archgents

GUÍA PRÁCTICA

# Data Streaming y agentes IA en producción

Cómo llevar agentes que actúan sobre datos en tiempo real, de la arquitectura al despliegue




**Archgents**

Edición 2026 · América Latina

archgents.com

INTRODUCCIÓN

# Cómo llevar agentes que actúan sobre datos en tiempo real, de la arquitectura al despliegue

Esta guía reúne lo que hemos aprendido implementando plataformas de datos en movimiento y agentes IA en América Latina. No es una introducción a la inteligencia artificial ni un catálogo de producto: está escrita para quien tiene que decidir la arquitectura y responder por ella cuando el sistema esté en producción.

Los cuatro capítulos van en el orden en que suelen aparecer los problemas. Primero, por qué un agente conectado a datos que llegan tarde toma decisiones equivocadas por más que se ajuste el modelo. Después, qué es realmente la plataforma que sostiene esos datos, ahora que IBM la compró. Luego, qué cambia cuando el agente ejecuta dentro del propio flujo de eventos. Y por último, cómo se migra a producción sin detener la operación.

Cada capítulo se puede leer por separado. Al final hay cuatro preguntas de control para revisar un proyecto antes de llevarlo a producción.

CONTENIDO

CAPÍTULO 01

# El lote nocturno ya no alcanza: por qué tus agentes IA necesitan datos en movimiento

Un agente conectado a un almacén que se actualiza cada noche razona sobre un negocio que ya cambió. El problema no está en el modelo, está en cómo llegan los datos.

---

Cuando un piloto de agentes IA no llega a producción, la conversación casi siempre gira alrededor del modelo. Que si hay que ajustar las instrucciones, que si conviene cambiar de proveedor, que si hace falta afinar el modelo con datos propios. En nuestra experiencia implementando plataformas de datos en América Latina, la causa suele estar en otra parte y es mucho menos glamorosa: el agente está mirando datos viejos.

## El agente no sabe que está desactualizado

Un modelo de lenguaje no tiene forma de saber si el dato que le entregaste corresponde a hace tres segundos o a hace catorce horas. Responderá con la misma seguridad en ambos casos. Esa es exactamente la propiedad que lo hace peligroso cuando el contexto llega tarde.

Piensa en un agente de atención al cliente en un banco. Le preguntas por el estado de una transferencia. El agente consulta el almacén de datos, que se alimenta de un proceso nocturno, y responde que la transferencia está en proceso. En realidad se rechazó hace dos horas y el cliente ya recibió la notificación. El agente no mintió: respondió correctamente sobre un estado que dejó de ser cierto.

Ninguna cantidad de ingeniería de instrucciones corrige eso.

## Por qué la arquitectura por lotes sobrevivió tanto tiempo

El modelo de mover datos copiándolos funcionó durante décadas por una razón sencilla: las decisiones podían esperar. El reporte de ventas del día servía para la reunión de la mañana siguiente. La conciliación bancaria cerraba en la noche. Nadie necesitaba saber el estado del inventario en el segundo exacto.

Ese supuesto se rompió por dos frentes al mismo tiempo.

El primero es el negocio. El fraude se decide dentro de la transacción, no en la revisión del día siguiente. El cliente que abandona el carrito lo hace en el minuto, no en el trimestre. La caída de red que va a generar reclamos ya está ocurriendo mientras el tablero se actualiza.

El segundo es el agente. Un agente autónomo no consulta datos para informar a una persona que va a aplicar su criterio. Consulta datos para **actuar**. Y actuar sobre un estado que ya cambió no es un error de reporte, es una decisión equivocada ejecutada de verdad.

## La trampa de conectar el agente directo a la base

La reacción natural cuando alguien se da cuenta de esto es evidente: si el almacén está viejo, que el agente consulte la base transaccional directamente.

Esto funciona con un agente y un caso de uso. Deja de funcionar rápido.

