THE MISSING LAYER

Rich Domain MCP Servers

Why AI on Enterprise Data Fails — and a Pattern That Could Fix It

---

> *Only about 5% of enterprise AI pilots achieve rapid revenue acceleration, according to MIT's NANDA initiative — the vast majority stall. Not because AI doesn't work. Not because the data is bad. But because enterprise implementations fail to adapt AI to their specific context — rarely does anyone tell the AI what the data means at the point where it reasons.*

David Golverdingen · AI Engineering & Technical Leadership · April 2026

MCP Architecture — Practitioner Report, v7

Rich Domain MCP Servers — The Missing Layer

Why AI on Enterprise Data Fails — and a Pattern That Could Fix It

David Golverdingen · AI Engineering & Technical Leadership · MCP Architecture · April 2026

> *Only about 5% of enterprise AI pilots achieve rapid revenue acceleration, according to MIT's NANDA initiative—the vast majority stall. Not because AI doesn't work. Not because the data is bad. But because enterprise implementations fail to adapt AI to their specific context—rarely does anyone tell the AI what the data **means** at the point where it reasons.*

Built on production implementations across seven MCP servers and 52 tools connecting five backend systems — a mid-market ERP system (11 tools), a BIM data explorer (Autodesk AEC Data Model, 7 tools), a fleet management platform (12 tools), an energy monitoring platform — a mixed server aggregating a smart meter data provider, a building automation platform, a meteorological API, and two government building registries into a single domain interface (10 tools) — and a construction standards database (4 tools), plus a visualization server (5 tools) and a games server (3 tools). Each server includes a dedicated feedback tool (included in the counts above). Rich domain metadata, a three-layer feedback architecture, server-rendered interactive MCP Apps, and a validated secure write pattern. Serving a Dutch installation company with 300–400 employees. The specific systems in this implementation are: AFAS Profit (ERP), Energiepartners and Priva Cloud (energy monitoring), Trips in the Cloud (fleet management), Autodesk AEC Data Model (BIM), Ketenstandaard (construction standards), and the Dutch Kadaster BAG and EP-Online registries (building data).

Abstract

Enterprise AI often fails not because models lack capability, but because the domain knowledge rarely reaches the AI at the point where it reasons — the tool interface. This paper introduces **Introspective Context Engineering**—a five-phase pattern where AI examines a data source, discovers domain knowledge through actual data exploration, writes that knowledge back as operational metadata in MCP tool descriptions and schemas, and continuously improves through a runtime feedback architecture. The result is a distinct category of AI-data integration: the **Rich MCP Domain Server**. Unlike RAG pipelines, Text-to-SQL engines, vendor-built copilots, or client-side skill files, this approach places domain intelligence directly in the tool interface where the LLM reasons over it at call time. The paper proposes a six-level maturity model for MCP servers — from bare API mappers through self-teaching domain servers to interactive MCP Apps with secure write operations — and presents six concrete recommendations for the MCP framework specification. The pattern has been applied in production across seven MCP servers totaling 52 tools, connecting nine external APIs across five domains (ERP, BIM, fleet management, energy monitoring, and construction standards) and serving a 300–400 person Dutch installation company. The first server took one developer roughly 40 hours with no prior MCP experience; subsequent servers took 1–2 days by reusing the established patterns.

Scope and evidence. This is a practitioner report, not an academic study. The pattern, claims, and recommendations are grounded in production experience across seven servers, nine external APIs, and one codebase — not in controlled experiments or broad benchmark studies. Where the paper states something as fact, it is observed behavior from these

implementations. Where it generalizes, it is inference from that experience. Where it recommends, it is the author's professional judgment. Throughout this paper, examples are drawn from the building data and ERP implementations to illustrate the pattern in concrete detail. The production codebase is referenced throughout so readers can trace claims back to actual implementation decisions.

Contents

---

01 The Context Gap

Enterprise AI investment tripled in a single year to $37 billion in 2025 [18]. The results don't match the spend. MIT's NANDA initiative analyzed 300+ deployments and found that only about 5% achieve rapid revenue acceleration [1]. S&P Global reports that 42% of companies abandoned most of their AI initiatives in 2025—up from 17% the year before [2]. RAND concludes that over 80% of AI projects fail, twice the failure rate of non-AI IT projects [3].

The industry blames data quality. Informatica names it the number one obstacle at 43% [4]. Gartner predicts 50%+ of generative AI projects stall in the pilot phase for the same reason [5]. But MIT identifies something deeper: not a data problem, but an adaptation failure — generic AI tools excel for individuals, but stall in enterprise use because they don't learn from or adapt to specific workflows [1]. Companies build the highway but forget the navigation system.

Clean data is not the same as understood data. Consider a service management ERP. A maintenance record has an "assigned technician" field. The AI reads it, finds it empty, and tells the user no technician was assigned. In reality, that field is never populated — the actual technician can only be found through time-booking records filtered by a specific work-order type. The data is perfectly clean. The AI simply doesn't have the interpretation layer to know that this field is a dead end. This is a data interpretation problem, not a data quality problem.

The market response follows a consistent pattern: solve the transport problem. RAG embeds documents in vector stores — useful for policy questions, useless for live operational queries [6]. Text-to-SQL translates natural language to syntax, but doesn't know that a 'type' field needs a specific numeric code or that certain schema fields are dead ends [7]. Vendor-built AI (SAP Joule, Microsoft Copilot for Dynamics 365, Oracle NetSuite AI) has intimate platform knowledge but mostly serves standard workflows — not free natural language access across domain-specific data models [8]. Enterprise data integration platforms (Informatica, K2view) connect the pipes but don't understand what flows through them [9]. In all cases, knowledge lives in infrastructure layers — vector stores, prompt templates, governance dashboards — rather than in the tool interface where the LLM actually reasons at call time.

Meanwhile, the Model Context Protocol (MCP) — adopted by Anthropic, OpenAI, Google, and Microsoft — is becoming the universal standard for connecting AI to tools and data [10]. With over 97 million monthly SDK downloads and 10,000+ active servers as of December 2025 [10], the ecosystem is growing rapidly. But transport alone is not enough.

The author's analysis of 50+ publicly available MCP servers reveals a consistent maturity distribution:

Level	Characteristics	Examples
1. API Mapper (~70%)	One-sentence descriptions. No domain context. AI must figure everything out alone.	Postgres, Filesystem, most community servers, generic ERP connectors, data platform wrappers
2. Functional (~20%)	Grouped tools, better organization. Still no domain knowledge or cross-references.	GitHub Official, Salesforce DX, Microsoft Dynamics 365 ERP MCP

Level	Characteristics	Examples
3. Metadata-Rich (~8%)	Knowledge graphs, glossaries, lineage. Metadata typically curated manually by dedicated teams over weeks or months.	OpenMetadata, DataHub, Actian, Oracle Analytics Cloud
4. Self-Teaching (<2%)	AI-discovered domain knowledge, cross-tool references, query strategies, fallback routes, uncertainty markers. Metadata generated and maintained by AI with human validation.	Introspective Context Engineering (this paper). Possibly emerging in internal enterprise tools not publicly visible.
5. Interactive App (emerging)	Server-rendered interactive UI components (charts, tables, maps, forms) returned directly in the conversation [26]. The agent coordinates; the server controls presentation. Data tools + visualization tools in the same ecosystem.	Production implementation in this paper: render_chart, render_table, render_map, start_duurzaam (energy analysis intake).
6. Secure Write App (frontier)	Graduated write operations: agent-initiated bounded writes (feedback, scores) through standard tools for low-risk mutations; user-initiated secure writes through validated MCP App interactions for business-critical data. WriteIntent pattern: agent opens the app, user drives the mutation, server enforces every boundary (one-shot, user-bound, time-limited, typed schema).	Production: report_problem (feedback to Firestore) and submit_tetris_score (leaderboard) deployed. Designed: ERP data corrections through MCP App forms with `visibility: ["app"]` commit tools.

