MCP in Production

What your AI architecture needs and probably doesn’t have yet

IA NFO Systems

2026

Contenido

MCP in Production

What your AI architecture needs and probably doesn’t have yet

IA NFO Systems · ia.nfo.systems

> This document is written for the technology director or CTO who already has AI agents in some state — pilot, partial production, or active evaluation — and is facing the same underlying problem: the integrations between your agents and your internal systems are handcrafted, fragile, and don’t scale. This is not an introductory document on artificial intelligence. It is a technical analysis of MCP: what it solves, what it doesn’t solve, and how to evaluate whether it’s worth implementing in your specific context.

Executive Summary

If your team has spent more than three months trying to get an AI agent to reliably use internal data in production, the problem already has a name: integration architecture. Every agent hand-connected to every system produces a dependency matrix that grows quadratically, breaks with every update, and has no centralized diagnostic point when it fails. MCP solves this with an architectural shift: instead of N×M point-to-point integrations, each system publishes its capabilities once and any compatible agent consumes them.

This document does not assume MCP is the right answer for your context. It assumes you need the technical criteria to decide.

In this document you will find:

- The precise diagnosis of why handcrafted integrations don’t scale — with the complexity formula that demonstrates it
- The MCP specification in concrete terms: what it defines, what it transports, what it leaves out
- What MCP doesn’t solve — the section most MCP documents omit
- Three implementation patterns on existing infrastructure, including the case of legacy systems with no API
- Real security implications: authentication, granular authorization, and controlling what the model sees
- A usability test to know whether your MCP server is production-ready before declaring it so
- The criteria for choosing the first server that produces evidence in under four weeks

---

The problem your team already knows

If you’ve spent more than three months trying to get an AI agent to reliably use internal data in production, your team has already described some version of this scenario:

The agent works in the demo. In the demo, the context is prepared, the data is correct, and the response is convincing. In production, the agent has no access to real systems, or has partial access built by hand for that specific case, or has access that breaks every time a source system changes something.

The result is predictable: the agent lives in a permanent pilot. Or it lives in production with a scope so narrow it doesn't justify the effort of maintaining it.

---

1. Why handcrafted integrations don't scale

The way most teams connect AI agents to internal systems produces a complexity structure that grows quadratically.

If you have N systems and M agents, each agent needs its own integration with each relevant system. In the best case, you reuse code between agents. In practice, each combination has its own quirks: the support agent needs read-only access to the CRM filtered by customer; the sales agent needs read-only access to the CRM filtered by open opportunity plus access to the price catalog; the executive agent needs both plus operation logs. Three agents, three distinct integrations to the same CRM.

Multiply that by your number of systems.

Each handcrafted integration has its own authentication scheme, its own error handling, its own logic for serializing context, and its own maintenance debt. When the CRM updates its API — which will happen, not if but when — something breaks. The team finds out when the agent produces an incorrect response or when a user reports the problem.

The cost is not just initial development time. It is the accumulated operational burden of maintaining an integration graph that grows with every new agent and with every update to every source system.

MCP solves this problem with an architectural shift: instead of N×M point-to-point integrations, each system publishes its capabilities once via an MCP server, and any compatible agent consumes them without additional integration.

---

2. What MCP is — the specification

MCP (Model Context Protocol) is an open protocol, published by Anthropic in November 2024, that standardizes communication between AI clients (the agents or the hosts that run them) and servers that expose capabilities — data, tools, or reusable prompts.

The specification defines three types of primitives an MCP server can expose:

Resources — data the client can read. A resource has a URI, a MIME type, and content. It can be static (a policy document) or dynamic (the current state of a production order). The client requests the resource; the server delivers it. The model receives the content as context.

Tools — actions the client can execute. A tool has a name, a parameter schema in JSON Schema, and produces a result. The model decides when to invoke a tool and with what arguments; the server executes the action and returns the result. The tool can be read-only (query a customer's status) or have side effects (create a ticket, update a record).

Prompts — prompt templates the server exposes for the client to use. Less common in initial implementations, but useful for standardizing complex instructions that depend on system context.

