fresh gravity™
Driven by Data. Inspired by Innovation.

WHITEPAPER

CHOOSING THE RIGHT

Enterprise Integration Interface for AI Ecosystems

WRITTEN BY
SIDDHARTH MOHANTY
DIRECTOR, DATA MANAGEMENT



The Problem: Three Wires, One AI Ecosystem

MCP, REST, or an Event Bus? The honest answer is "it depends on what's flowing and who is asking/consuming." Here's how to reason about it with things we as technical/solution/enterprise architects deploy for our data platform, medallion architecture, and agentic applications examples.

As enterprise AI moves from experimentation into production, integration becomes an architectural decision: how should models and agents interact with data platforms, applications, services, and clinical trial event streams? The choice affects latency, scalability, governance, auditability, cost, and how easily the architecture can evolve.

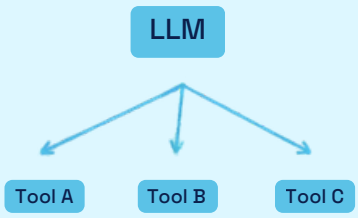
Three interface paradigms dominate enterprise AI integration today:

MCP

Agentic / Tool-Calling

Model Context Protocol - a standardized, tool-calling interface purpose-built for LLMs and agents to discover and invoke capabilities dynamically.

"What can I do right now?"



Dynamic discovery, variable / non-deterministic invocation, per-call reasoning

Agent decides what to call

REST API

Synchronous Request / Response

Synchronous request/response - the deterministic workhorse of service-to-service integration for three decades.

"Give me this specific thing"



Deterministic contract, fixed schema, blocking call, predictable cost per call

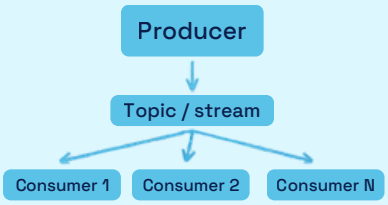
Caller knows exactly what it gets

EVENT BUS

Async Publish / Subscribe

Asynchronous publish/subscribe - Kafka, Event Hub, Service Bus-style backbones for decoupled, high-throughput data movement.

"Tell everyone who cares that this happened"



Decoupled, durable, high-throughput, fan-out; producer never blocks on consumer

Nobody waits for anybody

They are not competitors so much as three different answers to three different questions. Most mature AI ecosystems end up using all three - the skill is knowing the boundary lines.

Figure 1 - The three integration paradigms: who initiates, who waits, and what is exchanged.

1.1 Why This Deserves Real Architecture Attention

Before picking a wire, it is worth being honest about why this keeps coming up as a genuine design question rather than a solved one. Scale is part of it, but agentic AI also introduces failure modes that classic service-to-service integration never had to plan for.

A The immediate caller may no longer be a developer-defined workflow.
In a conventional integration, an engineer writes the exact call a service will receive, once, in advance. In an agentic system, the model itself decides at runtime which tool to reach for and with what parameters, so the interface has to be something a model can discover and reason about on the fly, not a hardcoded call somebody wired up ahead of time.

B The request may be generated dynamically rather than predetermined entirely in application code. Model output is inherently non-deterministic and occasionally malformed, so whatever sits between the model and the tool has to validate, guard, and gracefully recover from inputs a human-written client would rarely, if ever, produce. That is a category of defensive engineering classic REST integrations were never built to absorb. It also ends up introducing a layer of variability to the integration call itself.

C The tool count refuses to stay small.
A production agent ecosystem realistically touches dozens, sometimes hundreds, of internal and external systems. Hand-rolling a bespoke integration for every one of them doesn't survive contact with year two of the roadmap, which is exactly the pressure that pushes teams toward a shared, discoverable protocol instead of one-off glue code per tool.

D No single interaction style covers the whole workload.
Some calls need an answer immediately, some kick off work that finishes later in the background, and some are really about reacting to something that already happened elsewhere in the system. Trying to force all three through one paradigm is usually where an integration layer starts to strain.