The progression across levels follows a clear arc: expose data (Level 1) → organize tools (Level 2) → understand the domain (Level 3) → learn from data and feedback (Level 4) → present through interactive apps (Level 5) → act through secure writes (Level 6).

Even the MCP specification's own best-practice proposal (SEP-1382, currently dormant) considers a good tool description: "Read the contents of multiple files simultaneously. More efficient than reading files individually." This is the ceiling the community is aiming for [11].

The difference is not subtle. A Level 1 server says "query data from the ERP." A Level 4 server says "always start with a summary call — active projects have thousands of records.

Type codes determine which fields are populated. Use the budget tool for planned costs, this tool for actuals. Report friction via the feedback tool." Not the same product. Not the same category.

02 MCP Isn't Dead. It's Empty.

A growing discourse argues that MCP is dead — that CLIs and skill files make the protocol unnecessary [19][20]. The frustration is real, and CLIs are genuinely simpler for developers already living in a terminal. The counterarguments are equally real: MCP, skills, and CLIs solve different problems [21][22]. Both sides are right about transport — and arguing about the wrong layer.

The CLI/skills argument breaks down for enterprise data. A product manager asking about project margins won't `pip install` a CLI; a service coordinator won't write a `CLAUDE.md`. These users need a one-click OAuth and tools that just appear — exactly what MCP provides. And even with flawless transport, an estimated 70% of servers would still produce unreliable answers because they contain little to no domain knowledge. Wrapping the same hollow APIs in a CLI doesn't fix that.

What CLIs and Skills Cannot Replace

Four things only the server can provide: **server-side access control** (per-tool scoping verified against JWT claims, instead of whatever access the binary has); **structured tool metadata** (rich descriptions, typed input schemas, per-field output annotations — parsed by the model at tool-selection time, not stale `--help` output); **runtime feedback loops** (`report_problem`, `queryIntent`, session-correlated logs — reinventing these in a CLI is reinventing MCP, badly); and **enterprise governance** (audit trails, centralized auth, session correlation).

Skills: A Client-Side Patch for a Server-Side Problem

Anthropic's skills guide [25] diagnoses the problem correctly — users connect an MCP server but don't know what to do next — and prescribes a client-side "knowledge layer" of markdown recipes. The diagnosis is right; the prescription treats the symptom. Skills exist because MCP servers are Level 1: if the server already taught the agent what the data means, when to use which tool, and how to chain calls, there would be nothing left for the skill to teach. The kitchen has no recipes, so they hand out cookbooks at the door.

Distributing knowledge client-side reintroduces the problems skills claim to solve: it goes stale when the server updates, fragments across clients and versions, and only helps the one client that loaded it — a skill uploaded to Claude.ai doesn't help the same server when connected to Cursor, Copilot, or a custom agent. The natural home for domain expertise about a service is inside that service. When the server owns the knowledge, every connected agent benefits automatically.

MCP isn't dead. It's empty. The protocol works; the servers are hollow. The fix isn't a different transport or a client-side knowledge layer — it's filling existing servers with domain intelligence.

03 The Missing Layer

The transport problem is solved. What remains is the knowledge layer: metadata that tells the agent how the domain works — what the codes mean, which fields are reliable and which aren't, how to cross-reference between tables when the obvious path doesn't work, what patterns indicate problems versus normal operations.

Why is it rarely built?

Four reasons.

Developer mindset. Most MCP servers are built by developers wrapping APIs. Their mental model: "I make the API available, the LLM is smart enough to figure out the rest." For well-known domains—GitHub, Slack, file systems—this works. The LLM knows what a pull request is from its training data. But mid-market ERP systems with industry-specific configurations? Equipment type codes, maintenance scheduling rules, subcontractor assignment patterns? None of that is in any model's training data.

No complex domain. Most MCP servers don't have this problem because their domain is universally understood. A GitHub MCP server doesn't need to explain what a commit is. An ERP MCP server does need to explain what a service ticket (type 10) means versus a maintenance order (type 13) versus a service planning (type 48), and how they relate hierarchically.

No feedback loop. The build-vs-buy analysis for MCP servers focuses almost entirely on transport capabilities [17]. Most MCP server builders ship to a package registry, publish, and move to the next project. They don't have five power users hitting limits daily, revealing which metadata is missing and which interpretations lead to wrong answers. Without that feedback, you don't know what's missing.

New technology, unknown possibilities. MCP was introduced by Anthropic in late 2024 and is still maturing as a protocol. Most developers and organizations are still learning what MCP is, let alone what it can do beyond basic API wrapping. When a technology is this young, exploration tends to focus on getting it to work at all—not on pushing the boundaries of what metadata it can carry. The idea that an MCP tool description could contain multi-paragraph domain knowledge, query strategies, and cross-tool references simply hasn't occurred to most builders yet. The ecosystem needs time to discover its own possibilities.

The idea to enrich metadata is logical. But logical and obvious are different things. It is also logical to write unit tests and maintain documentation—not everyone does. McKinsey's 2025 AI Survey found that AI high performers are nearly three times more likely to have fundamentally redesigned workflows before selecting AI techniques [16]. The innovation here is not in the insight. It is in the degree of enrichment, the method (AI discovers domain and writes back), and the feedback mechanism to keep improving. The difference is in execution and systematization.

04 The Pattern: Introspective Context Engineering

Introspective Context Engineering is a five-phase pattern where an AI system examines its own data sources, discovers domain knowledge, writes that knowledge back as operational metadata, and continuously improves through a runtime feedback architecture.

The following sections detail each phase with concrete implementation patterns drawn from a production deployment. Where previous descriptions of this pattern were necessarily abstract, these guidelines are grounded in direct experience across five different data domains (ERP financial data, BIM building models, fleet management, energy monitoring, and construction standards) and documented in an operational metadata guide [23].

Phase 1 — Research

Have the AI research best practices for MCP tool design and metadata standards. Study what makes tool descriptions effective for AI reasoning. Analyze existing high-quality MCP servers for patterns worth adopting.

What specifically to research. The research phase should produce a clear understanding of which metadata delivery channels actually work. The hierarchy below reflects production testing across multiple Claude clients as of early 2026 [23]. This landscape is evolving — as MCP clients mature, model providers improve tool-use training, and the specification adds new primitives, these tiers may shift. Use them as a validated starting point, but re-evaluate against the clients and models you're actually deploying to. The guidelines described in this paper have been found to work in practice, but "works today" is not "works forever."

Tier 1 — Always works, all tested clients (~95% of the value). Tool descriptions (when to call, what it returns, how to use it). Input schema with .describe() annotations (formats, codes, relationships). Output schema with .describe() on every field (field-level documentation the agent reads on every call).

