Vamshi Jandhyala

Agent-ready financial data: an architectural view

✠

May 2026

An architectural view of how a financial data provider can make its data ready for autonomous and semi-autonomous agents. Workflow archetypes, the three human-in-the-loop bands, the surfaces (REST, SDK, MCP, `agents.md`, skills, code-as-action), and why the semantic layer is the product.

What "agent-ready" actually means

Agent-ready turns up a lot in vendor marketing now. Usually it means some combination of an API, an OpenAPI spec, and an MCP server shipped in the last quarter or two. Each of those is useful. None of them, by itself, gets you to the thing the phrase implies.

A more demanding working definition is something like this. A data product is agent-ready when an autonomous or semi-autonomous agent, written by someone who has never spoken to your customer success engineer, can reach correct answers to realistic customer questions without learning your house quirks. That implies a few things at the same time. The conventions a customer would otherwise have to absorb need to be discoverable. Defaults need to be right rather than expedient. The agent has to be able to find the right tool without coaching. When something goes wrong, the failure has to be legible enough that a human can step in. And the failure that matters most, the agent confidently producing a wrong number that a human downstream then publishes, has to be engineered out, not just hoped against.

That definition is the test the rest of the piece is measured against.

The shape of the work agents are being asked to do

Buyer segmentation (buy-side, sell-side, corporate, asset servicing) is the natural starting point for go-to-market. It is not a useful starting point for architecture. The architectural choices follow the shape of the workflow much more closely than they follow the buyer. The same hedge fund will run all five of the workflow archetypes below, and so will the same investment bank, and so will the same fund administrator. The buyer label tells you who pays. The workflow tells you what to build.

Five archetypes recur often enough to be worth naming.

- **Retrieval.** Look up a fact. *What was AAPL's adjusted close on 2010-06-09? What is the current credit rating on this CUSIP? Who is the lead manager on this*

issuance? Usually a single call, usually with sub-second tolerance, often safe to run autonomously, often the entry point for a longer chain of agent work.

- **Synthesis.** Compile a brief from several sources. *Produce a morning note on EU fixed income. Summarise overnight news affecting the portfolio. Build an IPO comparable pack.* Multi-call, tolerant of seconds rather than milliseconds, usually safe to run to a draft and then hand to a human for review.

- **Analytical.** Compute, often iteratively, often on results too large to fit in the agent's context. *Compute the realised volatility-adjusted return spread between these two portfolios over five years. Estimate the factor exposure of this fund using a rolling regression.* Multi-step, sometimes minutes long, sometimes carrying intermediate state. Correctness here is non-trivial. The gap between something that looks right and something that is right can be wide and silent.

- **Monitoring.** Watch and alert. *Tell me when this covenant ratio breaches. Flag when this issuer's CDS widens by more than two standard deviations. Notify on any rating action in this sector.* Long-running, scheduled, rules-driven, with the agent acting as both subscriber and triage layer.

- **Generative.** Produce a document or a piece of communication. *Draft the client report. Write the regulatory filing extract. Produce the fund factsheet.* Often the highest-stakes archetype, because the output gets published. Almost always benefits from a mandatory human checkpoint before it goes out.

The five differ in latency tolerance, autonomy, correctness threshold, and audit needs. The architecture has to follow.

Where the human stays

The right organising frame for human-in-the-loop is not the regulatory regime, even though regulation is what eventually codifies it. The frame is risk and reversibility. What does a wrong answer cost, and how easily can it be undone?

Three bands hold up across customer types.

Autonomous. The cost of a wrong answer is bounded and reversible. A wrong retrieval shows up as a confused user who asks again. A wrong overnight summary is replaced by tomorrow's. Most retrieval, most monitoring detection, and most exploratory analytics belong in this band. The right design move is to optimise for throughput and cost, accept a non-zero error rate, and instrument well enough to learn from the failures.

Reviewable. The cost is bounded but the answer is being acted on. A drafted client comm. An extracted figure that goes into a deck. A screen of names that will be traded later. The agent produces, a human reads, the human is the last line. The design question is not so much "how good is the agent" as "is review cheap enough that the agent earns its place in the workflow". Reviewability matters more than autonomy in this band. Citations have to be visible, intermediate steps have to be inspectable, and the diff against ground truth has to be easy to read.

