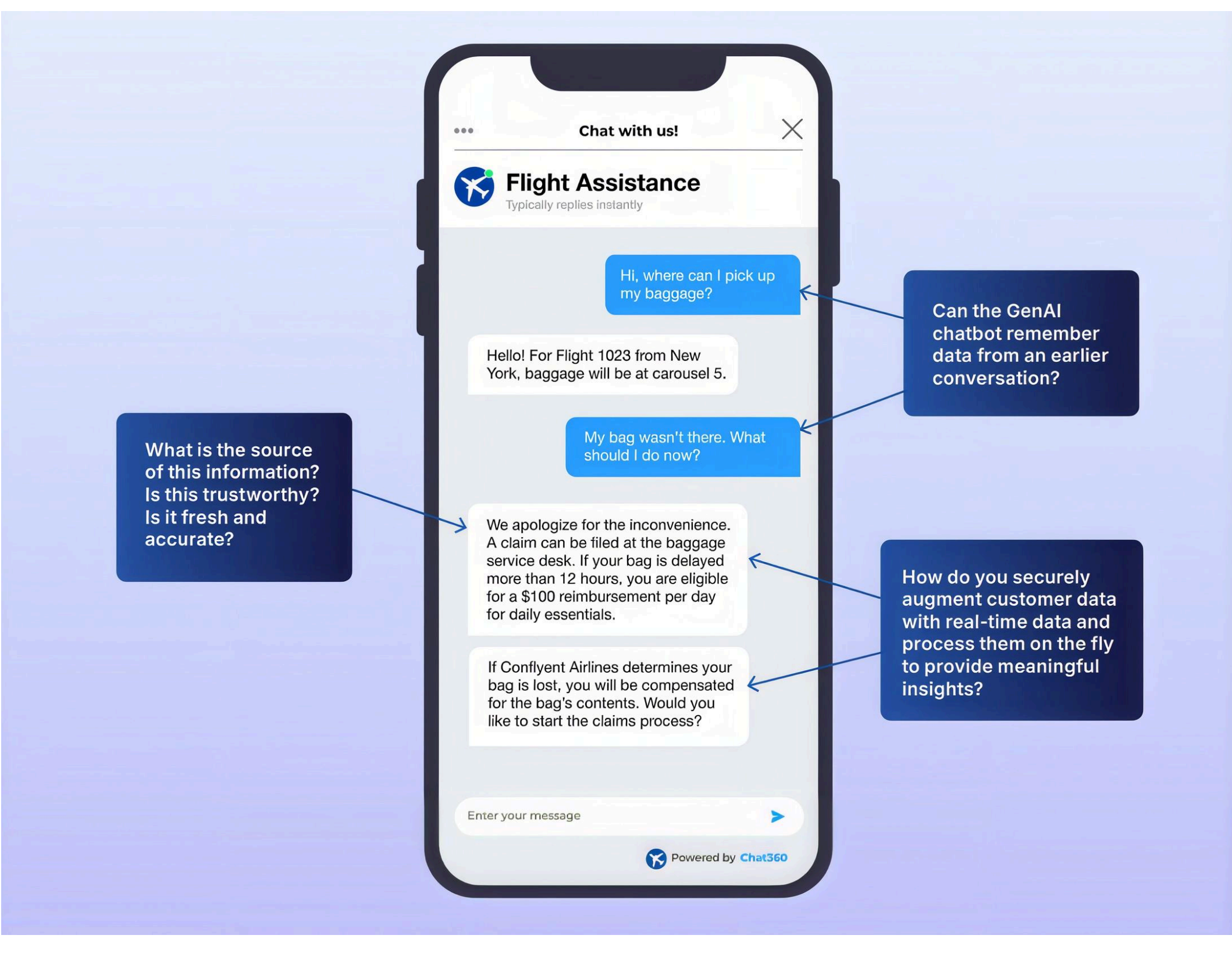
How to Build Retrieval-Augmented Generation (RAG) for Real-Time AI Applications

1 What Is RAG

Picture an airline chatbot. A passenger types: "Where's my bag?" The bot needs to know which flight the person was on, what the airline's lost-bag policy says, and whether this passenger already filed a claim last week.

Picture an airline chatbot. A passenger types: "Where's my bag?" The bot needs to know which flight the person was on, what the airline's lost-bag policy says, and whether this passenger already filed a claim last week.