Tier 2 — Works sometimes, client-dependent. Server instructions—cross-tool behavioral rules injected at session start. Claude Code and Claude Desktop inject them into the system prompt; other clients may not. Useful for global rules ("always report friction via report_problem") but not a substitute for per-tool descriptions.

Tier 3 — Does not work in practice (as of early 2026). MCP Resources (agents never request them across any tested Claude client). Meta-tools for reference lookups (agents never call parameterless guide tools—they infer what they need from input schema descriptions). These were built, deployed, and tested; both were removed after validation [23]. This tier is the most likely to change — if clients begin surfacing resources proactively or models are trained to request them, resources could become a viable channel for the richer metadata that currently must be compressed into tool descriptions.

Key validated insight: *as of this writing, everything the agent needs must live in the tool description + input/output schema. Resources and meta-tools for reference content that agents can already infer are not used. Tool descriptions are the only reliable delivery channel — but this is a client limitation, not a protocol limitation, and it may improve.*

Two additional spec features were evaluated and set aside. Sampling (server-initiated LLM reasoning requests) is unnecessary when rich metadata already guides the agent's reasoning through tool descriptions — the INTERPRETATION blocks and `.describe()` annotations do what sampling would do, but through metadata that's always present rather than a runtime request. Elicitation (server-initiated user input requests) is unsupported by key clients including Claude Desktop; MCP Apps provide a more capable alternative with full

interactive forms, typed validation, and multi-step workflows. Both cases follow the same principle: work with what clients actually support today, not what the spec promises.

Phase 2 — Generate Guidelines

Have the AI create a comprehensive guideline document: what types of metadata to include, how to structure cross-references, what format to use for query strategies and fallback routes. This becomes the blueprint for Phase 4.

Description block structure. The guidelines should define a standard set of blocks for every tool description. This block structure is not adopted from an existing standard—it emerged iteratively during production implementation, driven by observed agent failure modes. Each block was added because the agent failed without it. Recent academic research independently validates the need: an analysis of 856 tools across 103 MCP servers found that 97.1% of tool descriptions contain at least one "smell"—unstated limitations, missing usage guidelines, opaque parameters [24]. The block structure below addresses exactly these deficiencies. In the production implementation, each connector uses these blocks [23]:

Block	Purpose	Without it...
RETURNS	Agent knows which fields come back — drives tool selection	Agent can't determine if this tool has the data it needs
WHEN TO USE	Agent knows when this tool fits	Picks wrong tool or misses this one
WHEN NOT TO USE	Prevents wrong tool selection; points to the correct tool	Agent tries this tool for queries that belong elsewhere
QUERY STRATEGY	Teaches summary-first, then drill down	Agent fetches full records when a summary would suffice
INTERPRETATION	Cross-field behavioral rules, type→field-group mappings	Returns raw numbers without conclusions
RELATED TOOLS	Agent chains queries automatically via join keys	Stops after first tool call
FEEDBACK	Agent reports friction to report_problem	Issues go undetected, descriptions never improve
ALERTS	Tells agent to expect and surface server-generated warnings from the response	Agent ignores domain-specific warnings (unapproved records, poor equipment condition, partial summaries)

Language strategy. LLMs internally process through English representations, even for non-English input. The production implementation uses an English-with-glosses bridge: tool descriptions, block headers, and .describe() annotations are written in English, but domain-specific terms from the source system are kept in their original language with an English gloss — e.g., "Medewerker (employee ID)" or "Geaccordeerd (approved)." Tool titles stay in

the user's language for UI display; descriptions in English for agent reasoning. The agent responds in the user's language automatically [23].

A note on examples. The production system uses Dutch field names and tool names throughout — the ERP is a Dutch product serving a Dutch organization. Examples in this paper have been translated to English for readability (e.g., get_employee was originally get_medewerker, EmployeeName was MedewerkerNaam). The pattern and the language bridge apply regardless of source language.

Factory patterns. The guidelines should specify shared patterns that eliminate boilerplate. In the production implementation, every connector shares the same input schema factory (filters, pagination, sorting, summaryOnly, queryIntent) and the same handler factory (permission check → build params → call connector → sanitize dates → transform → summarize → validate output → return). Each connector only supplies its field metadata, an optional transform callback for derived fields, and an optional summarize callback for domain-specific aggregation. The same factory pattern was implemented for both REST (ERP, fleet) and GraphQL (BIM) backends — the handler abstraction is API-style agnostic [23].

Phase 3 — Discover

Have the AI iterate over your actual data—not the schema, the data. But this is not a passive scan. You actively direct the exploration: instruct the agent to apply specific filters, sort by different fields, compare subsets. Crucially, force the agent to look beyond the obvious. The default behavior of any AI is to fetch the first page of results and draw conclusions from the top 10 rows—but the top 10 rows are rarely representative. A table sorted by date shows only the most recent records. Sorted by ID, only the oldest. The interesting patterns live in the edges: the records where fields are unexpectedly null, where status codes don't match the expected flow, where cross-references break down.

This is an iterative process. The first pass reveals surface-level patterns. The second pass, informed by what you learned, goes deeper. By the third or fourth pass, you're discovering business rules that aren't documented anywhere—because the people who know them never thought to write them down. In the production implementation, the discovery phase was repeated across multiple sessions over several days. Each iteration refined the understanding: early passes caught null patterns and obvious code meanings; later passes uncovered conditional field population rules, cross-table join reliability issues, and exception patterns that only appeared when filtering for specific record types.

Direct the agent explicitly. Don't just say "examine the data." Give it specific instructions: filter by a particular type or status, sort by a numeric field descending, skip the first 50 records and look at what's underneath. Then change the filter and compare results. Then look at records where a key field is unexpectedly null and ask what other fields those records have in common. Each directed query teaches the AI something specific, and each answer informs the next question. You are guiding a structured exploration, not requesting a report.

Instruct the agent to flag uncertainty with confidence levels. When the AI encounters patterns it can observe but not explain — a status code that appears in unexpected contexts, a field that's populated for some record types but not others, a value distribution that doesn't match the schema documentation — instruct it to write a TODO marker with a confidence level immediately. Not after discovery, during discovery. "When you find something, tag it: HIGH if confirmed across many records, MEDIUM if observed but possibly incomplete, LOW if inferred from limited data. Always add a `[CONFIDENCE: <level> — <observation>. TODO: DOMAIN EXPERT — <question>]` note." These TODOs become the agenda for Phase 4's validation step, prioritized by confidence level — LOW markers get reviewed first because they represent the largest knowledge gaps. Without them, ambiguous discoveries

get silently resolved with the AI's best guess — which for enterprise-specific domain knowledge is often wrong.

This is where the AI finds things no API documentation contains. In a production ERP deployment, the AI discovered that the "assigned technician" field on maintenance location records was never populated despite being in the schema, that the actual technician could only be found through time-booking records filtered by a specific work-order type, and that equipment condition scores did not correlate with age—a 2001 installation could score better than a 2008 one due to maintenance quality. None of these discoveries came from the first query. They emerged from directed, repeated exploration of differentiated data slices.