Checkpoint. The cost of a wrong answer is unbounded or hard to reverse. Anything client-facing and published. Anything that moves money. Anything that touches

MNPI or order flow. Anything that changes a system of record. The agent stops and waits for an authoriser. The audit trail has to be complete enough to satisfy a model risk team six months later. The regulators that show up in this band (MAR, MiFID II best execution, SR-11-7 model risk, DORA operational resilience, the EU AI Act high-risk categorisation) are not the source of the constraint. They are the codified form of it. The checkpoint exists because the cost is real, and the regulations exist for the same reason.

The mapping back to the archetypes is approximate but useful. Retrieval is mostly autonomous. Synthesis is mostly reviewable. Analytical depends on what the answer drives: exploration sits in autonomous, position-sizing in checkpoint. Monitoring detection is autonomous, monitoring action is checkpoint. Generative output for clients is almost always checkpoint, internal generative drafting is reviewable.

The architectural consequence is that the surfaces have to support the bands, not just the archetypes. An agent operating in autonomous mode wants low-latency tools and minimal ceremony. An agent operating in checkpoint mode wants every call logged, every input attributable, every output reproducible. Both should be running on the same plumbing.

The surfaces, and what each is for

A financial data provider can ship through six surfaces. They compose rather than substitute. The right question for each is what it is good at, and what it is not.

REST/gRPC API plus a typed SDK is the substrate. The REST API is the contract. The SDK is the productised, idiomatic way to use the contract. The SDK is where the engineering investment goes: typed responses, retries, streaming, batch, async, observability hooks, and the composition helpers that bake in the right defaults. Most other surfaces end up being customers of the SDK rather than peers to it.

MCP is the reachability protocol. It exists to make the service callable from agents the provider does not host: Claude.ai, ChatGPT, IDE-embedded agents, hosted enterprise platforms. Customers running their own agents in their own infrastructure mostly do not need MCP, because they import the SDK directly. MCP earns its place when the agent host is not the provider's choice. Tools are the primary unit. Resources expose the catalogue and schema. Prompts encode the canonical patterns. The MCP server is most coherent when it is a thin facade over the SDK. Any capability that exists in the MCP server but not in the SDK creates a divergence between hosted-agent customers and self-hosted-agent customers, and that divergence eventually turns up as inconsistent behaviour.

`agents.md` (or whatever filename the platforms eventually settle on) is the static brief the model reads at the start of a session. It tells the agent the conventions. Dates default to trade date. Calendars default to the security's primary listing market. Prices are total-return-adjusted unless otherwise specified. Identifiers are resolved automatically across ISIN, FIGI, ticker. The brief is the cheapest, highest-leverage agent-readiness investment available. A page or two of well-written conventions does more for first-call accuracy than a hundred lines of tool description ever will.

Skills are named, replayable, multi-step workflows. They are the natural home for sequences that recur across customers: morning portfolio prep, NAV reconciliation, factor research bootstrap, IPO comp pack, covenant monitoring rule. A skill encodes the right sequence of tool calls and the right defaults for that sequence. An agent

invoking a skill inherits the correctness rather than re-deriving it. Skills also tend to be where domain semantics that are too workflow-specific to live in the SDK find their home.

Code-as-action is the high-bandwidth analytical surface. The agent, given a sandbox with the SDK pre-loaded as a module, writes Python that calls multiple tools, processes their results in memory, and returns the answer. For analytical workflows that would otherwise need thirty MCP round-trips and a context window stuffed with intermediate data, the saving is dramatic. Token cost reductions of an order of magnitude or more have been reported in the literature, and the figure broadly holds up in practice. Code-as-action earns its operational cost (sandbox lifecycle, security review, dependency curation) when the workflow is genuinely programmatic. For short, well-defined chains, the simpler tool-by-tool model is the better fit.

Reference agent clients, deliberately thin. Not a product, an on-ramp. A few hundred lines that connect to the MCP server (or import the SDK), run a loop, answer a question, return cited results. Their job is to dogfood the surfaces and to give a customer a five-minute path from "I have an API key" to "I have a working agent against this data".

The mapping to the archetypes follows the latency and bandwidth profile. Retrieval is well served by MCP tools or direct SDK calls. Synthesis is well served by skills calling MCP tools, with the skill encoding the source ordering. Analytical wants code-as-action over the SDK; the alternatives become too chatty. Monitoring wants scheduled agents calling the SDK directly, with results piped into whatever alerting the customer already runs. Generative wants skills with explicit checkpoint hand-offs encoded into the workflow.

