archgents

PRACTICAL GUIDE

Data Streaming and AI agents in production

How to take agents that act on real-time data from architecture to deployment

Archgents

2026 edition · Latin America

archgents.com

INTRODUCTION

How to take agents that act on real-time data from architecture to deployment

This guide gathers what we have learned implementing data-in-motion platforms and AI agents across Latin America. It is not an introduction to artificial intelligence, nor a product catalogue: it is written for whoever has to decide the architecture and answer for it once the system is in production.

The four chapters follow the order in which the problems usually appear. First, why an agent wired to data that arrives late makes wrong decisions no matter how much the model is tuned. Then, what the platform underneath that data actually is, now that IBM owns it. Next, what changes when the agent runs inside the event stream itself. And last, how it is migrated to production without stopping the business.

Each chapter stands on its own. At the end there are four control questions to review a project before it goes to production.

CONTENTS

CHAPTER 01

The nightly batch no longer cuts it: why your AI agents need data in motion

An agent wired to a warehouse that refreshes overnight reasons about a business that has already changed. The problem is not the model, it is how the data reaches it.

When an AI agent pilot fails to reach production, the conversation almost always turns to the model. Maybe the prompts need work. Maybe it is time to switch providers. Maybe the model should be fine-tuned on company data. In our experience implementing data platforms across Latin America, the cause usually sits somewhere else, and it is far less glamorous: the agent is looking at stale data.

The agent has no idea it is out of date

A language model has no way of telling whether the data you handed it is three seconds old or fourteen hours old. It will answer with the same confidence either way. That is exactly the property that makes it dangerous when context arrives late.

Take a customer service agent at a bank. You ask about the status of a transfer. The agent queries the data warehouse, which is fed by an overnight job, and reports that the transfer is in progress. In reality it was rejected two hours ago and the customer has already been notified. The agent did not lie: it answered correctly about a state that had stopped being true.

No amount of prompt engineering fixes that.

Why the batch architecture survived so long

Moving data by copying it worked for decades for a simple reason: decisions could wait. The daily sales report was there for the next morning's meeting. Bank reconciliation closed overnight. Nobody needed to know the state of inventory at a precise second.

That assumption broke on two fronts at once.

The first is the business. Fraud is settled inside the transaction, not in the following day's review. The customer abandoning a cart does it within the minute, not within the quarter. The network outage that will generate complaints is already happening while the dashboard refreshes.

The second is the agent. An autonomous agent does not query data to inform a person who will then apply judgment. It queries data in order to **act**. And acting on a state that has already changed is not a reporting error, it is a wrong decision actually carried out.

The trap of wiring the agent straight to the database

The natural reaction once someone sees this is obvious: if the warehouse is stale, let the agent query the transactional database directly.

That works with one agent and one use case. It stops working fast.

- Every new agent opens another connection against the system that keeps the business running. The core banking platform, the inventory system and the billing system were never designed to absorb analytical queries from agents that reason out loud.
- An agent's queries are unpredictable in both shape and volume. A spike in conversations turns into a spike in load on a critical system.
- Every integration is point to point. With ten systems and ten agents you have a hundred possible paths, and all of them have to be maintained.
- There is no record of what the agent saw. When someone asks why it decided what it decided, there is no way to reconstruct it.

That last point is usually what stalls the project in banking and telecommunications. It is not enough for the agent to be right: you have to be able to explain it afterwards.

Putting events at the center

The alternative we implement differs on one point that looks small and changes everything: instead of every consumer going to fetch data where it lives, each system **publishes what happens to it** as an event, and whoever needs it consumes it from there.

An approved payment is an event. An address change is an event. A degraded network cell is an event. A product leaving the warehouse is an event.

Three things the previous architecture could not deliver are resolved on top of that stream:

Context is built while the data is moving. A customer's behavioral profile is not recalculated every night: it stays current as their transactions arrive. By the time the agent asks, the answer is already there and already fresh.

Complexity stops growing with the square. A new system publishes its events once and any consumer picks them up. There is no integration to build for every pair.

Every decision is on the record. If the agent's decision is also published as an event, the full record of what data existed and what was decided on it lives in the same place. It can be audited and it can be replayed.

Where this starts in practice

It does not start with buying a license. It starts by picking a use case where real time changes a measurable outcome, and bringing into the stream only the events that case needs.

We have seen projects that set out to publish the entire core as events and spend six months stuck in modeling. And we have seen projects that start with three event types, solve one concrete fraud case and grow from there on real demand rather than speculative planning.