This is precisely why MCP, REST, and event buses coexist instead of competing.

MCP was shaped by the first two pressures above, REST by three decades of needing a stable, testable contract, and event buses by the last one - many independent consumers reacting to the same fact at their own pace.

What Each Interface Is Actually Optimized For

2.1 MCP (Model Context Protocol)

MCP

Variable, agent-directed invocation

MCP standardizes how an LLM or agent discovers "tools" (functions, data sources, retrieval endpoints) at runtime and decides, based on reasoning, which ones to call and in what order. MCP is particularly useful when an AI application needs to discover and invoke tools dynamically. In agent-directed implementations, the tools selected and sequence of calls may vary based on the model's reasoning and available context. This is what makes MCP powerful for agentic workflows, at the same time, making it unsuitable for anything requiring a fixed, auditable execution path without additional guardrails.

WORTH FLAGGING

These two layers are rarely an either/or in practice. An MCP server may still invoke an existing REST API or other underlying service to perform the requested operation. In such architectures, MCP provides the standardized discovery and invocation layer while the underlying service retains its existing interface. We recommend treating MCP as the discovery and invocation layer, not a replacement for the transport doing the real work underneath it.

2.2 REST API

REST API

Deterministic, contract-bound request/response

REST (or gRPC, for that matter) gives a fixed contract: a known endpoint, a known schema, a known set of status codes. Cost per call is predictable, latency is bounded, and the caller controls exactly when the call happens. This determinism is precisely why REST remains the backbone for anything that needs to be tested, versioned, rate-limited, and reasoned about the same way every time - model-serving endpoints, feature store lookups, or triggering a governed job.

2.3 Event Bus

EVENT BUS

Decoupled, throughput-oriented, fan-out

An event bus (Kafka, Azure Event Hub, Service Bus, AWS Kinesis) inverts the relationship: producers publish facts about what happened, and any number of consumers subscribe independently, at their own pace, without the producer knowing or caring who's listening. This is the only one of the three that treats time and volume as first-class concerns - it's built for continuous, high-volume, many-to-many data movement rather than a single logical request.

The Decision Criteria That Actually Matter

Below is the framework we use when a client or internal team asks “why not just use REST for everything” (a fair question) or “should this be an MCP tool” (an increasingly common one).

	MCP	REST API	EVENT BUS
CONSUMPTION PATTERN	Variable - agent decides at runtime which tools to invoke and how many times; call count is not knowable in advance	Deterministic. One request, one response, known schema and cost every time	Continuous/streaming, volume driven by upstream event rate, not by a caller’s decision
SCALABILITY MODEL	Scales with reasoning complexity, not raw volume; bottleneck is usually the underlying tool APIs, not MCP itself	Scales horizontally via load balancers/API gateways; well-understood autoscaling patterns	Scales via partitions/consumer groups; natively built for very high throughput and back-pressure handling
EXTENSIBILITY	Very high. New tools are added and discovered without touching agent code; ideal for growing tool ecosystems	Moderate. New capability means a new endpoint/version; contract changes need coordinated rollout	High for new consumers (just subscribe), moderate for new event types (schema evolution needed)
COST PROFILE	Token cost of agent reasoning + per- tool-call cost; can spike unpredictably with poor prompt/tool design	Predictable, usage-metered (per-call or compute-time); easiest to forecast and budget	Infrastructure-heavy (brokers, storage, partitions) but very cheap per unit of data at scale
LATENCY	Highest; includes model reasoning time on top of tool execution	Low and consistent (sub-second to low-second typically)	Near-real-time to eventual (ms to minutes), by design asynchronous

DETERMINISM / AUDITABILITY			
Low by default. Requires explicit confidence gating, logging, and human-in-the-loop checkpoints for regulated use	High. Same input reliably produces same call pattern; easy to unit test and log	High for the data itself (durable, replayable log) but consumer- side processing logic must be independently audited
COUPLING			
Loose between agent and tools, but tight between agent and its own reasoning quality	Tight. Caller and service share an explicit contract/version	Very loose. Producer and consumer never need to know about each other
BEST FIT			
Agentic orchestration, dynamic knowledge retrieval, multi-step reasoning workflows, copilots	Governed transactional operations, model inference endpoints, system-of-record CRUD	CDC pipelines, telemetry, streaming ingestion, cross-domain fan-out, medallion architecture ingestion