- Cada agente nuevo abre otra conexión contra el sistema que sostiene la operación. El core bancario, el sistema de inventario o el de facturación no fueron diseñados para absorber consultas analíticas de agentes que razonan en voz alta.
- Las consultas de un agente son impredecibles en forma y en volumen. Un pico de conversaciones se convierte en un pico de carga sobre el sistema crítico.
- Cada integración es punto a punto. Con diez sistemas y diez agentes tienes cien caminos posibles, y todos hay que mantenerlos.
- No queda registro de qué vio el agente. Cuando alguien pregunte por qué decidió lo que decidió, no hay forma de reconstruirlo.

Ese último punto es el que suele frenar el proyecto en banca y telecomunicaciones. No basta con que el agente acierte: hay que poder explicarlo después.

## Poner los eventos en el centro

La alternativa que implementamos es distinta en un punto que parece pequeño y lo cambia todo: en lugar de que cada consumidor vaya a buscar el dato donde vive, cada sistema **publica lo que le ocurre** como un evento, y quien lo necesite lo consume desde ahí.

Un pago aprobado es un evento. Un cambio de dirección es un evento. Una celda de red degradada es un evento. Un producto que sale de bodega es un evento.

Sobre ese flujo se resuelven las tres cosas que la arquitectura anterior no podía dar:

**El contexto se construye mientras los datos se mueven.** El perfil de comportamiento de un cliente no se recalcula cada noche: se mantiene actualizado a medida que llegan sus transacciones. Cuando el agente pregunta, la respuesta ya está lista y ya está fresca.

**La complejidad deja de crecer al cuadrado.** Un sistema nuevo publica sus eventos una vez y cualquier consumidor los toma. No hay que construir una integración por cada par.

**Cada decisión queda registrada.** Si la decisión del agente también se publica como evento, el registro completo de qué datos existían y qué se decidió sobre ellos queda en el mismo lugar. Se puede auditar y se puede reproducir.

## Dónde empieza esto en la práctica

No empieza comprando una licencia. Empieza eligiendo un caso de uso donde el tiempo real cambie un resultado medible, y trayendo al flujo solo los eventos que ese caso necesita.

Hemos visto proyectos que arrancan queriendo publicar todo el core como eventos y se atascan seis meses en modelado. Y hemos visto proyectos que arrancan con tres tipos de evento, resuelven un caso de fraude concreto y a partir de ahí crecen por demanda real, no por planificación especulativa.

El segundo camino llega a producción. El primero suele terminar como una presentación.

## Lo que conviene revisar antes de sumar otro agente

Si estás evaluando llevar un agente a producción, estas preguntas separan los pilotos que sobreviven de los que no:

1. ¿De dónde sale el dato que va a ver el agente, y con qué antigüedad llega?
2. Si ese dato cambia mientras el agente razona, ¿qué pasa?
3. ¿Puedes reconstruir, seis meses después, qué información tenía disponible cuando decidió?
4. Cuando agregues el quinto agente, ¿cuántas integraciones nuevas van a hacer falta?

Si alguna de las cuatro no tiene respuesta clara, el problema no está en el modelo.

CAPÍTULO 02

# Qué compró IBM cuando compró Confluent

Casi todo el mundo equipara Confluent con Kafka. Kafka es una de siete piezas, y entender las otras seis explica por qué IBM pagó once mil millones de dólares.

---

El 17 de marzo de 2026 IBM cerró la compra de Confluent por 31 dólares la acción, unos once mil millones de dólares. Confluent dejó de cotizar en Nasdaq y pasó a ser una subsidiaria de IBM.

La pregunta que nos hacen desde entonces, en reuniones y en el desayuno ejecutivo que dimos hace poco, es casi siempre la misma: si Kafka es open source y gratis, ¿qué compró IBM exactamente?

La respuesta corta es que Kafka es una de siete piezas. La respuesta larga es este artículo.

## Confluent no es Kafka con soporte

Es la confusión más común y la más cara, porque lleva a comparar una licencia contra cero y a concluir lo obvio.