This is the problem RAG solves. RAG stands for retrieval-augmented generation. In plain terms: before the AI answers, you hand it the right facts first. It reads those facts, then writes the answer.

Without this step, AI models tend to guess. And a confident guess that's wrong is called a hallucination. It sounds right. It reads well. It's just false. That's the real risk with AI chatbots left to answer from memory alone.

RAG fixes this by pulling in fresh, specific facts right before the model answers. Here's how it usually works.

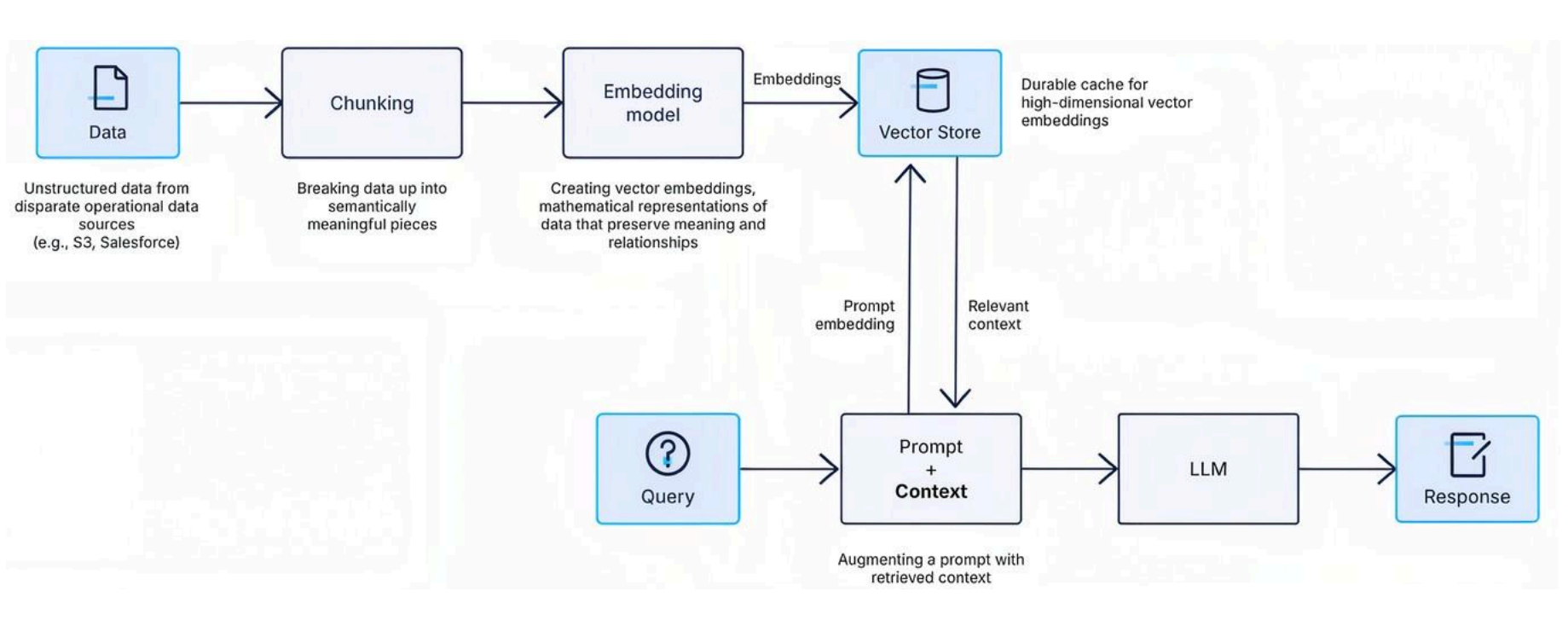
A simple version of the process:

Your documents get broken into small chunks. Each chunk gets turned into a kind of numeric fingerprint (called an embedding) that captures its meaning. These fingerprints get stored in a special database built for this, called a vector store.

When a question comes in, it gets the same treatment. It becomes a fingerprint too. The system then searches the vector store for chunks whose fingerprints look similar. Not exact word matches. Similar meaning. That's the part that makes RAG different from a normal database search.

Whatever chunks come back get handed to the AI model along with the original question. The model reads both and writes an answer based on what it already knows plus what it was just given.