RULE OF THUMB

If the exact sequence of calls is known before the request arrives, MCP may add value to that interaction. If workload volume is driven by events rather than caller requests, consider an event-driven interface. And if a regulated process requires a reproducible decision path, agent-directed execution needs explicit controls, logging, and validation.

Use-Case Scenarios: Which Interface, and Why

The table below maps common enterprise AI/data scenarios to the interface that fits best, and, importantly, where a hybrid is the honest answer. The following scenarios illustrate how these patterns apply across common enterprise AI and data workloads.

SCENARIO	PRIMARY	RATIONALE
Agent looking up a patient's protocol deviation history across three systems before drafting a query	MCP	Number and order of lookups depends on what the agent finds; a fixed REST call sequence can't anticipate every branch
Triggering a validated SDTM mapping job with a known set of parameters	REST API	Needs a versioned, testable, IQ/OQ/PQ-friendly contract with predictable inputs/outputs for audit
Streaming lab instrument readings into a Bronze layer for downstream curation	EVENT BUS	High-volume, continuous, many consumers (QC dashboards, ML features, archival) need the same events independently
Copilot answering "what changed in this dataset since last week" by choosing between schema diff, lineage, or CDC log tools	MCP	Tool selection genuinely depends on reasoning about the question, not a fixed workflow
Feature store lookup during real-time model inference	REST API	Millisecond-level, deterministic, needs strict SLAs - no room for agent reasoning overhead
Change-data-capture from an OLTP system feeding multiple downstream domains (finance, ops, ML)	EVENT BUS	Classic fan-out: one write, many independent readers, no coordination needed between them
Orchestrator routing ontology-tagged clinical text to the correct downstream mapping microservice based on a confidence score	MCP + REST API	MCP/agent layer decides which service to call based on confidence; the actual invocation of that service is a deterministic REST call underneath
Nightly batch reconciliation between two ledgers	REST API	Fixed schedule, fixed logic, no benefit from agentic flexibility or streaming infrastructure

Real-Life Execution Examples

Patterns are easy to describe in the abstract; they earn their keep in production. Below are three implementations from our data engineering and advanced analytics work, one per paradigm, followed by a hybrid case that is closer to what most mature platforms actually look like.

EXAMPLE 01 / MCP IN PRODUCTION

Agentic Data Engineering Orchestration

THE IMPLEMENTATION

A dual-orchestration architecture built on Databricks Mosaic AI combines n8n (workflow layer) with LangGraph (reasoning layer) to interpret ontology-tagged outputs (via SciBite) and dynamically route records to the correct containerized Python mapping service. The agent doesn't call a fixed endpoint; it evaluates a confidence score for each ontology match and decides, per record, whether to auto-route, escalate for a second confidence gate, or flag for human review.

WHY MCP-STYLE TOOL-CALLING FITS

The number and sequence of downstream calls is genuinely unknown ahead of time. It depends on data quality and ontology match confidence for that specific record. A fixed REST call sequence would either over-process clean records or under-handle ambiguous ones. Because this runs in a GxP context, every tool invocation is logged to a Delta Change Data Feed audit trail with IQ/OQ/PQ validation evidence, which is how the "variable consumption" of MCP is made auditable rather than opaque.

NOTE

Here is the optional usage of LangGraph for intent classification and routing to AI tools as needed.

EXAMPLE 02 / REST API IN PRODUCTION

Governed Job Triggering & Feature Serving

On the Azure side, a layered microservices architecture (AKS + API Management + FastAPI services on Container Apps/App Service) exposes deterministic endpoints for triggering data engineering incremental-load jobs and serving pre-computed features to downstream models. Analytical Platforms like Databricks and Snowflake connector services built with key-pair authentication expose the same predictable contract: known parameters in, known dataset out, versioned and rate-limited through APIM.