The second path reaches production. The first usually ends up as a slide deck.

What to review before adding another agent

If you are weighing whether to take an agent to production, these questions separate the pilots that survive from the ones that do not:

1. Where does the data the agent will see come from, and how old is it when it arrives?
2. If that data changes while the agent is reasoning, what happens?
3. Can you reconstruct, six months later, what information it had available when it decided?
4. When you add the fifth agent, how many new integrations will be needed?

If any of the four lacks a clear answer, the problem is not the model.

CHAPTER 02

What IBM bought when it bought Confluent

Almost everyone equates Confluent with Kafka. Kafka is one of seven pieces, and the other six explain why IBM paid eleven billion dollars.

On 17 March 2026 IBM closed its purchase of Confluent at 31 dollars a share, an enterprise value of about eleven billion dollars. Confluent delisted from Nasdaq and became an IBM subsidiary.

The question we have been asked ever since, in meetings and at the executive breakfast we hosted recently, is almost always the same: if Kafka is open source and free, what exactly did IBM buy?

The short answer is that Kafka is one of seven pieces. The long answer is this article.

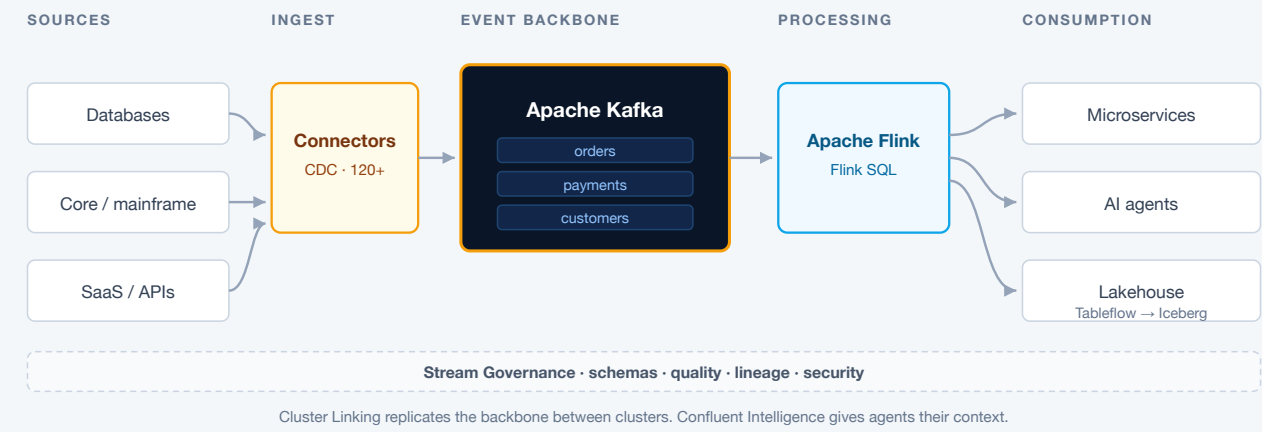
Confluent is not Kafka with support

It is the most common confusion and the most expensive one, because it leads you to compare a licence against zero and conclude the obvious.

Kafka solves one problem: holding events in a distributed, ordered, durable log that many consumers can read at their own pace, and reread when they need to. It is the central piece and it is excellent at its job.

But an event backbone in production needs six more things, and those are what a team ends up building by hand when it does not buy them.

The seven pieces



The seven pieces in one flow: the same colour is the same component in every design.

Apache Kafka is the event log. Ordered, durable, partitioned to scale horizontally, with retention that lets you replay history. It is the single source of truth for what is happening right now.

Connectors solve the connection to the real world. Wiring the core, dozens of databases and whichever SaaS is in play into the backbone usually means bespoke code that is fragile and expensive to maintain. Here it is more than a hundred and twenty ready connectors, with native change data capture to read from the core without touching it. Configuration, not development.

Apache Flink processes in flight. Raw events rarely serve on their own: they have to be joined, enriched and computed on at the moment, not in a batch query afterwards. Flink SQL makes that logic declarative and versionable, with results in milliseconds.