Kafka resuelve un problema: guardar eventos en un log distribuido, ordenado y persistente, del que muchos consumidores pueden leer a su ritmo y releer el pasado cuando lo necesiten. Es la pieza central y es excelente en lo suyo.

Pero un backbone de eventos en producción necesita seis cosas más, y esas son las que un equipo termina construyendo a mano cuando no las compra.

## Las siete piezas


Las siete piezas en un solo flujo: el mismo color es el mismo componente en cada diseño.

**Apache Kafka** es el log de eventos. Ordenado, persistente, particionado para escalar en horizontal, con retención que permite releer el historial. Es la fuente única de la verdad de lo que está pasando ahora.

**Connectors** resuelve la conexión con el mundo real. Conectar el core, decenas de bases de datos y el SaaS de turno al backbone suele significar código a medida, frágil y caro de mantener. Aquí son más de ciento veinte conectores listos, con captura de cambios nativa para leer del core sin tocarlo. Es configuración, no desarrollo.

**Apache Flink** procesa en vivo. Los eventos crudos rara vez sirven solos: hay que unirlos, enriquecerlos y calcular sobre ellos en el momento, no en una consulta batch posterior. Flink SQL hace esa lógica declarativa y versionable, con resultados en milisegundos.

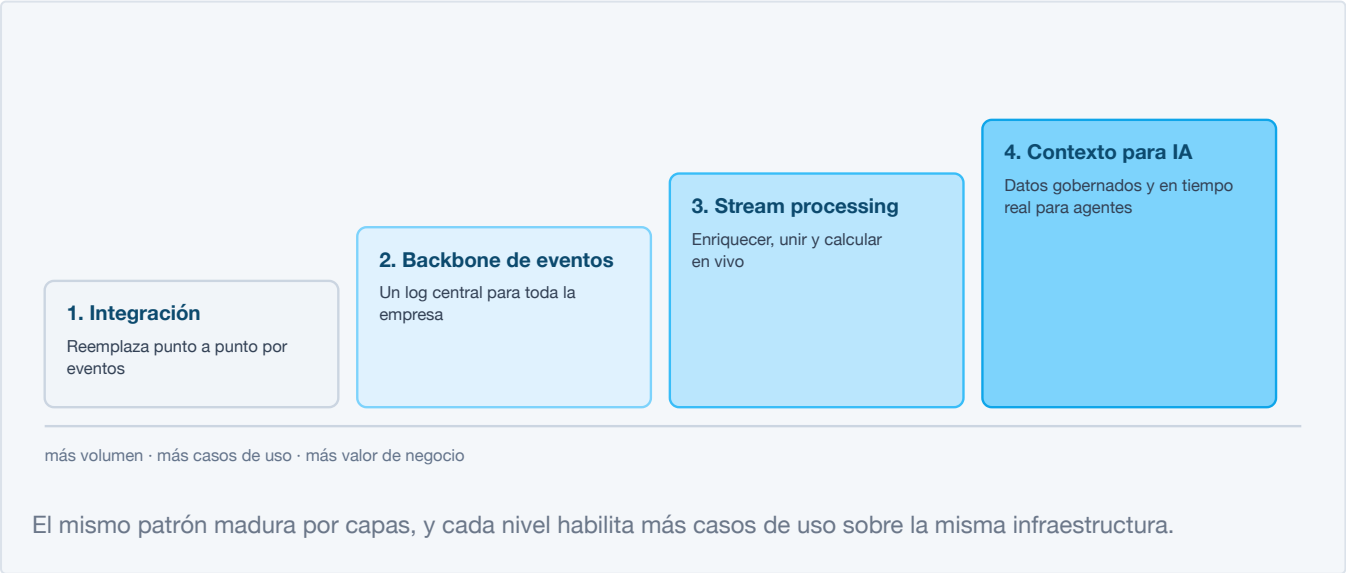
**Stream Governance** es la pieza que la mayoría subestima y la que decide si el proyecto pasa a producción en una entidad regulada. Sin control, un backbone de eventos se convierte en un caos de formatos donde nadie confía en el dato. Aquí hay un registro de esquemas con contratos versionados, reglas de calidad, linaje de punta a punta, cifrado y control de acceso.