The transport is JSON-RPC 2.0. The protocol supports two transport modes: stdio (for local processes) and HTTP with Server-Sent Events (for remote servers). Authentication is not defined by the base protocol — it is implemented at the transport layer, typically with OAuth 2.0 or API tokens over HTTPS.

The full specification is at modelcontextprotocol.io. What matters for architecture decisions is understanding what is inside the protocol and what is outside it.

---

3. What MCP solves — and what it doesn't

This is the section most MCP documents omit. Worth reading before committing to an implementation.

What MCP solves well

Standardization of the integration interface. If you build your MCP server correctly, any compatible client can consume your capabilities. Today that includes Claude, Anthropic's developer tooling ecosystem, and a growing number of third-party platforms. The integration surface goes from N×M to N+M.

Separation of concerns. The MCP server knows how to access data and what operations are valid. The agent knows when and why it needs that data. That separation makes both sides easier to maintain and reason about independently.

Context portability. An MCP server built today works with any compatible client in the future. If you change the language model you use, or the agent framework, or the AI provider, your MCP integrations don't need to be redone. The open standard is precisely the opposite of lock-in.

Audit surface. Every interaction between an agent and your systems passes through the MCP server. That creates a single point to record what the agent queried, when, with what parameters, and what the system returned. In environments with compliance requirements, that is significant.

What MCP doesn't solve

It doesn't solve data quality. If your data is poorly structured, duplicated, or stale, the MCP server will deliver it poorly structured, duplicated, or stale. MCP is a channel, not a data quality transformer.

It doesn’t solve the logic of what context the agent needs. Deciding when to invoke which tool, with what parameters, and how to use the result remains the responsibility of agent design. MCP makes capabilities available; it doesn’t determine how to use them.

It doesn’t prescribe granular authentication or authorization. The base protocol has no built-in permissions model. You implement access control in the server. That gives flexibility, but also means security is your responsibility, not a protocol guarantee.

It doesn’t solve legacy system latency. If your ERP takes four seconds to respond to a query, the agent will wait four seconds. MCP doesn’t add caching or query optimization on its own — that is server design.

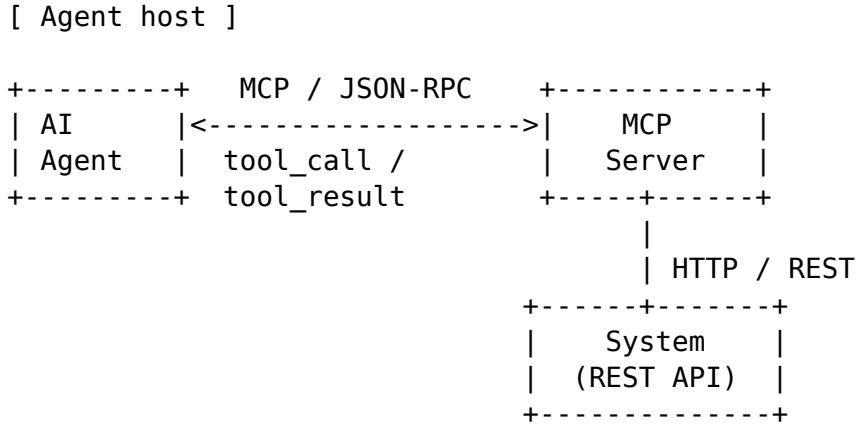
The specification is still maturing. MCP 1.0 was published in 2024. There are areas — particularly around the authorization model and streaming capabilities — that will continue to evolve. It is not an unstable specification, but it also doesn’t have ten years of history behind it like REST does. Worth designing your server with that in mind.

---

4. Implementation on existing infrastructure — three patterns

Pattern 1 — System with documented REST API

When it applies: The source system has a REST API with reasonable documentation and standard authentication (API key, OAuth, JWT). This is the cleanest starting point.



Architecture: The MCP server acts as a thin translation layer between the MCP protocol and the existing API. It receives a tool call from the agent, translates it to the corresponding HTTP call, receives the response, serializes it into the format the agent expects, and returns it.