Companies do have other options besides RAG. They could retrain the whole model on their own data. But that's expensive, slow, and risky. Retrain a model wrong and it can actually forget things it used to know. RAG skips all of that. You're not changing the model. You're just feeding it better notes right before it speaks.



Before any of this works well, though, a company usually has to fix a few things first:

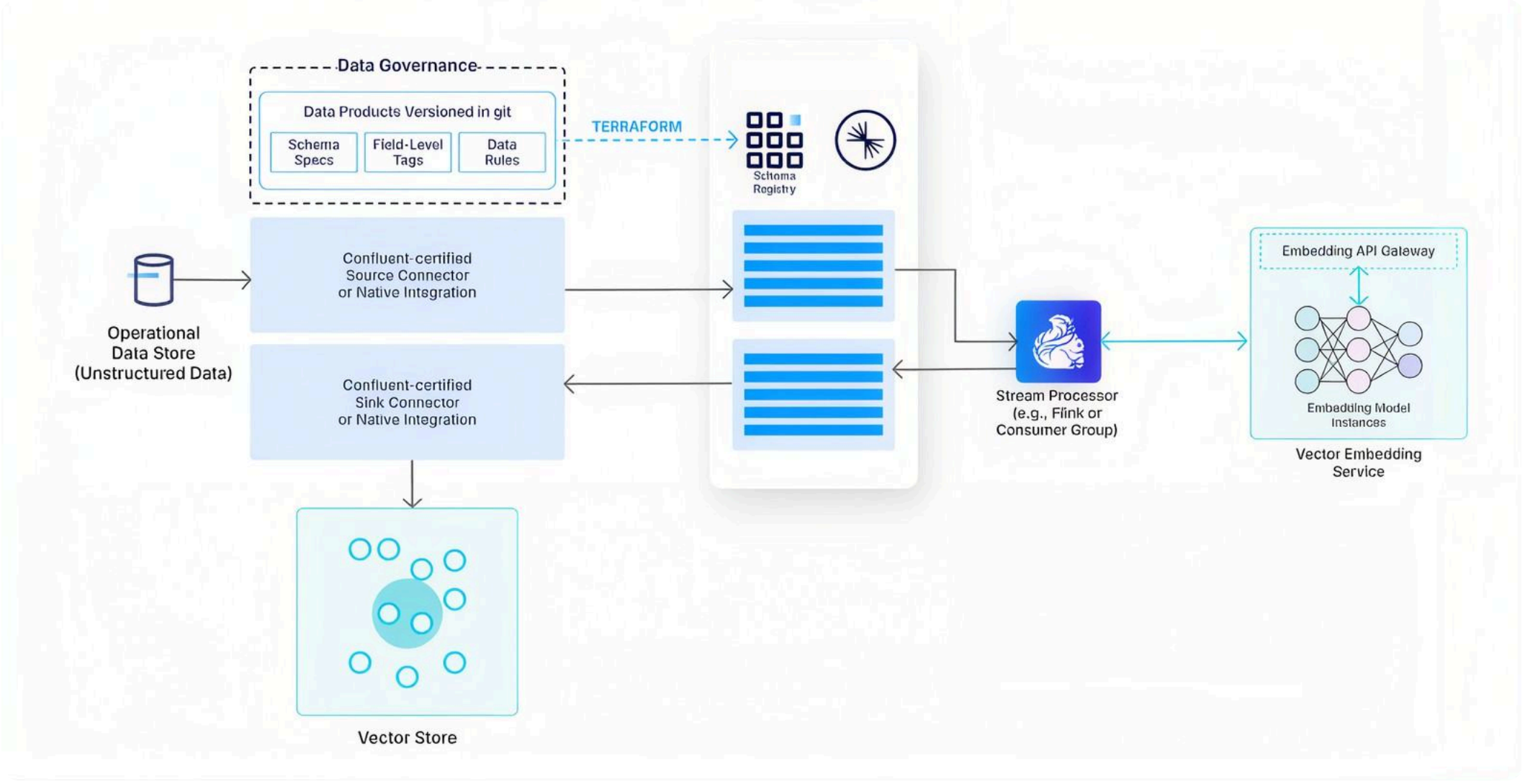
- Data that's stuck in batch jobs and only updates once a day (or once a week)
- Information scattered across ten different systems that don't talk to each other
- Data that's messy, inconsistent, or just wrong in places
- No clear record of where the data came from or who touched it last
- Systems that are too tightly wired together to change easily

This is where a proper data streaming setup earns its keep.