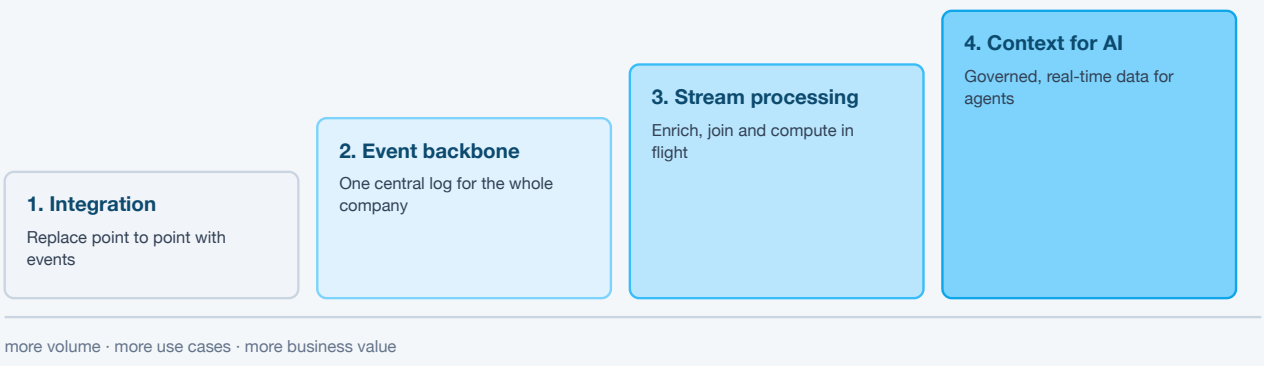
Stream Governance is the piece most people underestimate, and the one that decides whether the project reaches production in a regulated institution. Without control, an event backbone turns into a mess of formats that nobody trusts. Here there is a schema registry with versioned contracts, quality rules, end-to-end lineage, encryption and access control.

Tableflow turns streams into Iceberg or Delta tables automatically. Analytics and AI need the same data the business runs on, and duplicating it with an overnight ETL is expensive, slow and leaves the two views out of step. One copy of the data serves both.

Cluster Linking replicates topics between clusters, live. A single cluster is a single point of failure, and hybrid or multi-region setups need the same data available in several places at once. It enables active-active, disaster recovery and the bridge between on-premise and cloud with no ETL in between.

Confluent Intelligence is the newest piece and the one that explains IBM's interest. An agent hallucinates when it lacks fresh, trustworthy context about the business, and it cannot act on what is happening now. This layer gives the agent live context and lets it execute inside the event stream itself, over governed and traceable data.

How a platform like this matures



The same pattern matures in layers, and each level unlocks more use cases on the same infrastructure.

Nobody starts at the top layer, and trying to is the most common mistake.

1. **Integration.** Point-to-point connections are replaced by events. The goal is to stop having N systems talking to N systems.
2. **Event backbone.** The log stops belonging to one project and starts belonging to the company.
3. **Stream processing.** Enriching, joining and computing in flight on top of that backbone.
4. **Context for AI.** Agents consume governed, real-time data, and act on it.

Each layer unlocks more use cases and more volume on the same infrastructure. Decoupled microservices, live fraud scoring, customer 360, core modernization through change data capture, risk and exposure computed instantly, personalization at the moment the customer acts. They all run in parallel on the same backbone, without duplicating the platform.

What the platform gains with IBM

IBM's announcement is clear about its intent: **Confluent streams live operational events directly into watsonx.data**, so that every model, agent and workflow runs on continuously updated enterprise data. That is the thesis of the purchase, and it lines up with the fourth layer above.

If you already run Confluent, or you are evaluating whether to start, here is what changes in practice.

The platform now has IBM behind it. Eleven billion dollars is a large bet and it is a signal. Confluent stops answering to a quarterly market and becomes part of the data portfolio of a company with decades of enterprise presence, global contracts and support on the ground in the region.

The path to agents gets shorter. The watsonx.data integration is exactly the piece teams used to build by hand: carrying live business context all the way to the model. It now comes with the platform, and it comes governed.

The foundation stays open source. Kafka and Flink are Apache Software Foundation projects. That did not change with the acquisition and it is not going to. The event architecture you design is yours and it stays portable, and no corporate decision locks you in.

Existing contracts run their course. Multi-year agreements run to their term. Renewal is the natural moment to review the joint roadmap, and that is when it helps to sit down with someone who knows both houses.

Product leadership moved up a level. In August 2026 Jay Kreps, co-founder and creator of Kafka, handed the role to Shaun Clowes, until then chief product officer, who is now general manager for Confluent and Data at IBM. That is a good signal: data streaming did not become one more line in the catalogue, it got its own seat in IBM's data strategy.

Where to start

The acquisition does not change the technical analysis. The seven pieces are the same, the architectural pattern is the same and so are the use cases it unlocks. What changes is that there is more weight behind it now, and a shorter path to agents.