**Tableflow** convierte los streams en tablas Iceberg o Delta de forma automática. Analítica e IA necesitan los mismos datos que la operación, y duplicarlos con un ETL nocturno es caro, lento y desincroniza las dos vistas. Un solo dato sirve a las dos.

**Cluster Linking** replica topics entre clusters en vivo. Un solo cluster es un punto único de falla, y los escenarios híbridos o multi-región necesitan los mismos datos disponibles en varios sitios a la vez. Habilita activo-activo, recuperación ante desastres y el puente entre on-premise y nube sin un ETL de por medio.

**Confluent Intelligence** es la pieza más nueva y la que explica el interés de IBM. Un agente alucina cuando le falta contexto fresco y confiable del negocio, y no puede actuar sobre lo que está pasando ahora. Esta capa le da contexto en vivo al agente y permite que ejecute dentro del propio flujo de eventos, sobre datos gobernados y trazables.

## Cómo madura una plataforma así




El mismo patrón madura por capas, y cada nivel habilita más casos de uso sobre la misma infraestructura.

Nadie llega al último nivel de entrada, y el error más común es intentarlo.

1. **Integración.** Se reemplazan las conexiones punto a punto por eventos. El objetivo es dejar de tener N sistemas hablando con N sistemas.
2. **Backbone de eventos.** El log deja de ser de un proyecto y pasa a ser de la empresa.
3. **Stream processing.** Se enriquece, se une y se calcula en vivo sobre ese backbone.
4. **Contexto para IA.** Los agentes consumen datos gobernados y en tiempo real, y actúan sobre ellos.

Cada nivel habilita más casos de uso y más volumen sobre la misma infraestructura. Microservicios desacoplados, fraude en vivo, cliente 360, modernización del core con captura de cambios, riesgo y exposición al instante, personalización en el momento en que el cliente actúa. Todos corren en paralelo sobre el mismo backbone, sin duplicar plataforma.

## Qué gana la plataforma con IBM

El comunicado de IBM es claro en su intención: **Confluent transmite eventos operativos en vivo directamente a watsonx.data**, para que cada modelo, agente y flujo de trabajo corra sobre datos empresariales continuamente actualizados. Esa es la tesis de la compra, y es coherente con la cuarta capa de arriba.

Para quien ya opera Confluent, o está evaluando entrar, esto es lo que cambia en la práctica.

**La plataforma queda respaldada por IBM.** Once mil millones de dólares es una apuesta grande y es una señal. Confluent deja de responder a un mercado trimestral y pasa a ser parte del portafolio de datos de una compañía con décadas de presencia empresarial, contratos globales y soporte en la región.

**El camino hacia los agentes se acorta.** La integración con watsonx.data es justo la pieza que antes había que construir a mano: llevar el contexto vivo del negocio hasta el modelo. Ahora viene de fábrica y con gobierno.

**La base sigue siendo open source.** Kafka y Flink son proyectos de la Apache Software Foundation. Eso no cambió con la adquisición ni va a cambiar. La arquitectura de eventos que diseñes es tuya y es portable, y ninguna decisión corporativa te encierra.

**Los contratos vigentes siguen su curso.** Los acuerdos plurianuales corren hasta su término. La renovación es el momento natural para revisar el roadmap conjunto, y ahí conviene sentarse con alguien que conozca las dos casas.

**El liderazgo del producto subió de nivel.** En agosto de 2026 Jay Kreps, cofundador y creador de Kafka, entregó la dirección a Shaun Clowes, hasta entonces director de producto, que hoy es gerente general de Confluent y Datos en IBM. Es una buena señal: el data streaming no quedó como una línea más del catálogo, quedó con silla propia en la estrategia de datos de IBM.

## Por dónde empezar

La adquisición no cambia el análisis técnico. Las siete piezas son las mismas, el patrón de arquitectura es el mismo y los casos de uso que habilita también. Lo que cambia es que ahora hay un respaldo más grande detrás y un camino más corto hacia los agentes.