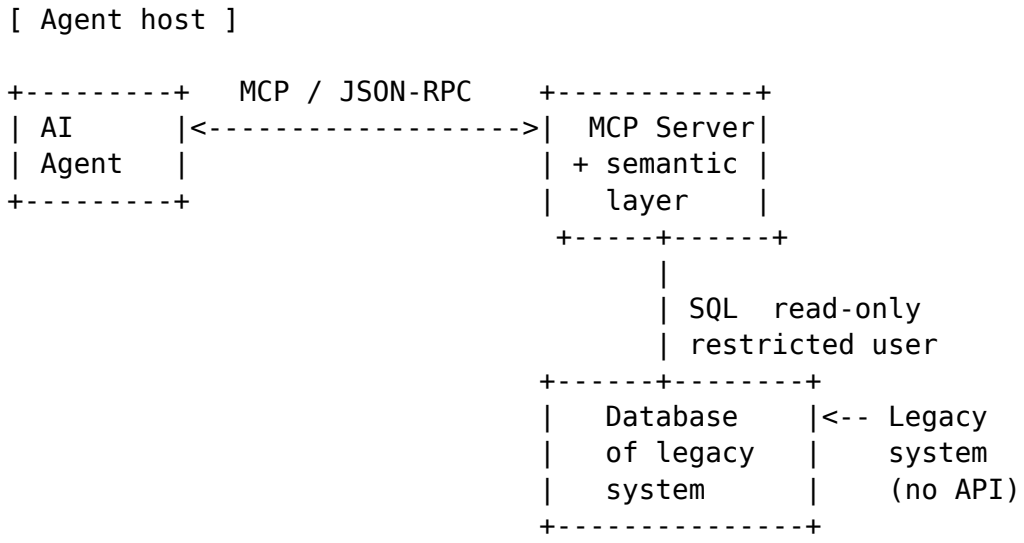
What it implies: An MCP server in this pattern is relatively simple — in most cases, fewer than 500 lines of code in the language of your choice (there are official SDKs for Python, TypeScript, and Java). The complexity is in deciding which subset of the API to expose and how to model the tools so the agent uses them correctly.

Main risk: Tool granularity. If you expose too much as a single tool (e.g., “query the

CRM"), the agent will have difficulty using it well. If the tools are too fine-grained (e.g., one tool per endpoint), the server becomes hard to maintain. The right balance depends on how the agent reasons about available capabilities.

Pattern 2 — Legacy system with no API

When it applies: The source system has no API. It has an accessible relational database, a structured file system, or a service with a proprietary protocol. This is the most frequent case in organizations with more than ten years of technology history.



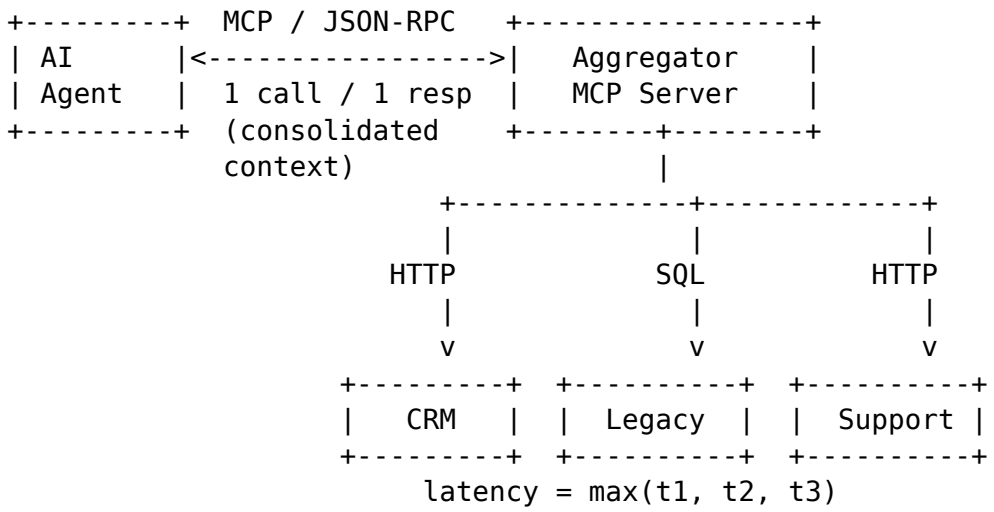
Architecture: The MCP server accesses the data source directly — typically the database — with a controlled read-only layer. The legacy system is not modified. A database user is created with read-only permissions on the relevant tables, and the MCP server exposes curated views of that data as resources or as tool results.