A single canonical surface is rarely the right choice. An MCP-only offering is unreachable from any agent that prefers a typed library. An SDK-only offering is invisible to hosted agent platforms. A skills-only offering has no escape hatch when a customer's question does not fit a named workflow. The surfaces work as a layered offer.

The semantic layer is the product

This is the load-bearing claim of the piece, and it deserves a careful defence because it changes the rest of the architecture.

The raw data a financial provider stores is largely commoditised. Many vendors have prices, fundamentals, ratings, holdings, reference data. The differentiation, the thing customers actually pay for, is the layer of conventions and transformations that turns raw storage into correct answers to real questions. That layer is what I will keep calling the semantic layer.

For financial data the layer includes a lot of things at once. Point-in-time correctness, where Apple's revenue for Q2 2018 has two answers (the originally reported number and the post-restatement number) and the system has to let the caller ask for either. Corporate action handling, with the splits, spinoffs, mergers, and ticker changes that quietly invalidate any naïve price series. Calendar and holiday handling, of the kind that has to know LSE was closed for the Queen's funeral and NYSE for Hurricane Sandy. Currency normalisation, where a return in GBP and the same return in USD differ in non-obvious ways depending on FX rate timing. Identifier resolution across ISIN, FIGI, ticker, SEDOL, RIC, all of which have lifecycles. Entitlement

enforcement, with the awkward fact that customer A is licensed for ratings but not holdings. NULL semantics, in which missing-because-unreported is not the same state as missing-because-not-applicable.

None of this is the data. It is the correct use of the data. And it is where most of the engineering effort in a serious data vendor actually lives, even though it is invisible to customers who never have to think about it.

Two consequences for the architecture follow.

The first is that the semantic layer is invariant under transport. Whether a customer reaches the provider through REST, an SDK, MCP, code-as-action, or a notebook, the semantic layer has to be in the path. The surfaces are interchangeable. The semantics are not. This is why the MCP server is most coherent as a customer of the SDK, the SDK as a thin wrapper over the REST API, and every internal pipeline as a customer of the same semantic library underneath. Logic that exists in two places drifts. The drift surfaces as confidently wrong agent answers, and those are expensive in ways the customer remembers.

The second is that the semantic layer has to be made legible to agents, not just enforced. The cheapest channel for legibility is the schema. Rich types. Branded identifiers. Distinct types for `TradeDate` and `SettleDate`. Enums for `AdjustmentBasis`. The next cheapest is naming. `get_total_return_prices(asof=...)` invites correct use. `get_prices()` invites guessing. For non-human consumers, verbose and unambiguous wins over terse and clever almost every time. The most leveraged channel is the brief that the model reads at session start. *Dates default to trade date. Calendars default to the security's primary listing market. Prices are total-return-adjusted unless otherwise specified.* That paragraph does more for first-call accuracy than any amount of tool description.

The shorter version: the data is the substrate, the semantic layer is the product, the surfaces are how the product reaches customers. Get the layer right and the surfaces become almost interchangeable. Get the layer wrong and the surfaces will not save you.

Long-running and file-based realities

Most serious financial data is not a synchronous RPC. Bulk fundamental extracts, factor histories spanning decades, intraday tick datasets, regulatory filing snapshots, derived datasets produced by overnight pipelines, all arrive on the order of minutes rather than milliseconds, and often as files rather than JSON payloads. An architecture that pretends everything is sub-second tool-calling will hit a wall on its first real customer question.

The patterns that work for this are not subtle, and they are mostly about being honest about what is happening.

Model the lifecycle explicitly. A long-running operation has three states: submitted, running, complete. Expose them as such. A five-minute extraction modelled as a slow tool call produces timeouts, wasted context, and unreviewable agent traces. The honest shape is: submit returns a job ID, status returns progress, fetch returns a result handle. The agent orchestrates the steps, the host shows the progress, and the customer can see what is going on at any point.

Use result handles, not inline payloads. The agent's context window is the binding constraint, not the wire. A two-gigabyte parquet result has no business being

marshalled into the agent's conversation. The result lives server-side, keyed by handle. The agent calls slicing and aggregation tools that come back with kilobytes. *Top ten by volatility. Mean of column X for these securities. Filter to issuers in this jurisdiction.* Compute close to the data, ship summaries.