Así lo hacemos nosotros, y funciona:

1. **Assessment.** Elegimos un caso de alto impacto, uno solo. No un plan a tres años.
2. **Diseño de arquitectura.** Modelamos la solución sobre las piezas que tu caso realmente necesita, ni una más.
3. **Prueba de concepto gobernada.** En Confluent Cloud, en semanas, con esquemas y linaje desde el primer día.

Llevamos siete años implementando Confluent en América Latina y hoy somos partners de IBM. Es la misma arquitectura que venimos haciendo desde antes de la compra, ahora con un socio más grande detrás.

---

**Fuentes:** el comunicado de cierre de la operación está en el newsroom de IBM, y el anuncio original de la adquisición en el blog de Confluent.

CAPÍTULO 03

# Streaming Agents: agentes que ejecutan dentro del flujo de datos

Cuando el agente corre como un job de Flink en lugar de un servicio aparte, deja de pedir contexto y empieza a tenerlo. Qué cambia y cuándo conviene.

---

La forma habitual de construir un agente es como un servicio: recibe una petición, consulta lo que necesita, razona y responde. Es el modelo que heredamos de las aplicaciones web y funciona bien cuando alguien pregunta algo.

Hay una segunda forma, menos conocida, que cambia dónde vive el agente: en lugar de ser un servicio al que se le pregunta, el agente es un **proceso que ejecuta dentro del flujo de eventos**. Nadie lo invoca. Se activa cuando ocurre el evento que le corresponde atender.

## La diferencia no es de infraestructura

A primera vista parece una decisión de despliegue. En la práctica cambia tres cosas de fondo.

**El agente deja de pedir contexto.** Un agente-servicio, cuando necesita saber el historial de un cliente, hace una consulta y espera. Un agente que corre dentro del flujo ya tiene ese estado a mano, porque el motor de procesamiento lo viene manteniendo con cada evento que pasó. La diferencia entre consultar y tener suele ser de cientos de milisegundos por decisión, que a volumen de producción no es un detalle.

**El agente reacciona en lugar de esperar.** Un servicio no hace nada hasta que alguien lo llama. Eso significa que el proceso solo empieza cuando un humano o un sistema decide iniciarlo. Cuando el agente vive en el flujo, el disparador es el hecho mismo. La degradación de red no espera a que el cliente llame para que alguien la mire.

**El agente escala como escala el flujo.** El motor ya sabe repartir el trabajo por partición y garantizar que los eventos de una misma entidad los procese la misma tarea. Ese es exactamente el reparto que un agente con estado necesita, y viene resuelto.

## Qué se gana en trazabilidad

Hay un beneficio menos obvio que en sectores regulados suele ser el decisivo.

Cuando el agente ejecuta dentro del flujo, tanto los datos que vio como la decisión que tomó son eventos en el mismo registro. Eso permite algo que con un agente-servicio es muy difícil: **reproducir el caso**. Se puede volver a la posición exacta del flujo, correr la versión del agente que estaba vigente ese día y ver por qué decidió lo que decidió.

Con un agente-servicio esa reconstrucción exige haber registrado por separado la petición, la respuesta de cada sistema consultado y la versión del modelo, y confiar en que los tres registros cuadren. Casi nunca cuadran.

## Cuándo no conviene

Este patrón no reemplaza al agente conversacional, y forzarlo donde no corresponde es un error caro.

No conviene cuando **la interacción la inicia una persona**. Si el caso de uso es alguien escribiendo en un chat, el agente-servicio es la forma correcta.

No conviene cuando **el razonamiento es largo y de muchos pasos**. Un flujo de eventos está optimizado para procesar mucho con baja latencia. Un agente que va a hacer quince llamadas a herramientas y tardar cuarenta segundos no encaja en ese modelo, y meterlo ahí genera contrapresión aguas arriba.

No conviene cuando **el volumen es bajo**. Si son cien eventos al día, la complejidad operativa no se justifica.