What it implies: Greater responsibility in the server. The server can't rely on the legacy system's API to validate its parameters — it has to do that itself. The SQL queries the server executes need to be correctly parameterized to avoid injection. The database schema may carry decades of historical decisions that need to be interpreted before exposing them.

Main risk: Legacy data schemas rarely match the agent's mental model. A table with two-letter column names and integer-encoded values requires a semantic translation layer that is non-trivial. Plan time for that interpretation before assuming the server will be simple.

Pattern 3 — Multiple heterogeneous systems

When it applies: The agent needs context from several systems simultaneously, and consolidating that context in the server is more efficient than letting the agent make multiple independent calls.



Architecture: An aggregator MCP server that exposes composite tools. A tool like `get_customer_context` can internally query the CRM, the billing system, and the support logs, and return a consolidated object. The agent makes one call; the server makes three.

What it implies: The server adds operational complexity — it now has multiple dependencies, and if any of the source systems is down, the server has to decide what to do: fail completely, return partial context with an indication of what's missing, or serve from cache. That resilience logic has to be designed explicitly.

Main risk: The server adds latency. If the three systems it queries respond in parallel in 800ms each, the result arrives in 800ms. If they respond in series, in 2.4 seconds. The server design has to force parallelism wherever possible.

---

5. Security and access control

The security questions that matter in MCP are no different from those that matter in any system that exposes internal data. What changes is who is asking the questions and from where.

Client authentication

The MCP server needs to know that the client talking to it is authorized. For remote servers (HTTP + SSE), the current specification recommendation is OAuth 2.0. For local processes (stdio), authentication happens at the process level — the server trusts the process that invokes it.

In enterprise environments, the MCP client is typically the agent host (the framework that runs the agent), not the model directly. Authentication is configured between the host and the server.

Granular authorization per tool

The server controls which tools it exposes and with what restrictions. There is no permissions model built into the protocol — you implement it. Common options:

- **Per tool:** some clients can only invoke a subset of tools
- **Per parameter:** the tool accepts any client, but parameter values are validated against the client's profile (e.g., an agent can only query data from customers assigned to its region)
- **Per volume:** invocation rate limits to prevent abusive use

What the model sees

This is the point that most concerns security teams, and rightly so. The content the MCP server returns — the result of a tool, the text of a resource — enters the model's context. If that content includes sensitive information, the model has it available when generating its response.

The mitigation is server design, not protocol design. The server decides which fields to include in the response. If a customer query returns financial data, the server can return only the fields relevant to the agent's use case — not the complete object.

A useful operational rule: design each tool as if its response could appear in a public log. If that makes you uncomfortable, the tool is exposing more than it should.

Auditing

The MCP server is the natural audit point. Every tool invocation can be logged: timestamp, client, tool, parameters, duration, result (or result hash if the content is sensitive). In regulated environments, that log is the evidence of what the agent queried and when.

---

6. Real adoption challenges

"MCP is an Anthropic initiative — if they pivot, we lose the investment"

MCP was published as an open specification under the MIT license. Governance is designed to be provider-independent. Most relevant: adoption already extends beyond Anthropic. OpenAI announced MCP support in its APIs. Microsoft Copilot has MCP integration. AWS, Google Cloud, and Cloudflare have their own implementations. The developer tooling ecosystem — Cursor, Zed, Continue — has native MCP support.

A protocol with multi-vendor adoption is not a bet on a provider. It is a bet on a standard.

"It adds another layer that can fail"

Correct. An MCP server is an additional process with its own dependencies, its own lifecycle, and its own failure surface. That is real.

The relevant comparison is not “MCP vs. nothing.” It is “MCP vs. the handcrafted integrations you already have.” A handcrafted integration per agent can also fail — and when it does, the failure is opaque, with no centralized audit point, and the fix requires understanding the specific code of that integration. An MCP server centralizes failures at an instrumentable point.