What the AI should look for:

- **Null patterns:** which fields are consistently empty for which record types? In the production system, the AI discovered that EmployeeName is always null for article bookings (TypeItem='Mat') and TicketId is null for project bookings but populated for service bookings. These null patterns become .describe() annotations that prevent the agent from expecting data where there is none.
- **Code-to-meaning mappings:** what do the values actually mean? Status codes, type codes, category codes—the AI reads real data and documents what each value represents in practice, not in theory.
- **Cross-table join paths:** which ID fields connect which tables, and are those connections reliable? The AI discovers that TicketId in the actual-costs table links to the service ticket table, that EmployeeId is a join key to the employee table, and that TicketLineNumber on maintenance orders maps 1:1 to subscription line numbers.
- **Business rules hidden in data:** the AI finds that service bookings on numeric-only project IDs (e.g. 6600xxx) are contract-covered and don't generate per-visit invoices, that project ID letter prefixes encode project types, and that the invoicing flow follows Approved → Completed → Invoiced sequentially.
- **Cross-tool relationships:** when your server exposes multiple tools, instruct the agent to explore how they relate. Have it fetch a record from one tool, then use an ID field from that record to query another tool. Does the join work? Is the ID always populated? Does the related record add context that the first tool alone couldn't provide? In the production implementation, the AI discovered that the planned-vs-actual-vs-invoiced triangle (budget → actual costs → invoice lines) shared ProjectId as a reliable join key, but that the service ticket link only worked for service bookings, not project bookings. These cross-tool relationships become the RELATED TOOLS block in tool descriptions and the join-key annotations in output field .describe() entries — they are what enables the agent to chain tool calls instead of stopping after the first one.

Phase 4 — Write Back

Apply discoveries as operational metadata in the tool descriptions. This is where the five phases diverge most sharply from traditional approaches. The knowledge doesn't go into a wiki, a data catalog, or a RAG index. It goes directly into the tool's description, input schema, and output schema—the only channels the agent reliably reads.

Six types of knowledge get encoded. These map directly to the description blocks defined in Phase 2: domain interpretation populates the INTERPRETATION block, cross-tool references populate RELATED TOOLS, query strategies populate QUERY STRATEGY, fallback routes populate WHEN NOT TO USE, and uncertainty markers populate TODO annotations throughout. The RETURNS, WHEN TO USE, and FEEDBACK blocks provide the structural scaffolding that organizes this knowledge for the agent. The ALERTS block

serves a different role: it doesn't define alert content in the description — it instructs the agent to expect server-generated warnings in the alerts array of every response and to always surface them to the user.

1. Domain interpretation — what codes and statuses mean in the business context.

> *TypeItem: 'Lab' = labor hours, 'Cost' = costs,*
> *'Mat' = physical articles/materials*
> *ChargeStatus: derived from 3 boolean flags into one readable label*

2. Cross-tool references — join keys and resolution paths between tools, documented in both the description's RELATED TOOLS block and in output field .describe() annotations.

> *EmployeeId — join key to get_employee*
> *TicketId — join key to get_service_ticket and get_actual_costs*
> *VisitNumber = TicketId — for invoice verification*
> *Triangle: planned costs → actual costs → invoice lines*

3. Query strategies — performance guidance and multi-phase patterns.

> *ALWAYS start with summaryOnly=true. Active projects*
> *accumulate thousands of records. Summary returns byMonth,*
> *byEmployee, byTypeItem, approval counts — no records.*
> *Then drill down with date range + type filters.*

4. Fallback routes — alternatives when the expected path doesn't work.

> *MaintenanceLocationTechnician is NOT populated in practice.*
> *To find the technician:*
>
> - *get_actual_costs with TicketType=52 → EmployeeName*
> - *get_ticket_responses → technicianFeedback*

5. Hierarchical relationships — parent-child structures.

> *Type 48 (service planning) → parent type 10 (servicemelding)*
> *Type 13 (onderhoud opdracht) → parent type 32 (onderhoud locatie)*
> *Type 52 (onderhoud planning) → parent onderhoud locatie*

6. Uncertainty markers — explicit TODO flags for knowledge the AI discovered but cannot fully verify alone. The recommended pattern is a three-tier confidence system: HIGH (confirmed across large datasets — e.g., "observed across thousands of records"), MEDIUM (observed patterns that may have exceptions — e.g., invoicing flow rules), and LOW (inferred from limited data or unclear whether it's configuration-specific — e.g., field meanings that might differ per environment). Every marker follows the same structure: confidence level, what was observed, and a specific question for a domain expert.

> *[CONFIDENCE: HIGH — observed across thousands of records.*
> *TODO: DOMAIN EXPERT — is the 0.17-8 hour range for Wst*
> *correct? Are there legitimate bookings outside this range?]*
> *[CONFIDENCE: MEDIUM — observed flow, but the ERP may allow*
> *skipping steps. TODO: DOMAIN EXPERT — is the invoicing*

> *flow Approved → Completed → Invoiced strictly*
>
> *sequential, or can steps be skipped?]*
>
> *[CONFIDENCE: LOW — only observed in test environment.*
>
> *TODO: DOMAIN EXPERT — is BasisverslagAanwezig=false a test*
>
> *artifact, or does it occur in production?]*

The confidence level tells the domain expert where to focus: LOW markers are the most valuable to resolve because they represent the largest knowledge gaps. Each resolved marker eliminates an entire category of potential misinterpretation — permanently. In the production deployment, domain-expert review has been completed across the ERP connectors: roughly 90% of the AI-discovered metadata held up under review, with the remaining ~10% corrected. Resolved markers are removed from the tool descriptions as their questions are answered, so what remains inline at any given time is the open edge — not the total work done.

Implementation pattern: Two-Layer Data Enrichment

A critical write-back technique is making data self-documenting rather than explaining codes in tool descriptions. The production implementation uses two layers [23]:

Layer 1: Source-side description joins. The ERP's data connectors are configured to JOIN code fields with their human-readable descriptions. Every code gets a companion field: InstallationTypeCode → InstallationTypeDescription, ItemCode → ItemDescription. The agent reads "Service-specialist" directly from the data instead of looking up a mapping table. This eliminates ~2,650 tokens of code tables from tool descriptions.

Layer 2: Server-side derived fields. For interpretations that don't fit as simple JOINs, the MCP server computes derived fields before returning data. The production implementation derives a single ChargeStatus label from two boolean flags (a line-level "is charged" flag combined with either a labor or cost "is charged" flag, depending on the record type). The result is one readable label per row — "Fully charged", "Line-only charged", "Internal with cost tracking", "Cost allocation", "Internal" — chosen from a closed enum the agent can reason about directly. This replaced ~200 tokens of boolean-combination explanation in the tool description with one .describe() line.

Implementation pattern: Annotation-driven field documentation

Every output field carries a .describe() annotation that serves as the agent's field-level documentation. These are not generic type descriptions—they encode domain knowledge:

> *EmployeeName: z.string().nullable().describe(*
>
> *'Employee full name. Null for Mat records.*
>
> *Use for display — only call get_employee*
>
> *for department/email.')*
>
> *ProjectId: z.string().nullable().describe(*
>
> *'Project code — letter prefixes encode project*
>
> *types (G=general, K=small-scale, etc.),*
>
> *numeric-only = service contracts.*
>
> *Null for records without project allocation.')*

Cross-tool join keys are documented directly in field annotations: EmployeeId — join key to get_employee. This is how the agent learns to chain tool calls—not from a separate reference document, but from the field itself.