El criterio práctico: si el disparador es un hecho del negocio, el volumen es alto y la decisión tiene que salir rápido, el agente va en el flujo. Si el disparador es una persona y el razonamiento es abierto, va como servicio.

## Los dos conviviendo

En la mayoría de las implementaciones reales terminan operando los dos, y esa es la arquitectura sana.

El agente en el flujo hace el trabajo continuo: evalúa cada transacción, mantiene el contexto actualizado, detecta lo que se sale de lo normal y dispara la acción cuando corresponde. El agente conversacional atiende a la persona, y cuando necesita saber el estado del negocio consulta el contexto que el primero viene manteniendo.

Puesto de otro modo: uno construye la verdad sobre el presente, el otro la explica. Es una división de trabajo bastante más limpia que pedirle a un solo agente que haga ambas.

## Por dónde empezar

Si ya tienes una plataforma de eventos operando, el primer caso suele ser el más aburrido a propósito: tomar una regla que hoy corre por lotes y moverla al flujo, sin agentes de por medio. Eso valida el modelo de estado, el dimensionamiento y la operación.

Con eso funcionando, agregar razonamiento sobre ese mismo flujo es un paso incremental y no un salto. El error más común que vemos es el inverso: empezar por el agente y descubrir después que la plataforma de datos que necesitaba no estaba lista.

CAPÍTULO 04

# Cómo migrar de Apache Kafka open source a Confluent sin detener la operación

Una migración de plataforma de streaming no se hace con una ventana de mantenimiento. Se hace conviviendo, por dominios, y con la posibilidad de devolverse en cada etapa.

---

Muchas organizaciones de la región llegaron a Apache Kafka por el camino correcto: un equipo con criterio lo levantó para resolver un problema concreto, funcionó, y se fue convirtiendo en infraestructura crítica sin que nadie tomara la decisión formal de que lo fuera.

El punto de quiebre llega cuando ese clúster que empezó como un proyecto de un equipo sostiene procesos que no pueden caerse, y la organización se da cuenta de que la operación depende de dos o tres personas que saben cómo funciona.

Ahí es donde aparece la conversación de migrar a una plataforma gestionada. Y ahí es donde la mayoría de los planes se rompen, porque se plantean como un corte.

## Por qué no funciona la ventana de mantenimiento

El instinto natural es programar un fin de semana, mover todo y volver el lunes. Con una base de datos a veces se puede. Con una plataforma de eventos, casi nunca.

La razón es que Kafka no es un almacén, es un flujo con estado distribuido en muchos consumidores. Migrar significa mover no solo los datos sino la **posición de lectura** de cada aplicación consumidora, y hacerlo de forma que ninguna reprocese lo que ya procesó ni se salte lo que no.

Con cinco aplicaciones puede salir bien. Con cuarenta, la probabilidad de que las cuarenta queden exactamente donde deben estar en la misma ventana es baja. Y el modo de falla no es un error visible: es un consumidor que quedó atrasado y nadie lo nota hasta que aparece una diferencia en una conciliación tres días después.

## Convivencia en lugar de corte

El enfoque que aplicamos es distinto: los dos clústeres operan en paralelo durante la migración, y los dominios se mueven de uno a otro cuando cada uno está listo.

La pieza que lo hace posible es la replicación entre clústeres. Los topics se replican del origen al destino de forma continua, incluyendo la posición de los consumidores. Eso convierte un evento único e irreversible en una serie de decisiones pequeñas y reversibles.

El orden que nos ha funcionado:

**Primero se replica, sin mover a nadie.** El clúster destino recibe todo y no lo consume nadie. Aquí se valida el rendimiento, la retención, los esquemas y el dimensionamiento real, con tráfico de producción y sin riesgo.

**Después se mueven los consumidores, no los productores.** Un consumidor que lee del destino en lugar del origen es un cambio de configuración reversible. Si algo sale mal, vuelve a apuntar al origen y sigue operando. Se empieza por los consumidores menos críticos.