This is how we do it, and it works:

1. **Assessment.** We pick one high-impact use case. One, not a three-year plan.
2. **Architecture design.** We model the solution on the pieces your case actually needs, and no more.
3. **A governed proof of concept.** On Confluent Cloud, in weeks, with schemas and lineage from day one.

We have spent seven years implementing Confluent across Latin America, and today we are IBM partners. It is the same architecture we were building before the deal, now with a bigger partner behind it.

Sources: the closing announcement is on IBM's newsroom, and the original acquisition announcement is on Confluent's blog.

CHAPTER 03

Streaming Agents: agents that run inside the data stream

When the agent runs as a Flink job instead of a separate service, it stops asking for context and starts having it. What changes, and when it makes sense.

The usual way to build an agent is as a service: it receives a request, queries what it needs, reasons and answers. It is the model we inherited from web applications and it works well when somebody is asking a question.

There is a second way, less well known, that changes where the agent lives: instead of being a service you query, the agent is a **process that runs inside the event stream**. Nobody invokes it. It activates when the event it is responsible for occurs.

The difference is not about infrastructure

At first glance this looks like a deployment decision. In practice it changes three fundamental things.

The agent stops asking for context. A service-style agent that needs a customer's history issues a query and waits. An agent running inside the stream already has that state at hand, because the processing engine has been maintaining it with every event that went by. The difference between querying and having is typically hundreds of milliseconds per decision, which at production volume is not a detail.

The agent reacts instead of waiting. A service does nothing until somebody calls it. That means the process only begins when a human or a system decides to start it. When the agent lives in the stream, the trigger is the fact itself. Network degradation does not wait for the customer to call before someone looks at it.

The agent scales the way the stream scales. The engine already knows how to distribute work by partition and guarantee that events for the same entity are handled by the same task. That is exactly the distribution a stateful agent needs, and it comes solved.

What you gain in traceability

There is a less obvious benefit that in regulated sectors is usually the deciding one.

When the agent runs inside the stream, both the data it saw and the decision it made are events in the same log. That enables something very hard to do with a service-style agent: **replaying the case**. You can go back to the exact position in the stream, run the version of the agent that was live that day and see why it decided what it decided.

With a service-style agent, that reconstruction requires having separately logged the request, the response from every system it queried and the model version, and trusting that the three records line up. They almost never do.

When it is the wrong fit

This pattern does not replace the conversational agent, and forcing it where it does not belong is an expensive mistake.

It is the wrong fit when **a person initiates the interaction**. If the use case is somebody typing in a chat, the service-style agent is the right form.

It is the wrong fit when **the reasoning is long and multi-step**. An event stream is optimized to process a lot at low latency. An agent that is going to make fifteen tool calls and take forty seconds does not fit that model, and putting it there creates backpressure upstream.

It is the wrong fit when **volume is low**. At a hundred events a day, the operational complexity is not worth it.

The practical rule: if the trigger is a business fact, volume is high and the decision has to come out fast, the agent belongs in the stream. If the trigger is a person and the reasoning is open-ended, it belongs as a service.

The two running side by side

In most real implementations both end up running, and that is the healthy architecture.

The agent in the stream does the continuous work: it evaluates every transaction, keeps context up to date, detects what falls outside the norm and triggers action when it should. The conversational agent handles the person, and when it needs to know the state of the business it queries the context the first one has been maintaining.

Put another way: one builds the truth about the present, the other explains it. That is a considerably cleaner division of labor than asking a single agent to do both.

Where to start

If you already have an event platform running, the first case is usually the most boring one on purpose: take a rule that runs in batch today and move it to the stream, with no agents involved. That validates the state model, the sizing and the operational side.

With that working, adding reasoning on top of the same stream is an incremental step rather than a leap. The most common mistake we see is the reverse: starting with the agent and finding out later that the data platform it needed was not ready.

CHAPTER 04

How to migrate from open source Apache Kafka to Confluent without stopping the business

A streaming platform migration is not done in a maintenance window. It is done through coexistence, domain by domain, with the option to roll back at every stage.

Plenty of organizations in the region arrived at Apache Kafka the right way: a capable team stood it up to solve a specific problem, it worked, and it gradually became critical infrastructure without anyone ever formally deciding that it should be.

The breaking point comes when the cluster that started as one team's project is holding up processes that cannot go down, and the organization realizes the whole operation depends on the two or three people who know how it works.

That is when the conversation about moving to a managed platform starts. And that is where most plans fall apart, because they are framed as a cutover.