WHY REST FITS

Every call needs to be independently testable, has a fixed cost per invocation that finance can forecast, and must behave identically whether it's called by a scheduler, a human, or another service - no room for "the agent decided to do something slightly different this time." A key advantage of this architecture is that it can pair a user-facing interface with real-time observability and monitoring while retaining a modular microservices backend.

EXAMPLE 03 / EVENT BUS IN PRODUCTION

Multi-Stream Medallion Ingestion

A Kubernetes-native data platform on AWS EKS uses Kafka (via Strimzi) and Flink for stream processing, with Debezium handling change-data-capture from source OLTP systems, feeding a multi-stream Medallion architecture (Bronze → Silver → Gold). On Azure, the equivalent pattern uses Event Hub and Service Bus feeding the same layered ingestion model.

WHY AN EVENT BUS FITS

Dozens of independent consumers - quality dashboards, feature pipelines, archival jobs, ML training extracts - all need the same underlying events without coordinating with the producer or each other. Volume is driven by instrument and transaction activity, not by anyone "asking" for data, and the durable log means late-joining consumers (like a newly onboarded ML pipeline) can replay history without disrupting anything already running.

5.1 THE HYBRID REALITY: CONFIDENCE-GATED ROUTING

Agent decides. API executes. Event stream records.

The clinical oncology-routing example above is worth expanding, because it's the pattern we see recurring across advanced analytics builds: agentic orchestration decides, REST executes, an event bus records. The LangGraph reasoning layer applies a confidence gate (for example, routing only above a 0.85 threshold; anything below goes through a second n8n-level gate), then calls the winning containerized service over a conventional REST contract and finally emits a durable event describing what happened for downstream audit and analytics consumption.