Phase 5 — Feedback Architecture

The first four phases create the initial knowledge. Phase 5 keeps it improving in production through three layers. The concept of self-improving AI systems is not new—OpenAI's Self-Evolving Agents cookbook [13] and open-source projects like the Ultimate MCP Server's Autonomous Refiner [14] explore similar ideas. But these focus on prompt optimization and development-time simulation. The approach here operates on production runtime data, improving metadata rather than prompts.

Layer 1: Tool Call Logs (foundation — 70% of value). Log every tool call: request parameters, response summary, session ID, timestamp. No conversation text, no user questions, no agent reasoning—just the MCP layer. This is the primary data source. Three consecutive calls with tightening filters on the same table means the agent is struggling with something the metadata should have covered.

In the production implementation, every tool call—success, validation fallback, or error—writes to a persistent store with: tool name, connector ID, user, queryIntent, filter fields and operators (not values, for privacy), summaryOnly flag, row count, hasMore, duration, and error type. Session IDs correlate multi-tool sequences [23].

Layer 2: Pattern Analysis (intelligence — 20% of value). Periodic (weekly) analysis of raw logs. Detect repeated call sequences, redundant calls, unused tools that should have been used, tools called in unexpected order. Session IDs enable correlation. This finds silent failures—cases where the agent got a wrong answer without realizing it.

Layer 3: QueryIntent + Report Tool (accelerator — 10% of value). A queryIntent parameter on every tool call: one sentence describing the business question being answered. Plus a report tool the agent can use when genuinely stuck—no mandatory instructions, no nudges, just available if needed. Academic research on LLM metacognition shows mixed results—models can sometimes calibrate confidence but not reliably [12], and emerging work on metacognitive state vectors suggests this may improve [15]. This layer is designed as a bonus signal, not the foundation.

The critical design: session ID as join key

When an agent reports an issue, the session ID links to the complete tool call sequence that preceded it. You see not just what the problem was, but how the agent tried to solve it. That's the recipe for a targeted metadata fix.

Nudge patterns: teaching agents to self-report

The production implementation embeds contextual nudges in tool responses rather than relying on agent initiative [23]:

- **Empty results:** "No records matched your filters. If unexpected, consider calling report_problem with category 'unclear_filter'."
- **Pagination detected:** "You're paginating (skip=200). If you're doing this to manually aggregate data the summary doesn't provide, call report_problem with category 'pagination_struggle'—we may be able to add that summary dimension."
- **Incomplete summary:** "Summary is based on the first 500 records only—there are more. Tighten your filters for a complete summary."

- **Every response:** A feedback reminder as the last alert: "If this query took multiple attempts, returned confusing data, or required workarounds, call report_problem before your next step."

These nudges are injected by the handler factory—not by the agent's own judgment. They target specific friction patterns observed in production and give the agent a low-effort path to report issues it would otherwise silently work around.

05 Before and After

A concrete comparison using a universally understandable domain: building data. The same tool — looking up a building by address — described at Level 1 versus Level 4. An open-source extract of this tool — stripped of the production handler factory, security layer, cross-tool references, and feedback architecture, but preserving the core metadata patterns — is available as a standalone working example [28].

Level 1: Typical MCP Server

The entire tool definition:

> *name: get_building_profile*
>
> *description: Look up building information by postcode and house number.*
>
> *parameters: {*
>
> *postcode: { type: string },*
>
> *huisnummer: { type: integer }*
>
> *}*

No field documentation. No explanation of what gets returned. No cross-tool references. The agent has to guess what an energy label means, whether the building year is reliable, which postcode format the API accepts, and how to use the data for any follow-up analysis. The neighboring tools in this domain — `get_meters`, `get_usages`, `get_weather_context` — are equally hollow one-liners. The transport works; the interpretation layer is missing.

Level 4: After Introspective Context Engineering

The same tool, but with five layers of metadata that the agent reads and acts on. Each layer solves a different reasoning problem.

Layer 1: Structured description with behavioral blocks

The tool description is not a sentence — it is an operational manual organized into standardized blocks:

> *WHEN TO USE:*
>
> > *• "What kind of building is at [address]?"*
> >
> > *• "Does this building have an energy label?"*
> >
> > *• Before starting an energy analysis — get building context first*
> >
> > *• Cross-reference with get_usages: oppervlakte_m2 as denominator for kWh/m2 benchmarking*
> >
> > *• Cross-reference with get_weather_context: coordinaten.lat/lon for local weather data*
>
> *WHEN NOT TO USE:*
>
> > *• You already have the building profile and need meter data → use get_meters, get_usages*
> >
> > *• You need weather context → use get_weather_context (but coordinaten from this tool set the location)*
>
> *QUERY STRATEGY:*
>
> > *• Postcode: 4 digits + 2 uppercase letters, no space (e.g. "3543AR"). Huisnummer: integer only.*

> • ⚠️ *ADDRESS SOURCE WARNING: If you obtained the postcode from a smart meterconnection (get_meters), that is the EAN registration address at the grid operator —NOT necessarily the physical building address. Cross-validate with the ERP projectaddress before calling this tool.*
>
> • *If matchStatus = 'multiple_vbos': retry with huisletter (e.g. "A") to target a specific unit.*
>
> • *EP-Online coverage: most labeled utility buildings have data. Residential withoutrecent certification often lacks a label — energielabel null is normal for pre-2008 homes.*
>
> • *For kWh/m2 benchmarks: use oppervlakte_m2 (BAG) as denominator. For comparing againstEP-1: use gebruiksoppervlakte_thermische_zone_m2 (EP-Online) — these two areas differ by 10-30%.*
>
> *INTERPRETATION:*
>
> • *Which fields are populated depends on the berekeningstype:NTA 8800 (post-2021 labels): ep1, ep2, warmtebehoefte, compactheid populated. energie_index = null.NEN 7120 (commercial pre-2021): energie_index populated; ep1/ep2 = null.Nader Voorschrift (residential pre-2021): CRITICAL — berekend_energieverbruik is MJ totalbuilding, NOT kWh/m2. Do NOT benchmark these values.*
>
> • *Paris Proof 2040 targets — kantoor: 70 kWh/m2, woningbouw: 100 kWh/m2*
>
> • *label_geldig_tot in the past: label expired, may need renewal for Label C compliance*
>
> • *BAG oppervlakte_m2 vs EP-Online thermische_zone_m2: 10-30% difference is normal*
>
> *RELATED TOOLS:*
>
> • *get_usages (oppervlakte_m2 as denominator for kWh/m2 benchmarking)*
>
> • *get_weather_context (pass coordinaten.lat/lon directly for local degree days)*
>
> • *start_duurzaam (call this tool BEFORE starting an energy analysis)*
>
> *FEEDBACK: report_problem tool available if BAG lookup returns unexpected results.*
>
> *ALERTS: Always check interpretation.alerts — they contain bouwjaar era warnings,*
>
> *Paris Proof threshold breaches, label expiry notices, BENG compliance violations.*

Each block solves a specific agent failure mode. WHEN NOT TO USE prevents calling this tool when the agent already has the data. QUERY STRATEGY prevents the agent from passing an EAN registration address when the physical building address is needed. INTERPRETATION prevents the agent from comparing NTA 8800 kWh/m² values against Nader Voorschrift MJ totals — a mistake that would produce wildly wrong conclusions.