This is also where code-as-action earns its operational cost. The polling-and-slicing dance that takes thirty round-trips through MCP collapses into eight lines of Python in a sandbox. The agent submits the job, waits, slices the resulting dataframe, returns the number it actually cares about. One trip across the network surface, one well-typed answer. For analytical workflows on file-based data, code-as-action is what stops the architecture fighting itself.

Evaluating whether it works

Tool-use benchmarks are the easy starting point and a shallow signal. They measure whether the agent picks the right tool from the menu. They do not measure whether the menu is the right one, and they do not measure whether the answers are correct. Effectiveness for a data provider has to be measured against the questions a customer would actually ask.

A useful evaluation has roughly four parts.

A golden set of representative end-user questions, curated by domain experts, with ground-truth answers. Two hundred to five hundred questions, distributed across the five archetypes. For each question the eval captures whether the answer is correct (within tolerance), whether the citation is correct (the right dataset, the right point-in-time snapshot), how many turns the agent needed, and what the run cost.

Cross-model variance, which is procurement-grade signal. Run the same eval against Claude, GPT, Gemini, Mistral. High variance suggests the tool descriptions are accidentally tuned to one model's idioms. For a data provider selling into a heterogeneous client base, low variance is differentiating. It tells the customer that whichever model their stack runs, the answer quality stays consistent.

Adversarial probes. Ambiguous queries to test whether the agent asks back or hallucinates. Entitlement boundary tests, where the agent asks for data the customer is not licensed for and the server (not the agent) enforces the boundary. Prompt-injection probes inside data fields. Cost-runaway tests, because a misbehaving loop can burn real money fast.

Validation packs that look like model-risk documentation. The customer's MRM team will not approve an agent in production unless they can audit it. A validation pack with documented test methodology, reproducible runs, and characterised failure modes is what makes the offering procurement-ready. It is not a separate deliverable. It is the form your eval suite takes when you hand it to a buyer.

Two hundred questions of the right kind beats a thousand of the wrong kind. The expert curation cost is real and has to be budgeted. It is also, in practice, the single highest-leverage spend on the agent-readiness side. The eval suite is what turns vague claims about agent-readiness into something a buyer can verify.

What this means for the shape of the company

A few observations rather than prescriptions.

The agent-ready data provider has a smaller customer-facing surface than the API-era one. Fewer endpoints, more semantics. The pressure to expose every internal table as a public API gives way to pressure to encode every internal convention into the semantic layer. The catalog of public capabilities shrinks, and the depth of each capability grows.

The investment in DevRel for non-human consumers becomes real. Tool descriptions, `agents.md`, skill bundles, sample reference clients, eval suites, model-specific tuning. Work that used to be implicit in human DX (good docs, good IDE autocomplete) becomes an explicit deliverable for agent DX. The team that owns this is small, but it has to be senior. It sits between product, engineering, and customer success, and it has its own roadmap.

The cannibalisation question is real and worth stating openly. A provider whose terminal or feed business represents a large share of revenue will find that agent-ready surfaces erode seat-based pricing in directions that are hard to reverse. Delaying the agent surfaces does not solve that problem. The priced unit changes when the consumer is software. Tokens, queries, surfaced facts, validated answers, certified extracts. None of those map cleanly onto the legacy SKU. The provider that resolves the question earliest is the one that captures the agent-era spend rather than spending the next few years defending the terminal-era spend.

Org structure and pricing depend heavily on the provider's starting position, and they are out of scope here. Both deserve their own piece. The point worth landing is that the architectural choices above have organisational consequences that show up within twelve to eighteen months of shipping the first agent surfaces. They are worth planning for now, not after.

Closing

The interesting shift is that the data provider's product becomes the semantic layer rather than the data feed. The surfaces (REST, SDK, MCP, `agents.md`, skills, code-as-action) are how the product reaches customers. The semantics are why customers stay. A provider that ships every surface but neglects the semantic layer is a weaker competitor than one that ships only the SDK with the conventions right. Exquisite surfaces over a thin semantic core is a recurring shape. Surfaces over a strong semantic core is the rarer and more durable one.

Most of the work over the next two years is invisible from the outside. It looks like cleaning up calendars, codifying corporate-action conventions, building a bitemporal engine, naming methods unambiguously, writing the brief that the model reads. None of it is glamorous. All of it is what separates an agent that occasionally produces the right number from one that consistently does. The data that ends up in the agent answers customers trust will be the data whose providers chose to do the unglamorous work first.