2 RAG: Data Augmentation

This step is about building the knowledge base your AI model will draw from.

Most companies have data sitting in a dozen different places. A support ticket system. A product database. Old PDFs on a shared drive. Ready-made connectors can pull from all of these without your team writing custom code for each one. This replaces the old batch-job approach, where data might sit stale for a day before anyone sees it.



Raw data rarely comes in ready to use, though. It needs cleaning first. Maybe you're joining a customer's profile with their order history to build one useful record instead of two scattered ones. Once that work is done, other teams can reuse it instead of redoing the same cleanup themselves.

From there, the cleaned data gets broken into chunks and turned into those numeric fingerprints (embeddings) mentioned earlier, then saved to a vector store built to search by meaning instead of exact words.

Not everything belongs in a vector store, though. This trips people up more than you'd expect.

Good candidates for a vector store: Documents that are mostly plain text. Files that mix text and tables, like a research report. Spreadsheets where one column is customer comments or reviews.

Bad candidates: A plain list of ID numbers has no real meaning to search by. Data you only need for counting or averages (like "what percent of visitors clicked this button") belongs in a normal database, not a vector store. Search by meaning doesn't help you there.

One more wrinkle: numbers on their own don't mean much to an AI model. A price of "450" tells the model nothing unless it knows that's expensive for this category. Reframing that field as "expensive" or "budget-friendly" before it goes in often works better than the raw number.

Good candidates for a vector store:	Bad candidates:
Documents that are mostly plain text. Files that mix text and tables, like a research report. Spreadsheets where one column is customer comments or reviews.	A plain list of ID numbers has no real meaning to search by. Data you only need for counting or averages (like "what percent of visitors clicked this button") belongs in a normal database, not a vector store. Search by meaning doesn't help you there. One more wrinkle: numbers on their own don't mean much to an AI model. A price of "450" tells the model nothing unless it knows that's expensive for this category. Reframing that field as "expensive" or "budget-friendly" before it goes in often works better than the raw number.

A quick look at how the technical side works, in case your engineering team needs it:

A connection gets set up first, pointing to wherever the embedding service lives:

```
create connection openai-vector-connection
--cloud aws
 --region us-west-2
 --type openai
 --endpoint 'https://api.openai.com/v1/embeddings'
 --api-key 'secret_key'
```

Then a model gets defined using that connection, and new records get run through it automatically:

```
CREATE MODEL `openai_embeddings`
INPUT (input STRING)
OUTPUT (vector ARRAY<FLOAT>)
WITH (
  'TASK' = 'embedding',
  'PROVIDER' = 'OPENAI',
  'OPENAI.CONNECTION' = 'openai-vector-connection'
);

INSERT INTO customer_service_logs_embeddings
SELECT * FROM customer_service_logs
LATERAL TABLE (ML_PREDICT(openai_embeddings, input));
```

