabalho por partição e garantir que os eventos de uma mesma entidade sejam processados pela mesma tarefa. Essa é exatamente a repartição de que um agente com estado precisa, e ela vem resolvida.

O que se ganha em rastreabilidade

Há um benefício menos óbvio que em setores regulados costuma ser o decisivo.

Quando o agente executa dentro do fluxo, tanto os dados que ele viu quanto a decisão que tomou são eventos no mesmo registro. Isso permite algo que com um agente-serviço é muito difícil: **reproduzir o caso**. Dá para voltar à posição exata do fluxo, rodar a versão do agente que estava em vigor naquele dia e ver por que ele decidiu o que decidiu.

Com um agente-serviço, essa reconstrução exige ter registrado separadamente a requisição, a resposta de cada sistema consultado e a versão do modelo, e confiar que os três registros batam. Quase nunca batem.

Quando não vale a pena

Este padrão não substitui o agente conversacional, e forçá-lo onde não cabe é um erro caro.

Não vale a pena quando **a interação é iniciada por uma pessoa**. Se o caso de uso é alguém escrevendo em um chat, o agente-serviço é a forma correta.

Não vale a pena quando **o raciocínio é longo e de muitos passos**. Um fluxo de eventos é otimizado para processar muito com baixa latência. Um agente que vai fazer quinze chamadas a ferramentas e demorar quarenta segundos não encaixa nesse modelo, e colocá-lo ali gera backpressure rio acima.

Não vale a pena quando **o volume é baixo**. Se são cem eventos por dia, a complexidade operacional não se justifica.

O critério prático: se o gatilho é um fato do negócio, o volume é alto e a decisão precisa sair rápido, o agente vai no fluxo. Se o gatilho é uma pessoa e o raciocínio é aberto, ele vai como serviço.

Os dois convivendo

Na maioria das implementações reais os dois acabam operando juntos, e essa é a arquitetura saudável.

O agente no fluxo faz o trabalho contínuo: avalia cada transação, mantém o contexto atualizado, detecta o que foge do normal e dispara a ação quando é o caso. O agente conversacional atende a pessoa, e quando precisa saber o estado do negócio consulta o contexto que o primeiro vem mantendo.

Dito de outro modo: um constrói a verdade sobre o presente, o outro a explica. É uma divisão de trabalho bem mais limpa do que pedir a um único agente que faça as duas coisas.

Por onde começar

Se você já tem uma plataforma de eventos operando, o primeiro caso costuma ser o mais chato de propósito: pegar uma regra que hoje roda em lote e movê-la para o fluxo, sem agentes no meio. Isso valida o modelo de estado, o dimensionamento e a operação.

Com isso funcionando, acrescentar raciocínio sobre esse mesmo fluxo é um passo incremental e não um salto. O erro mais comum que vemos é o inverso: começar pelo agente e descobrir depois que a plataforma de dados de que ele precisava não estava pronta.

CAPÍTULO 04

Como migrar do Apache Kafka open source para a Confluent sem parar a operação

Uma migração de plataforma de streaming não se faz com uma janela de manutenção. Faz-se em convivência, por domínios, e com a possibilidade de voltar atrás em cada etapa.

Muitas organizações da região chegaram ao Apache Kafka pelo caminho certo: uma equipe com bom critério subiu o cluster para resolver um problema concreto, funcionou, e ele foi virando infraestrutura crítica sem que ninguém tomasse a decisão formal de que seria.

O ponto de ruptura chega quando esse cluster que começou como projeto de uma equipe sustenta processos que não podem cair, e a organização percebe que a operação depende de duas ou três pessoas que sabem como aquilo funciona.

É aí que aparece a conversa sobre migrar para uma plataforma gerenciada. E é aí que a maioria dos planos se quebra, porque são desenhados como um corte.

Por que a janela de manutenção não funciona

O instinto natural é agendar um fim de semana, mover tudo e voltar na segunda-feira. Com um banco de dados às vezes dá. Com uma plataforma de eventos, quase nunca.

A razão é que o Kafka não é um repositório, é um fluxo com estado distribuído em muitos consumidores. Migrar significa mover não só os dados, mas a **posição de leitura** de cada aplicação consumidora, e fazer isso de forma que nenhuma reprocesse o que já processou nem pule o que faltava.

Com cinco aplicações pode dar certo. Com quarenta, a probabilidade de que todas as quarenta fiquem exatamente onde deveriam estar na mesma janela é baixa. E o modo de falha não é um erro visível: é um consumidor que ficou atrasado e ninguém percebe até aparecer uma diferença em uma conciliação três dias depois.

Convivência em vez de corte

A abordagem que aplicamos é outra: os dois clusters operam em paralelo durante a migração, e os domínios se movem de um para o outro quando cada um está pronto.