Confidence-gated routing:
MCP decides → REST executes → Event Bus records

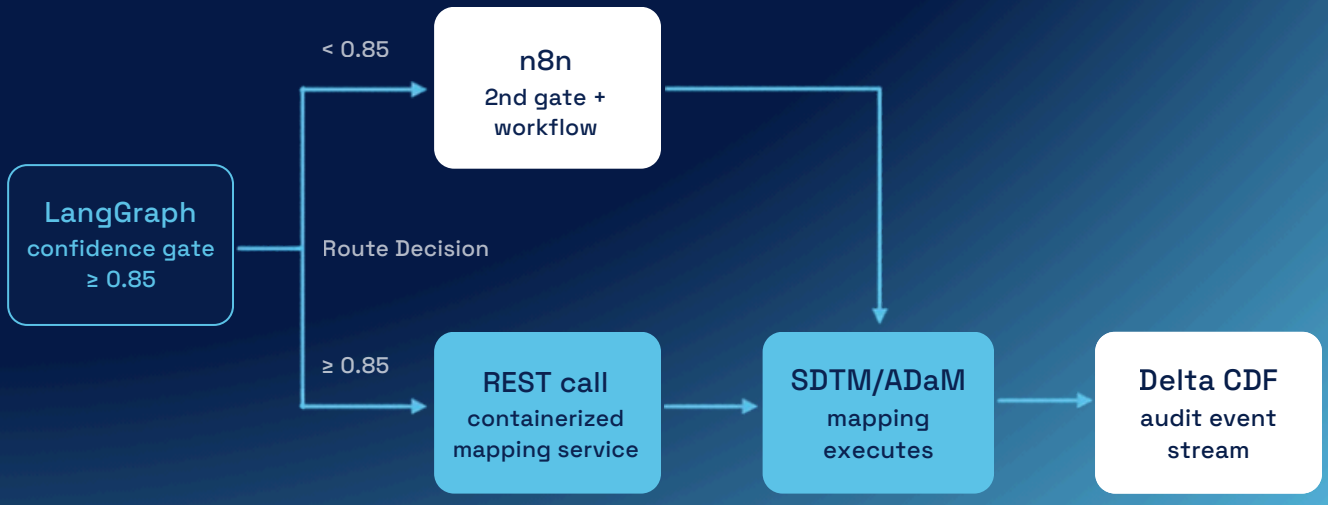
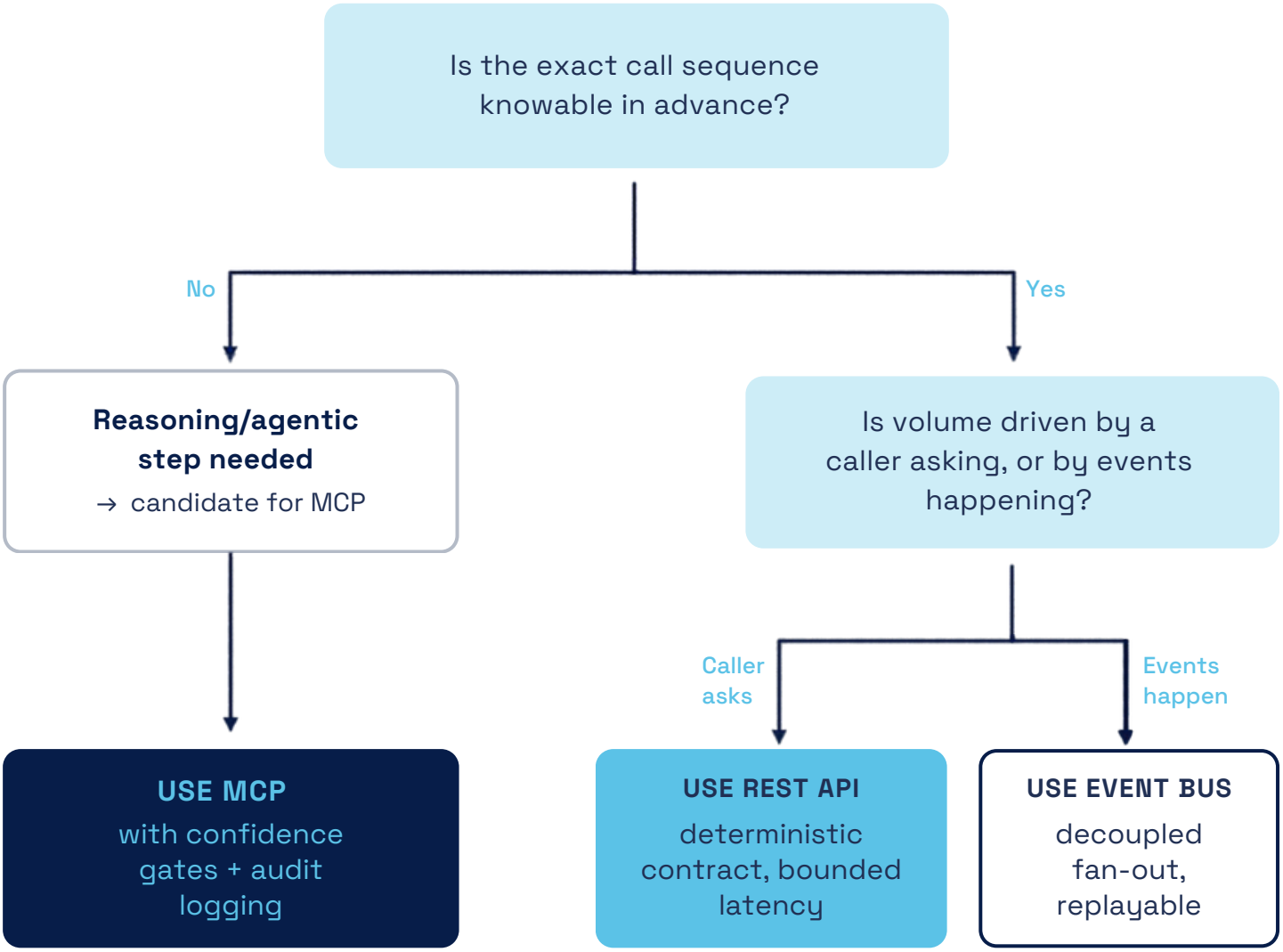


Figure 2 - A production confidence-gated routing pattern:

the agentic layer decides, REST executes deterministically, and every step lands in a durable, auditable event stream.