**Al final se mueven los productores, dominio por dominio.** Este es el paso que hace irreversible el corte para ese dominio, y por eso es el último. Cuando un productor escribe en el destino, sus consumidores ya llevan tiempo leyendo de ahí sin incidentes.

**El origen se apaga cuando ya no lo usa nadie**, no cuando el plan decía que tocaba.

## Lo que se descubre por el camino

Una migración de streaming casi siempre destapa cosas que estaban escondidas. Conviene esperarlas en lugar de que sorprendan.

**Topics que nadie consume.** Es normal encontrar que una parte de lo que se está publicando no lo lee nadie desde hace meses. Migrar es una buena oportunidad para dejar de pagar por moverlos.

**Esquemas que en realidad no existen.** Muchos clústeres open source operan sin registro de esquemas, con el contrato viviendo en la cabeza del equipo que lo escribió. Al pasar a una plataforma con registro de esquemas aparecen las incompatibilidades que estaban latentes. Es mejor que aparezcan en la migración que en un incidente.

**Configuraciones de retención heredadas.** Retenciones de siete días puestas por defecto hace tres años, sobre topics donde el negocio necesitaría treinta, o al revés.

**Consumidores que dependen del orden sin saberlo.** Aplicaciones que funcionan porque las particiones les llegaron en cierto orden y nadie lo documentó.

## Lo que hay que tener antes de empezar

Antes de replicar el primer topic conviene tener tres cosas resueltas.

Un **inventario real de productores y consumidores**, no el diagrama de arquitectura sino quién está conectado hoy. Casi siempre difieren.

Un **criterio de dominio** para decidir qué se mueve junto. Mover por orden alfabético de topic garantiza problemas; mover por dominio de negocio, no.

Una **definición explícita de qué significa que un dominio quedó migrado**, acordada antes y no negociada durante. Sin eso, cada corte se convierte en una discusión.

## Cuánto toma

Depende casi por completo del número de aplicaciones productoras y de qué tan acoplado esté el consumo, no del volumen de datos. Un clúster con mucho tráfico y pocas aplicaciones se migra rápido. Un clúster mediano con decenas de aplicaciones de equipos distintos toma bastante más, porque el trabajo real es de coordinación, no técnico.

Lo que sí se puede afirmar con seguridad es lo contrario del instinto inicial: no hace falta detener la operación. Si el plan de migración requiere una ventana de indisponibilidad para una plataforma de eventos, probablemente el plan esté mal.

LISTA DE CONTROL

# Antes de llevar un agente a producción

Cuatro preguntas que separan los pilotos que llegan a producción de los que se quedan en demo. Si alguna no tiene respuesta clara, el problema probablemente no está en el modelo.

1 **¿De dónde sale el dato que va a ver el agente, y con qué antigüedad llega?**

Si la respuesta es "del almacén, que se carga de noche", el agente está razonando sobre un negocio que ya cambió.

2 **¿Qué pasa si ese dato cambia mientras el agente razona?**

Un agente que consulta una vez y decide después no ve la corrección. Uno que ejecuta dentro del flujo sí.

3 **¿Puedes reconstruir, seis meses después, qué información tenía el agente cuando decidió?**

Sin un log de eventos inmutable no hay auditoría posible, y eso bloquea el paso a producción en cualquier entidad regulada.

4 **Cuando agregues el quinto agente, ¿cuántas integraciones nuevas van a hacer falta?**

Si la respuesta crece con cada agente, el problema es la arquitectura de integración, no el agente.

## Hablemos de tu caso

Llevamos siete años implementando Confluent en América Latina. Hoy somos partners de IBM y trabajamos con Google ADK e IBM watsonx para llevar agentes a producción sobre datos en movimiento.

**Agenda 30 minutos con un arquitecto senior. Revisamos tu caso concreto, te decimos qué haría falta y qué no. Sin presentación de ventas.**

**Agendar diagnóstico de 30 minutos**

O escríbenos por WhatsApp: **+57 300 561 3240**

MÁS SOBRE ESTO

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