What the agent cannot figure out alone. The berekeningstype routing is invisible without metadata. A building certified under NTA 8800 (post-2021) returns ep1, ep2, and warmtebehoefte but null for energie_index. A building certified under NEN 7120 (pre-2021) returns energie_index but null for ep1 and ep2. A building certified under Nader Voorschrift returns values that look like energy metrics but use entirely different units (MJ total instead of kWh/m²). Without the INTERPRETATION block, the agent sees a number, assumes it's kWh/m², and gives confidently wrong advice. This is not a data quality problem. The data is correct. The interpretation requires domain knowledge that no model has in its training data.

The multi-API aggregation pattern. Behind this single tool, the server calls two separate government APIs: Kadaster BAG (building characteristics — year, floor area, intended use, GPS coordinates) and EP-Online (energy performance — label, certification method, CO2

emissions, BENG requirements) [27]. The agent calls one tool with a postcode and house number. It doesn't know — and doesn't need to know — that two APIs were involved. This is domain-aware routing that a Level 1 server cannot provide.

Layer 2: Input schema with typed, annotated parameters

Every input parameter carries .describe() documentation that teaches the agent how to use it:

> *postcode: z.string().regex(/^\d{4}[A-Z]{2}$/).describe(*
> *'Postcode in P6 format without space (e.g. 3751LN)')*
> *huisnummer: z.integer().positive().describe(*
> *'House number (integer only, no letter)')*
> *huisletter: z.string().optional().describe(*
> *'House letter (e.g. A, B) — use when matchStatus = multiple_vbos')*

The regex pattern on postcode prevents malformed queries. The huisletter parameter exists specifically for the disambiguation case described in QUERY STRATEGY — the schema and the description work together.

Layer 3: Output schema with field-level domain knowledge

Every output field documents not just its type, but its domain meaning, conditional population rules, and cross-tool relationships:

> *energielabel: z.string().nullable().describe(*
> *'Energy label A++++ through G. Null = no registered label in EP-Online.*
> *Common for pre-2008 homes and unlabeled buildings.')*
> *oppervlakte_m2: z.number().nullable().describe(*
> *'BAG gross floor area (m2). Use as denominator for kWh/m2*
> *benchmarking with get_usages. For EP-1 comparison, use*
> *gebruiksoppervlakte_thermische_zone_m2 instead (10-30% lower).')*
> *coordinaten: z.object({ lat: z.number(), lon: z.number() }).describe(*
> *'WGS84 coordinates. Pass directly to get_weather_context*
> *latitude/longitude parameters for local degree-day data.')*

The agent reads these annotations on every call. They tell it that a null energy label is normal (not an error), that two different surface area fields exist for different purposes, and that the coordinates feed directly into the next tool in the chain.

Layer 4: Server-side derived fields and multi-API aggregation

The server computes the matchStatus field that doesn't exist in either source API — it's derived from the BAG response to guide the agent's next action:

> *matchStatus: z.enum(['exact', 'multiple_vbos', 'not_found']).describe(*
> *'exact: single building unit found, high confidence.*
> *multiple_vbos: multiple units at this address — retry with huisletter.*
> *not_found: no BAG match — check postcode format.')*

This is metadata doing computation, not just documentation. The server examines the BAG response, determines whether disambiguation is needed, and tells the agent exactly what to do next.

Layer 5: Runtime alerts in responses

Every response includes contextual alerts generated server-side:

> *// Building year warning (suppressed for good labels A/A+):*
>
> *"Bouwjaar 1974 — pre-1992 buildings typically have poor insulation.*
>
> *Consider energy audit."*
>
> *// Paris Proof threshold breach:*
>
> *"EP-1 energiebehoefte 142 kWh/m2 exceeds Paris Proof 2040 target*
>
> *of 70 kWh/m2 for kantoorgebouwen."*
>
> *// Label expiry:*
>
> *"Energielabel registered 2013-03-15, geldig tot 2023-03-15 —*
>
> *label is expired. Heropname may be needed for Label C compliance."*

The agent never computes these warnings — it reads them from the response and surfaces them to the user. The server knows the domain rules; the agent delivers the message.

The Same Pattern Applied to Complex Enterprise Data

The building profile example demonstrates the pattern using universally understood data — addresses, building years, energy labels. But the same five-layer structure works identically on deeply domain-specific data that only industry insiders understand.

In the production ERP deployment, the same block structure is applied to a financial actuals tool (get_actual_costs) with domain knowledge that includes: type codes that determine which fields are populated ('Lab' populates employee name, 'Mat' has null employee), a three-flag charge status derived into a single readable label, contract-covered work detection (numeric-only project IDs indicate subscription billing, not missing invoices), a 13-status service flow, cross-tool join keys connecting actuals to budgets to invoices, and a summaryOnly pattern that aggregates thousands of records server-side to avoid context overflow. The metadata is entirely different. The structure — WHEN TO USE, WHEN NOT TO USE, QUERY STRATEGY, INTERPRETATION, RELATED TOOLS, FEEDBACK, ALERTS — is identical. The pattern transfers.

The Level 1 server exposes an API. The Level 4 server teaches the agent how a domain expert thinks about this data — what to query first, what the values mean, when to chain tools, and when to report friction.

06 Why It Works

What the Feedback Architecture Reveals in Practice

To see why the system works, consider a concrete example. The same three-call sequence, with and without the queryIntent parameter:

Without Intent:

> *09:14:03 | get_service_records | summaryOnly=true, project=P-7056*
>
> *09:14:07 | get_service_records | type=13, project=P-7056*
>
> *09:14:09 | get_service_records | type=13, project=P-7056, completed=true*

Three calls. The last one added a filter. Why? Unknown from the log alone.

With Intent:

> *09:14:03 | intent: Overview of all service records for project P-7056*
>
> *09:14:07 | intent: Which maintenance orders exist for this project*
>
> *09:14:09 | intent: Only completed orders — previous call also returned open items*

The third call reveals exactly what went wrong: the agent expected only completed items but the metadata didn't make clear that a completion filter is needed. That's a targeted metadata improvement requiring minutes to implement. This is the system working as designed — the feedback architecture identifies the gap, and a metadata update fixes it permanently.

The Underlying Properties

Introspective Context Engineering produces a two-layer intelligence system. The metadata layer provides domain knowledge discovered through data exploration. The reasoning layer (the LLM) fills remaining gaps using that knowledge as foundation. This creates several powerful properties.

Graceful degradation. Imperfect metadata does not crash the system—it gives the LLM less to work with. Every improvement eliminates an entire category of uncertainty, freeing reasoning capacity for harder problems.

Compound improvement. Each resolved TODO marker makes the entire system permanently smarter. The metadata layer grows richer over time while the reasoning layer stays the same. This is cheap, permanent, compounding progress.