Deciding in Practice: A Simple Flow



Most production platforms need all three layers working together - the goal is drawing the boundary correctly, not picking a single winner.

Figure 3 - A practical decision tree:

start with “is the call sequence knowable,” then split remaining traffic by whether it’s caller-driven or event-driven.

Cost & Scale Cheat Sheet

	MCP	REST API	EVENT BUS
DOMINANT COST DRIVER	LLM token consumption + orchestration compute; scales with reasoning steps, not data volume	Compute-time or per-call metering; scales linearly and predictably with call count	Broker/cluster infrastructure (partitions, retention, brokers); largely volume-independent per-message once provisioned
COST PREDICTABILITY	Low - a single ambiguous input can trigger a materially longer reasoning chain	High - near-fixed cost per transaction, easy to budget and chargeback	Medium-high - infra cost is stable, but retention/replay policies can add unplanned storage cost
SCALE-OUT MECHANISM	Parallel agent sessions; bottlenecked by downstream tool capacity and model rate limits	Stateless horizontal scaling behind a gateway/load balancer	Partition/consumer-group scaling; near-linear throughput scaling with broker sizing
TYPICAL GOVERNANCE OVERHEAD	High - requires confidence thresholds, guardrails, and full call-chain logging for regulated use	Low-moderate - standard API versioning, contract testing, and rate limiting suffice	Moderate - schema registry and consumer-side lineage tracking needed for full auditability

NOTE

Actual latency, cost and scalability depend on platform, workload, implementation, deployment model and service configuration; the table reflects common architectural tendencies rather than guaranteed properties.

Closing Take

The mistake we see most often isn't picking the "wrong" interface outright; it's treating this as a single, ecosystem-wide choice instead of a per-interaction one.

> A platform that routes ontology-tagged clinical text through an agent, executes the actual mapping over REST, and streams the outcome onto an event bus for downstream analytics isn't being indecisive, it's matching each hop in the workflow to the property that actually matters at that hop: variable reasoning, deterministic execution, or decoupled fan-out.

The practical exercise worth running on your own architecture walk every integration point in your AI ecosystem and ask three questions:

01

Is the sequence of calls knowable in advance?

02

Does the caller need to wait for a specific answer?

03

Is the volume driven by a request or by an event?

The answers, not a framework preference, should decide the wire.

This framework draws on hands-on experience building regulated and GxP-compliant clinical data platforms, agentic data engineering solutions, Ai-powered applications, using Databricks, Snowflake, LangGraph, LangChain, Kafka, and Event Hubs-based services across pharmaceutical and life sciences environments.

ABOUT THE AUTHOR



Siddharth Mohanty

Director, Data Management

Fresh Gravity

Siddharth Mohanty is Director of Data Management with over 11 years of experience in building scalable data products and solutions across industry domains. As a certified Enterprise Architect and Solution Architect, Siddharth possesses extensive knowledge in delivering data projects like - Data Warehousing, Cloud Migration, Data Integration, Advanced Analytics and Machine Log Data Analytics solutions on cloud and on-premises. Siddharth has designed highly scalable and regulatory compliant solutions for clients at Fresh Gravity.

Connect on LinkedIn

ABOUT FRESH GRAVITY

Founded in 2015, Fresh Gravity helps businesses make data-driven decisions. We are driven by data and its potential as an asset to drive business growth and efficiency. Our consultants are passionate innovators who solve clients' business problems by applying best-in-class data and analytics solutions. To know more about us and our offerings, contact us at info@freshgravity.com.

U.S. CORPORATE HQ - DALLAS, TX

2201 Spinks Road, Suite 205
Flower Mound, TX, USA, 75022

OTHER OFFICES

U.S.A. Dallas
Canada Toronto
UK London
India Pune

www.freshgravity.com