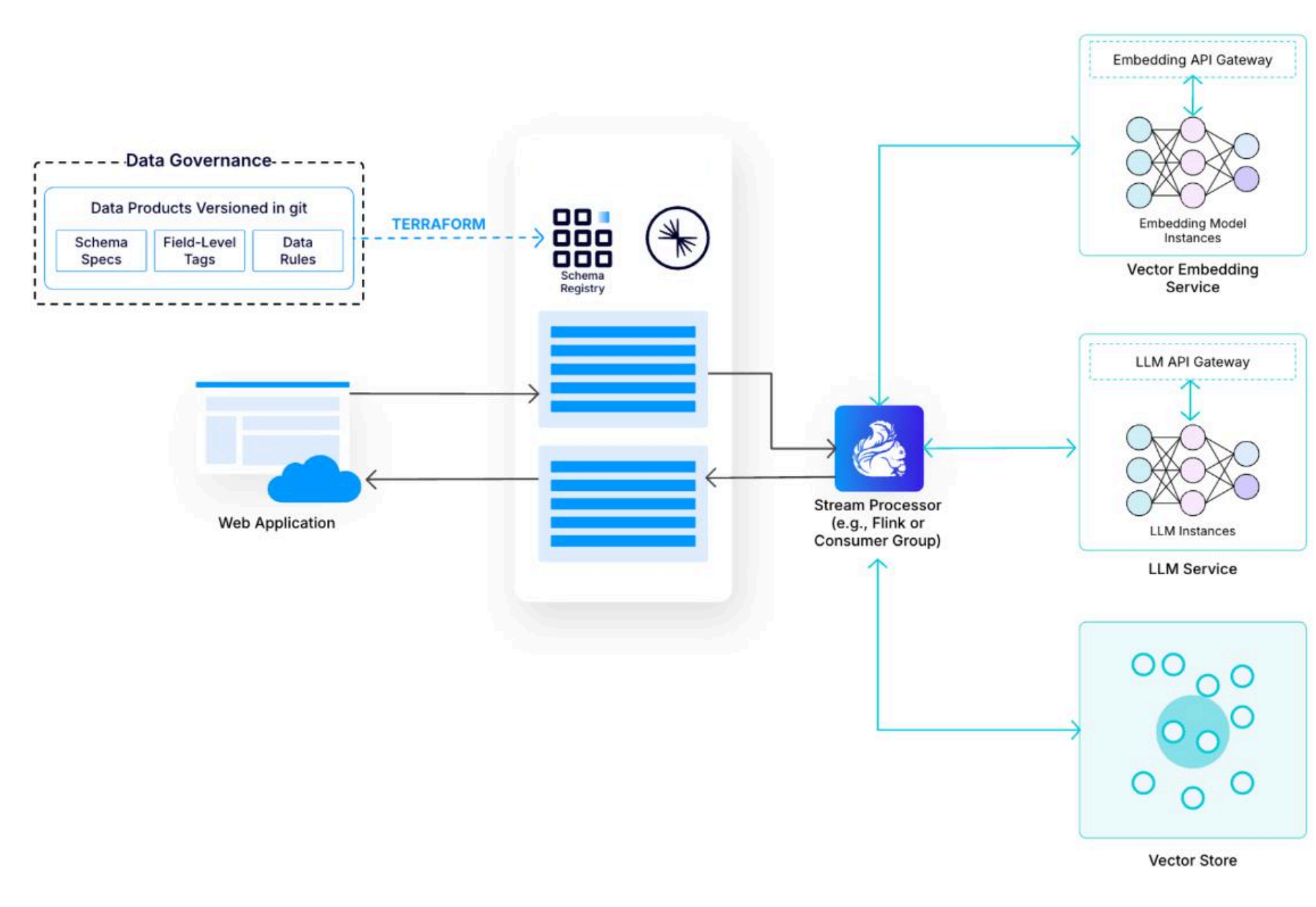
As more of these pipelines get connected, good governance starts to matter a lot. You want a clear record of where each piece of data came from, who's allowed to see it, and whether it's actually trustworthy. Tagging, encrypting, and logging access aren't nice extras here. They're what keeps a growing system from turning into a mess nobody trusts.

That trust actually pays off in the numbers. Studies suggest around 90% of the data inside a typical company is unstructured (emails, PDFs, chat logs, not tidy spreadsheet rows). Once that pile of messy information is searchable by meaning, it stops being dead weight and starts being something your AI model can actually use.

3 RAG: Inference

Now that the knowledge base exists, the system needs to use it the moment a real question comes in.

Here's what happens behind the scenes. A user asks something in the chatbot. That question gets enriched on the spot: pulled together with relevant records from other systems, plus whatever the vector store finds as a match. All of that gets bundled and sent to the AI model. The model's answer flows straight back to the chatbot.



This setup keeps the pieces loosely connected. Standard formats mean the team building the chatbot doesn't need to wait on the team managing the vector store, and vice versa. One event triggering another, automatically, without a person in the loop babysitting it.

A simplified flow:

User asks a question → Chatbot → Enrichment step → AI model call → Answer sent back

Enrichment step ⇅ Vector store (finds matching context)

Keeping teams decoupled like this has a real payoff. The web team, the data team, and the model team can each move at their own pace. Shared rules (like "ID numbers must always be this many digits") keep everyone's work compatible without constant meetings to coordinate it.

Some of the cleanup happens automatically too, like scrubbing a credit card number out of a record before it's stored, or turning a pile of raw reviews into something the model can actually search through. There are also ready-made tools for common jobs like removing duplicate entries. Point one at a stream of claims, and it'll spit out a clean version with duplicates gone, no custom code required.