Transferable pattern. The five phases work identically regardless of data source, API style, or domain. The production implementation validated this across seven servers covering five fundamentally different backend systems:

	ERP	BIM (Autodesk AEC)	Fleet management	Energy monitoring (mixed)	Construction standards
API style	REST	GraphQL	REST	REST (5 APIs)	REST
Tools	11 (9 data + 1 app + feedback)	7 (6 data + feedback)	12 (11 data + feedback)	10 (8 data + 1 app + feedback)	4 (3 data + feedback)

	ERP	BIM (Autodesk AEC)	Fleet management	Energy monitoring (mixed)	Construction standards
Data shape	Relational, flat records	Hierarchical, spatial	Time-series, GPS telemetry	Time-series, building profiles, weather data, cadastral records	Hierarchical classification
Key discoveries	Invoice triangle (budget → actual → invoiced), null patterns per record type, technician fallback path, contract-covered work detection, ItemCode groupings per discipline	Hub → project → model → element navigation, HVAC system classifications, pipe/duct specifications, API pagination quirks (silently returns 0 at limit=100+)	Driving behavior scoring (0-100 scale), idling detection, speed alerts (>130 km/h), unit inconsistencies (odometer in meters, distances in km), score API timeout at >1 week ranges	Smart meter vs building automation as separate tools with metadata-guided selection, weather context enrichment via meteorological API, building profile from two government registry endpoints, Priva variable discovery across 30K-60K sensors per building	NL-SfB classification hierarchy, code → description resolution
Domain metadata	Financial interpretation, cross-tool join keys, charge status derivation, 13-status service flow, 30-status quote lifecycle	Spatial relationship navigation, element classification hierarchies, Revit category mappings, custom parameter filtering limitations	Vehicle-driver assignment patterns, trip type classification, fuel/CO2 metrics, safety alerts (seatbelt violations)	Energy analysis methodology, building profile enrichment from two government APIs, sustainability scoring, five-API routing logic hidden behind unified tools	Construction standard lookups, code validation
Cross-system links	—	Project ID mapping to ERP	Driver erpCode → ERP employee ID (~106/128 populated)	search_projects → ERP project ID	NL-SfB codes → BIM element classification
Confidence markers	59 (9 HIGH, 24 MEDIUM, 26 LOW)	Verified project mappings,	Data quality flags (empty erpCode,	Data source routing validated	Code hierarchy validated

	ERP	BIM (Autodesk AEC)	Fleet management	Energy monitoring (mixed)	Construction standards
		API limitation flags	missing CAN bus fuel data)		
Build time	~40 hours (first server, from scratch)	~1–1.5 days (reusing patterns)	~0.5 day (full foundation in place)	~1 day (energy domain discovery)	~0.5 day (classification mapping)

Additionally, a **Utility server** provides three interactive MCP App tools (render_chart, render_table, render_map) that every data tool in the ecosystem can use for presentation, and a **Games server** demonstrates MCP Apps with Firebase-backed leaderboards. Across all seven servers, every tool includes a dedicated `report_problem` feedback tool — the same pattern, replicated per server for session-correlated friction tracking.

Each domain required entirely different metadata — but the five-phase method, the description block structure, the handler factory pattern, and the feedback architecture were identical across all seven servers. The BIM server operates on a GraphQL API rather than REST; the energy server is a mixed-backend server aggregating five separate APIs (a smart meter data provider, a building automation platform, a meteorological API, and two government building registries) behind a unified tool interface — separate tools exist for smart meter data (`get_usages`) and building automation data (`get_priva_usages`), and the metadata's WHEN TO USE / WHEN NOT TO USE blocks guide the agent to the correct tool for each building. The agent doesn't need to know the backend architecture; it reads the tool descriptions and picks the right one. The standards server maps a hierarchical classification system. The same introspective approach — discover what the data means through actual exploration, write it back as operational metadata — applies to any structured data source.

Agent-agnostic by design. Because the domain intelligence lives in the MCP server — not in the client — a single Rich MCP Domain Server can serve multiple AI agents simultaneously. Claude.ai, Microsoft Copilot agents, Cursor, custom-built assistants: any MCP-compatible client benefits from the same metadata without duplicating the knowledge investment. Build the domain layer once, connect it to every agent your organization adopts. As the ecosystem of MCP clients grows, so does the return on every hour spent writing metadata.

Enterprise-grade security. All seven servers share a common security layer: Microsoft Entra ID OAuth 2.1 authentication with JWT validation on every tool call, audience and domain restrictions, and role-based access control (RBAC) scoped per tool. Only authenticated employees within the organization can access any tool. The security infrastructure — OAuth proxy, JWT verification, RBAC checks, session correlation — is implemented once in a shared factory and reused by every server, adding zero security engineering per new deployment. Tool call logging to Firestore provides a complete audit trail: tool name, user identity, query parameters, session ID, and timing — without capturing conversation content.

No new infrastructure required. The upgrade from Level 1 to Level 4 needs no infrastructure beyond what any MCP server already deploys — only better content in fields the protocol already defines.

Server-side summarization. The production implementation moves analytical computation from the agent to the server through a `summaryOnly` pattern. When set to true, the server paginates through up to 5,000 matching records internally and returns pre-aggregated

domain-specific breakdowns — by month, by employee, by project, by item type, by installation type, by condition score — plus financial totals and approval counts. Each connector supplies its own `summarize` callback that produces different aggregation dimensions relevant to its domain, along with contextual alerts (e.g., "23% of records are unapproved — approval backlog?" or "summary is based on the first 500 records only — tighten your filters"). One `summaryOnly` call answers most analytical questions ("who worked on this project?", "how are hours distributed?", "what was the busiest month?") without returning a single individual record. This is server-side intelligence that no amount of prompt engineering on a Level 1 server can replicate — the server does the computation, the agent does the interpretation.

Metadata as instruction set. The metadata serves a dual purpose: it documents the domain for human maintainers and simultaneously instructs the AI agent at call time. There is no separate "AI training" step — the same tool descriptions, `.describe()` annotations, and INTERPRETATION blocks that a new developer reads to understand the system are exactly what the agent reads to decide how to query, interpret, and chain tools. Write better metadata, and every connected agent immediately behaves better. This eliminates the duplication inherent in client-side approaches like skill files, where domain knowledge must be maintained in two places — the server and the instruction file — and kept in sync manually.

07 Honest Limitations

Intellectual honesty requires acknowledging what this pattern does not solve.

Domain expert dependency. The 40 hours are not 40 hours from zero. They are 40 hours from someone who already works with the data daily and can validate what the AI discovers. The AI generates the metadata; a human confirms it. Without someone sitting on top of the data daily, the error margin is higher and the TODO markers take longer to resolve.

Single-tenant depth. The current implementation serves one company, one ERP configuration, one domain. The same status code can mean different things at different companies running the same ERP. Scaling to multi-tenant requires per-tenant metadata variants—not a vector database, but more than one set of tool descriptions.

From data to applications (Levels 5 and 6). Levels 1–4 address understanding — how well does the server help the agent interpret data? But the production implementation has already moved beyond understanding into two additional capabilities that are different in kind, not just degree.

Level 5: Interactive MCP Apps (read). The production deployment includes server-rendered interactive UI components — charts (9 types including sankey flows), sortable/filterable data tables (8 cell types including badges, icons, currency formatting), geographic maps with typed markers, and a guided energy analysis intake form. Each MCP App is built as a standard Angular + Tailwind component, compiled by Vite into a single self-contained HTML file that the MCP server serves directly — no separate hosting, no iframe URLs, no build infrastructure beyond what any frontend developer already uses. These are not client-side widgets; they are MCP App tools served by the server, rendered directly in the conversation. The agent selects the right visualization and parameters; the server controls the presentation. Even a perfect Level 4 server with flawless domain knowledge still relies on the agent to present 847 rows of data as text. A Level 5 server renders that data as an interactive, properly-formatted table — directly from the server. The effect is combinatorial: each new visualization tool multiplies the presentation capabilities of every data tool in the ecosystem.