Why the maintenance window does not work

The natural instinct is to schedule a weekend, move everything and come back on Monday. With a database that sometimes works. With an event platform, almost never.

The reason is that Kafka is not a store, it is a stream with state distributed across many consumers. Migrating means moving not just the data but the **read position** of every consuming application, and doing it so that none of them reprocesses what it already handled or skips what it has not.

With five applications it can go well. With forty, the odds that all forty land exactly where they should inside the same window are low. And the failure mode is not a visible error: it is one consumer left behind that nobody notices until a discrepancy shows up in a reconciliation three days later.

Coexistence instead of cutover

The approach we take is different: both clusters run in parallel during the migration, and domains move from one to the other when each one is ready.

The piece that makes this possible is replication between clusters. Topics are replicated continuously from source to target, including consumer positions. That turns a single irreversible event into a series of small, reversible decisions.

The order that has worked for us:

Replicate first, move nobody. The target cluster receives everything and nobody consumes from it. This is where you validate performance, retention, schemas and real sizing, with production traffic and no risk.

Move the consumers next, not the producers. A consumer reading from the target instead of the source is a reversible configuration change. If something goes wrong, point it back at the source and it keeps running. Start with the least critical consumers.

Move the producers last, domain by domain. This is the step that makes the cutover irreversible for that domain, which is exactly why it comes last. By the time a producer writes to the target, its consumers have been reading from there for a while without incident.

The source is shut down when nobody is using it any more, not when the plan said it was due.

What you find along the way

A streaming migration almost always uncovers things that were hidden. Better to expect them than to be surprised by them.

Topics nobody consumes. It is normal to find that part of what is being published has not been read by anyone in months. A migration is a good moment to stop paying to move it.

Schemas that do not actually exist. Many open source clusters run without a Schema Registry, with the contract living in the heads of the team that wrote it. Moving to a platform with a Schema Registry surfaces the incompatibilities that were sitting there latent. Better they surface during the migration than during an incident.

Inherited retention settings. Seven-day retention set by default three years ago, on topics where the business would need thirty, or the other way around.

Consumers that depend on ordering without knowing it. Applications that work because the partitions happened to reach them in a certain order and nobody ever wrote it down.

What you need before you start

Before replicating the first topic, three things are worth having settled.

A **real inventory of producers and consumers**, not the architecture diagram but who is actually connected today. They almost always differ.

A **criterion for grouping domains** to decide what moves together. Moving in alphabetical order by topic guarantees problems; moving by business domain does not.

An **explicit definition of what it means for a domain to be migrated**, agreed up front rather than negotiated in flight. Without it, every cutover turns into an argument.

How long it takes

It depends almost entirely on the number of producing applications and how coupled the consumption is, not on data volume. A cluster with heavy traffic and few applications migrates quickly. A mid-sized cluster with dozens of applications from different teams takes considerably longer, because the real work is

coordination, not engineering.

What can be said with confidence is the opposite of the initial instinct: you do not need to stop the business. If a migration plan requires a downtime window for an event platform, the plan is probably wrong.

CHECKLIST

Before taking an agent to production

Four questions that separate the pilots that reach production from the ones that stay demos. If any of them has no clear answer, the problem is probably not the model.

1 **Where does the data the agent sees come from, and how old is it when it arrives?**

If the answer is "from the warehouse, loaded overnight", the agent is reasoning about a business that has already changed.

2 **What happens if that data changes while the agent is reasoning?**

An agent that queries once and decides later never sees the correction. One that runs inside the stream does.

3 **Can you reconstruct, six months later, what the agent knew when it decided?**

Without an immutable event log there is no audit trail, and that blocks production in any regulated institution.

4 **When you add the fifth agent, how many new integrations will that take?**

If the answer grows with every agent, the problem is the integration architecture, not the agent.

Let us talk about your case

We have spent seven years implementing Confluent across Latin America. Today we are IBM partners, and we work with Google ADK and IBM watsonx to take agents to production on top of data in motion.

Book 30 minutes with a senior architect. We look at your actual case and tell you what it would take, and what it would not. No sales deck.

Book a 30-minute diagnostic

Or reach us on WhatsApp: **+57 300 561 3240**

MORE ON THIS

Data Streaming with Confluent	archgents.com/en/data-streaming/
Agents with Google ADK	archgents.com/en/google-adk-agents/
Agents with IBM watsonx	archgents.com/en/ibm-watsonx-agents/
Use cases by industry	archgents.com/en/use-cases/