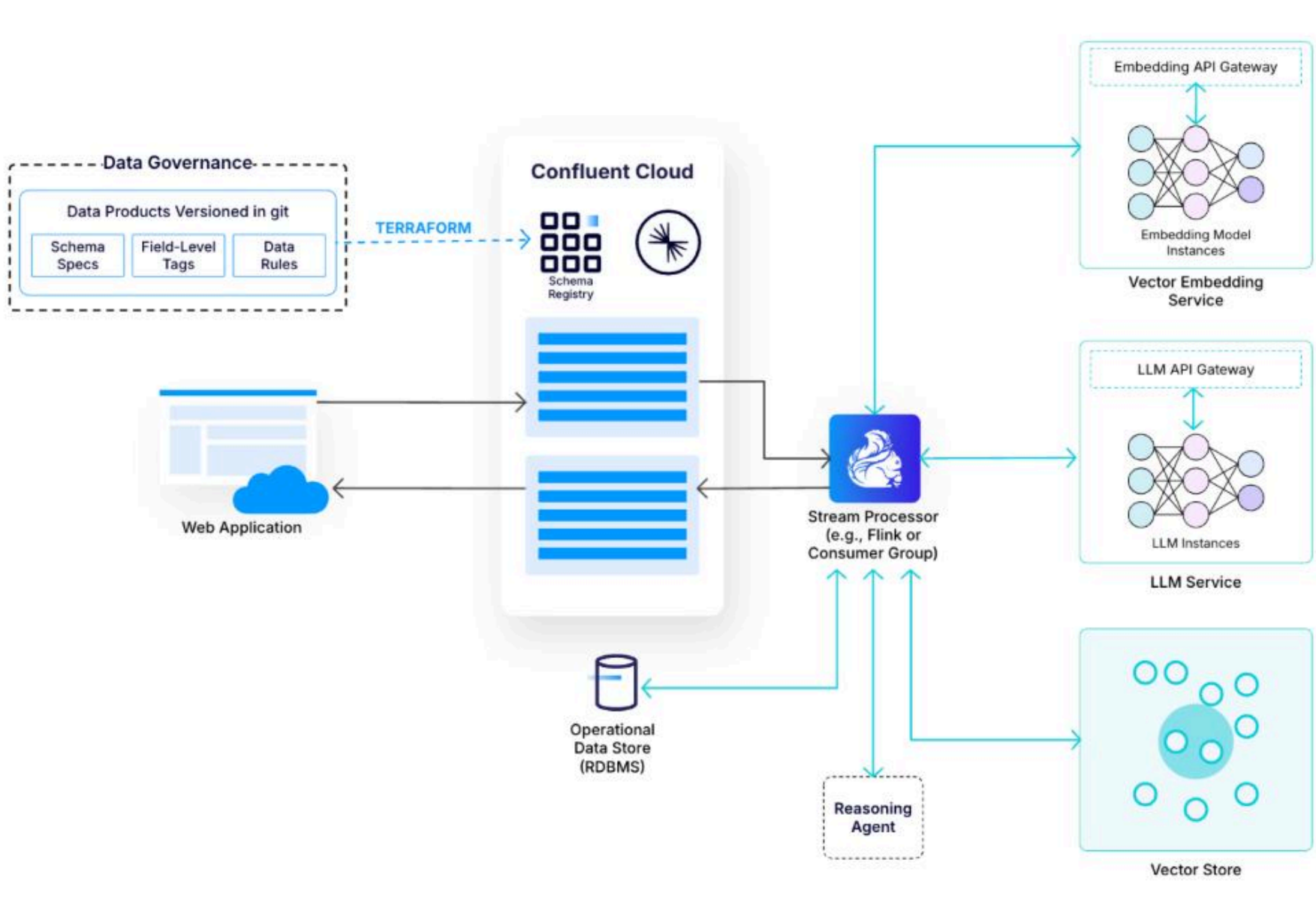
All of this adds up to fewer wrong answers, because the model is working from tighter, more relevant information instead of a pile of noise.

4 RAG: Workflows

Some questions aren't simple. Here's one: "Can I use my frequent flyer miles or credit card points to upgrade my flight and add a second checked bag?"

That's really four questions stitched together. And AI models tend to do better with four small questions than one big tangled one.