“Our systems don’t have APIs”

Pattern 2 from the previous section applies directly. Most legacy systems have an accessible relational database even if they don’t expose an API. What that path implies in practice: create a database user with read-only permissions on the relevant tables, build the MCP server with parameterized queries, and allocate real time for semantic translation — data schemas with decades of history rarely speak the same language as a modern agent.

That semantic translation work is where effort is consistently underestimated. It is not technical complexity: it is domain understanding. A column called `CUST_ST` with values 1, 2, 3 needs a translator who knows it means “active,” “inactive,” and “at risk” before the agent can do anything useful with it. Plan that time explicitly.

The result, well executed, is an MCP server more secure than any handcrafted integration accessing the same data without a centralized control point — because the server defines exactly what is exposed, with what granularity, and with full audit of every query.

“How do we prevent the model from seeing data it shouldn’t?”

The server controls exactly what it returns. The correct design exposes purpose-specific tools, not generic system access. “Get purchase history for customer X” is a tool with a controlled response. “Query the database with arbitrary SQL” is not a tool — it is a vulnerability.

“What happens when the specification changes?”

MCP has semantic versioning and breaking changes are announced in advance. The correct defensive design is to version your server explicitly and update when the client ecosystem you use requires it — not on every release. Backward compatibility has been a declared priority of the maintenance team since initial publication, and the specification’s current state of maturity was addressed in the previous section.

---

7. Where to start — the first MCP server worth building

The first MCP server should not be the most ambitious. It should be the one that produces evidence fastest.

Criteria for choosing the entry point:

High query frequency. If the agent will query that system many times per day, the value of the integration accumulates quickly and performance problems surface soon — when they are still cheap to fix.

Well-structured data. The first server is not the time to resolve semantic ambiguity in legacy data. Choose a system whose data structure is clear, with comprehensible field names and consistent types.

Low security risk. Start with data that is not sensitive. A product catalog, an internal knowledge base, an operations status log. Prove the architecture works before connecting financial or customer data.

Measurable result. What changes if this agent has access to this system? If you can't answer that with a number or an observable behavior, the use case is not sufficiently defined.

The usability test most teams skip

Before declaring the server production-ready, run this test: take a model that has never seen the system before. Give it access to the tools with no explanation in the system prompt about what they do or when to use them. Run real scenarios. Observe whether the model invokes them correctly, with the right parameters, at the right moment.

If the model doesn't use them well without explicit instructions, the problem is rarely the model. It is in how the tools are named and described. A successful MCP server is not measured by whether the agent can invoke its tools — it is measured by whether it invokes them correctly without guidance.

That test is the difference between a server that works in demos and one that works in production.

---

8. Next steps — for those who want to move faster than their team can alone

If your team already has the problem clearly defined but implementation is blocked — by time, by the complexity of source systems, or by the learning curve of the specification — there is a technical conversation worth having.

It is not a product presentation. It is a thirty-minute architecture review where we work with your infrastructure data and leave with three concrete things:

1. Which of your current systems has the highest context value for an agent — with the technical argument that justifies it
2. Which implementation pattern applies and what complexity it actually implies in your specific stack
3. The scope of the first server: what it exposes, what it doesn't expose, and why it can be in production in four weeks or less

The deliverable from the session is that scope — documented, with defined success criteria, ready for your team to evaluate or execute.

To schedule:

- → contacto@nfo.systems
- → ia.nfo.systems

About IA NFO Systems

We have spent thirteen years integrating production systems — ERPs, CRMs, legacy databases, proprietary APIs, industry protocols. When MCP arrived, we recognized the pattern: it is the same integration problem we have been solving by hand for a decade, now with a standard that makes it sustainable.

We implement MCP servers on existing infrastructure. We know Pattern 2 — legacy systems with no API — not as a theoretical case but as frequent work. We know how long it takes to semantically translate a data schema with twenty years of historical decisions, and when it is better not to.

If the problem you describe is not something we can solve well, we say so before we start.

> **For your organization's CEO or general manager:** there is an executive version of this document that explains MCP without technical jargon, with three business impact scenarios and answers to the five most common objections. Available at ia.nfo.systems or by request at contacto@nfo.systems.

---

ia.nfo.systems · México
contacto@nfo.systems

Servicios provistos por NFO Systems SAPI de CV.