A peça que torna isso possível é a replicação entre clusters. Os topics são replicados da origem para o destino de forma contínua, incluindo a posição dos consumidores. Isso transforma um evento único e irreversível em uma série de decisões pequenas e reversíveis.

A ordem que funcionou para nós:

Primeiro se replica, sem mover ninguém. O cluster destino recebe tudo e ninguém consome. Aqui se validam o desempenho, a retenção, os esquemas e o dimensionamento real, com tráfego de produção e sem risco.

Depois se movem os consumidores, não os produtores. Um consumidor que lê do destino em vez da origem é uma mudança de configuração reversível. Se algo der errado, ele volta a apontar para a origem e segue operando. Começa-se pelos consumidores menos críticos.

No final se movem os produtores, domínio por domínio. Este é o passo que torna o corte irreversível para aquele domínio, e por isso é o último. Quando um produtor escreve no destino, seus consumidores já vêm lendo dali há tempo sem incidentes.

A origem se desliga quando ninguém mais a usa, não quando o plano dizia que era a hora.

O que se descobre pelo caminho

Uma migração de streaming quase sempre revela coisas que estavam escondidas. Vale esperá-las em vez de ser surpreendido por elas.

Topics que ninguém consome. É comum descobrir que parte do que está sendo publicado não é lido por ninguém há meses. Migrar é uma boa oportunidade para parar de pagar por movê-los.

Esquemas que na verdade não existem. Muitos clusters open source operam sem Schema Registry, com o contrato vivendo na cabeça da equipe que o escreveu. Ao passar para uma plataforma com Schema Registry aparecem as incompatibilidades que estavam latentes. É melhor que apareçam na migração do que em um incidente.

Configurações de retenção herdadas. Retenções de sete dias colocadas por padrão três anos atrás, sobre topics em que o negócio precisaria de trinta, ou o contrário.

Consumidores que dependem da ordem sem saber. Aplicações que funcionam porque as partições chegaram até elas em certa ordem e ninguém documentou isso.

O que é preciso ter antes de começar

Antes de replicar o primeiro topic, convém ter três coisas resolvidas.

Um **inventário real de produtores e consumidores**, não o diagrama de arquitetura, mas quem está conectado hoje. Quase sempre divergem.

Um **critério de domínio** para decidir o que se move junto. Mover por ordem alfabética de topic garante problemas; mover por domínio de negócio, não.

Uma **definição explícita do que significa um domínio estar migrado**, acordada antes e não negociada durante. Sem isso, cada corte vira uma discussão.

Quanto tempo leva

Depende quase inteiramente do número de aplicações produtoras e de quão acoplado está o consumo, não do volume de dados. Um cluster com muito tráfego e poucas aplicações se migra rápido. Um cluster médio com dezenas de aplicações de equipes diferentes leva bem mais tempo, porque o trabalho real é de coordenação, não técnico.

O que dá para afirmar com segurança é o contrário do instinto inicial: não é preciso parar a operação. Se o plano de migração exige uma janela de indisponibilidade para uma plataforma de eventos, provavelmente o plano está errado.

LISTA DE VERIFICAÇÃO

Antes de levar um agente para produção

Quatro perguntas que separam os pilotos que chegam à produção dos que ficam em demonstração. Se alguma não tiver resposta clara, o problema provavelmente não está no modelo.

1 **De onde vem o dado que o agente vai ver, e com que idade ele chega?**

Se a resposta for "do armazém, carregado à noite", o agente está raciocinando sobre um negócio que já mudou.

2 **O que acontece se esse dado mudar enquanto o agente raciocina?**

Um agente que consulta uma vez e decide depois nunca vê a correção. Um que executa dentro do fluxo vê.

3 **Você consegue reconstruir, seis meses depois, que informação o agente tinha quando decidiu?**

Sem um log de eventos imutável não há auditoria possível, e isso bloqueia a produção em qualquer instituição regulada.

4 **Quando você adicionar o quinto agente, quantas integrações novas serão necessárias?**

Se a resposta cresce a cada agente, o problema é a arquitetura de integração, não o agente.

Vamos falar do seu caso

Passamos sete anos implementando Confluent na América Latina. Hoje somos parceiros IBM e trabalhamos com Google ADK e IBM watsonx para levar agentes à produção sobre dados em movimento.

Agende 30 minutos com um arquiteto sênior. Analisamos o seu caso concreto e dizemos o que seria necessário, e o que não seria. Sem apresentação de vendas.

Agendar diagnóstico de 30 minutos

Ou fale conosco pelo WhatsApp: **+57 300 561 3240**

MAIS SOBRE ISSO

Data Streaming com Confluent	archgents.com/pt/data-streaming/
Agentes com Google ADK	archgents.com/pt/agentes-google-adk/
Agentes com IBM watsonx	archgents.com/pt/agentes-ibm-watsonx/
Casos de uso por indústria	archgents.com/pt/casos-de-uso/