This is where a reasoning agent comes in. Think of it as a coordinator. It looks at a complicated question, breaks it into steps, and figures out which tool or data source to check for each piece. Search a document? Query a database? Call an API? The agent decides, one step at a time.



Here's how that upgrade question might actually get handled:

First, check flight schedules to see which flights even have upgrade seats open, and what they'd cost in miles.

Second, check the airline's baggage policy document for this passenger's ticket class.

Third, call a service to check the exchange rate between miles and credit card points, then check if the passenger actually has enough of either.

Finally, hand all of that to the AI model so it can put together one clear, complete answer.

Each of those steps might hit a different system. The reasoning agent is what stitches the results together into something coherent instead of four disconnected facts.

One trick worth mentioning: caching. If someone already asked almost the same question an hour ago, there's no reason to run the whole process again. Pulling a recent answer from a cache is faster and cheaper than calling the AI model fresh every time. Good systems check the cache first and only call the model if they need to.

Security matters here too, maybe more than anywhere else in the pipeline, since this step often touches multiple systems at once. Sensitive fields get filtered out along the way. Every step gets logged, so if something goes wrong, someone can trace exactly what happened.

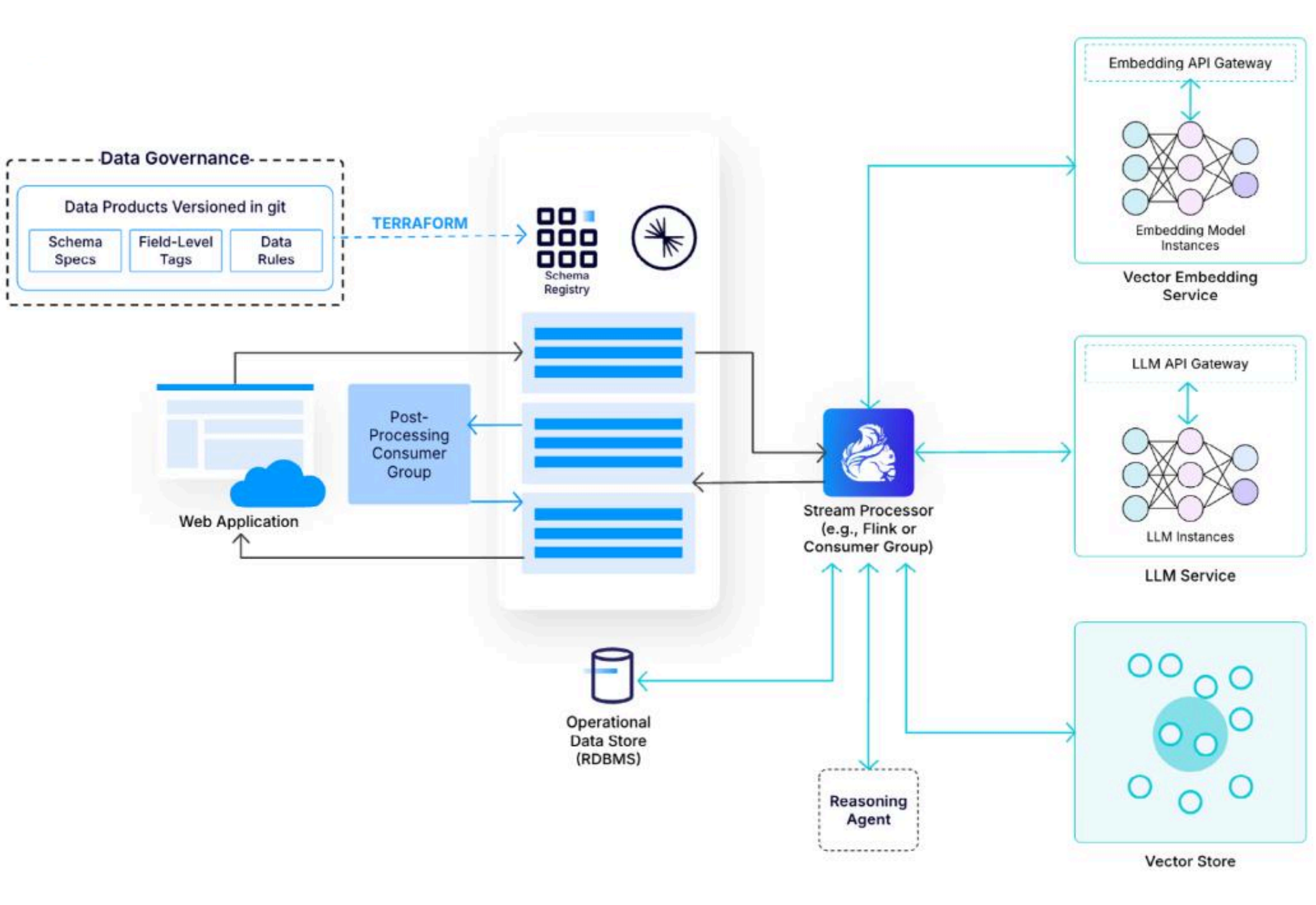
At the end of the day, this step is a balancing act. Fast and roughly right, or slower and closer to perfect? Different situations call for different answers. A VIP customer's request might be worth the extra two seconds for accuracy. A casual browsing question probably isn't.

5 RAG: Post-Processing

Even with all the setup above, AI models still make mistakes sometimes. So the last step is a check: does this answer actually hold up?

This check runs separately from the main chatbot logic. That's on purpose. The team writing compliance rules usually isn't the same team building the chatbot, and this separation means one team can change their rules without breaking the other team's code.

For an airline, this might mean a separate check on baggage pricing, or a rule that verifies a refund amount matches policy before it's shown to the customer.



A checking process often runs in stages:

First, a rules check flags anything that breaks a known policy.

Second, a deeper check verifies specific details, like whether a ticket number or claim ID is actually real and formatted correctly.

Third, more checks can be layered in over time as new edge cases show up.

Each of those steps might hit a different system. The reasoning agent is what stitches the results together into something coherent instead of four disconnected facts.

What that kind of rule looks like in practice:

```
SELECT
  claim_id,
  account_id,
  amount,
  location,
  claim_time
FROM
  transactions
WHERE
  amount > defined_threshold
  OR location IN (defined_blacklist)
  OR claim_time NOT BETWEEN defined_normal_hours_start
  AND defined_normal_hours_end;
```

Every claim runs through this check as it comes in. Flagged claims get routed somewhere useful, like an alert dashboard or a system that pauses the claim automatically until a human looks at it.

Speed versus accuracy comes up again here too. A VIP customer might need a near-perfect answer even if it takes a few extra seconds. A routine question can afford to move faster. That trade-off should be a setting your team controls, not something baked in and fixed forever.

6 Use Cases of RAG in Action

Product Recommendation Engine

Retail data (what's in stock, what's selling, what customers are saying in reviews, loyalty points) gets streamed in, processed, and turned into searchable embeddings. Combine that with a shopper's profile and the system can recommend products that actually fit, not just generic bestsellers.

Real-Time Sentiment Tracking for Stock Markets

Live financial news and market data get pulled in continuously and matched against relevant context in a vector store. The payoff is fast, specific insight: how the market's reacting to a company's earnings call today, what people are actually saying about a product this week, or whether a competitor's move is shifting sentiment.

Jobs Portal

Someone uploads a resume, and the system pulls out the useful parts (skills, experience, background) automatically. From there it can suggest relevant openings and tailor career advice. On the other side, employers get help writing job posts and matching them to strong candidates already in the system.

Customer Support Knowledge Base

Support tickets, product manuals, and past chat transcripts get streamed in and indexed the same way. When a customer asks a question, the system searches that whole history for the closest match instead of making an agent dig through old tickets by hand. New product updates and policy changes flow in automatically too, so the answers a customer gets today reflect this week's policy, not last quarter's. Agents get a drafted answer with the source ticket attached, and can edit before sending instead of writing from scratch.

7 Getting Started

Reading about RAG is one thing. Getting it working on your product's real data, without weeks of trial and error, is another.

This is exactly the kind of build iTechnolabs takes on for founders and product teams who need AI that actually knows their business, not just a chatbot wrapped around a generic model. From setting up the data pipeline to wiring in the vector store and getting the answers checked before they reach a user, our team handles the parts that usually eat up the most engineering time.

About iTechnolabs

iTechnolabs is ISO 27001 and ISO 9001 certified, with 500+ apps delivered and teams across Canada and the US. If your product needs real-time AI features built on your own data, book a discovery call and we'll walk through what that build would actually look like for you.

To learn more, please visit: www.itechnolabs.ca