Level 6: Secure Write Apps (graduated deployment). Because every tool call is already authenticated via Entra ID, authorized via RBAC, and logged to Firestore, the write security concern is narrower than it first appears: the question is not "can someone unauthorized do this?" — that's already solved — but "can the agent trigger a mutation without the user intending it?" This focused threat model makes bounded writes practical.

The production implementation already includes agent-initiated bounded writes: every server has a `report_problem` feedback tool that writes structured friction reports to Firestore (typed Zod schema, session-correlated, categorized by friction type), and a game server writes leaderboard scores to Firebase. These are low-risk, bounded writes through standard model-visible tools — the agent calls them directly, the schema constrains what can be written, and every call is logged.

For business-critical writes — ERP corrections, invoice approvals, status updates — a more restrictive pattern has been designed and validated: the WriteIntent architecture. The agent opens an MCP App form (model-visible tool), the server mints a one-shot WriteIntent bound to the user and scoped to a specific resource, the user makes corrections in the form, and the form calls a commit tool that is hidden from the agent (`visibility: ["app"]`). The server validates the intent (same user, not expired, not consumed) before executing the write. Security properties: agent cannot see the commit tool, agent cannot obtain the intentId (stored in `structuredContent`, not `content`), intent is user-bound, resource-scoped, one-shot, time-limited (15 minutes), RBAC re-checked at commit, and both open and commit

calls are logged to Firestore. The blast radius is bounded even if every host-layer guarantee fails — each write requires two separate tool calls, each intent is single-use, and the trail is noisy and auditable.

The graduated approach — agent-initiated writes for low-risk operations, user-initiated writes for business-critical mutations — reflects a practical security model: match the write pattern to the risk level. This would allow Rich MCP Domain Servers to evolve into Rich MCP Domain Apps — not just reading and interpreting data, but safely writing it back through validated, schema-enforced interactions at the appropriate trust level.

Agent metacognition limits. Academic research shows mixed results on whether LLMs can reliably assess their own knowledge boundaries [12]. The feedback architecture accounts for this: tool call logs (always reliable) are the foundation, queryIntent (sometimes useful) is a bonus, and the report tool (rarely used without strong prompting) captures only genuine blocking issues. The system is designed for the agent's actual behavior, not ideal behavior.

Client fragmentation. Not all MCP clients handle metadata equally. Server instructions are injected by some clients but not others. Output schemas with structured content are supported by some but ignored by others. The implementation maintains both structured and text fallback responses for every tool call, and pre-validates output before the SDK's own validation to maintain control over error handling [23].

These limitations are real, but they are bounded. None of them invalidate the core pattern.

A note on evidence. This paper is a practitioner report, not a controlled study. The claims are grounded in production experience across seven servers and nine external APIs, not in comparative benchmarks. The natural next step would be to measure the pattern's impact quantitatively: before-and-after comparisons on answer accuracy, tool-call retry rates, token consumption, task completion time, and user satisfaction — ideally across multiple organizations and domains. The author considers this valuable future work that would strengthen the case from "this works in practice" to "this works measurably better." The codebase is an in-progress system, with confidence markers still being resolved with domain experts and metadata continuously improving through the feedback architecture — which is, in fact, exactly how the pattern is designed to work.

08 Recommendations for the MCP Framework

The limitations described above are not inherent to the protocol—they are gaps in the current specification and client ecosystem. Based on production experience building a Level 4 Rich MCP Domain Server, these are the changes that would make the biggest difference.

1. Smarter Tool Discovery

Today, every connected MCP server dumps all its tools into the agent's context at once. For a single dedicated agent connecting to one domain server, this is fine — 8-10 tools with rich descriptions fit comfortably in context. The problem emerges at the client level: a developer in an IDE with 10 MCP servers connected simultaneously faces 100+ tool descriptions competing for context. This is not a server-side problem — it's a client-side discovery problem. But it has a chilling effect on metadata investment: why write 200 lines of rich domain knowledge per tool if the client is going to flood the context with descriptions from every connected server? The specification needs a discovery layer: a way for clients to query available tools by category, domain, or capability before loading them into context. Think of it as DNS for tools — resolve first, load second. A simple approach: let servers declare tool groups with short summaries. The client presents groups to the model, the model selects the relevant group, and only those tool definitions are injected. This is not hypothetical — it mirrors how the skills pattern already works, and it is one of the things skills genuinely do better than MCP today.

2. Extensible Metadata Without Context Flooding

The current specification forces all metadata into tool descriptions and input schema descriptions—both of which consume context tokens on every call. There is no mechanism for metadata that the agent can request on demand. Resources were designed for this, but in practice agents never request them autonomously (validated across Claude Desktop, Claude Code, and Cursor [23]). The framework needs a middle ground: a metadata field on tool definitions that clients can surface selectively. For example, a tool could declare 200 lines of field-level documentation, cross-reference tables, and query strategy guides as structured metadata. The client loads the tool's name and short description into context by default, but injects the full metadata only when the model indicates it wants to call that tool. This would eliminate the current trade-off between rich documentation and context efficiency—a trade-off that currently forces server authors to compress critical domain knowledge into artificially short descriptions.

3. Resources That Actually Work

MCP Resources are the specification's answer to reference documentation—static or dynamic content that agents can request to inform their reasoning. In theory, this is the perfect channel for domain guides, field dictionaries, and cross-reference tables. In practice, no tested client reliably surfaces resources to agents. Claude Desktop lists them in a sidebar but agents don't request them. Claude Code and Cursor ignore them entirely. The production implementation built resources, tested them, and removed them after validation showed zero agent-initiated requests [23]. For resources to fulfill their design intent, clients need to either: (a) automatically inject relevant resource content when a related tool is selected, (b) allow servers to mark resources as required_context for specific tools, or (c) give the model an explicit get_resource primitive that it is trained to use. Without at least one of these, resources remain a specification feature that does not work in the real world.

4. First-Class Feedback Loops

The current specification has no concept of agent-to-server feedback. Every interaction is request-response: the agent calls a tool, gets data back, and moves on. There is no

standard way for agents to report confusion, flag data quality issues, or indicate which tool calls were unhelpful. The production implementation solved this by building a custom report_problem tool and injecting queryIntent parameters into every tool's input schema [23]. This works, but it is a workaround. The specification should support feedback as a first-class primitive: a standardized way for agents to annotate tool calls with intent, satisfaction, and issues encountered. A feedback field on tool responses. A report method on the protocol itself. Server authors could subscribe to feedback events and use them to improve metadata iteratively—closing the loop that currently requires custom infrastructure.

5. Agent Training on MCP Patterns

Perhaps the most impactful change would happen not in the specification but in model training. Current foundation models are not trained on rich MCP interactions. They don't know that a tool description containing WHEN TO USE and FALLBACK blocks should be parsed differently than a generic API description. They don't understand that .describe() annotations on output schema fields are there to be read and used in responses. They don't reliably call meta-tools or request resources because they were never trained to. If model providers included rich MCP server interactions in their training data—tool descriptions with domain metadata, multi-tool sequences with cross-references, feedback loops with intent parameters—agents would naturally leverage the metadata that server authors invest in writing. The metadata is already machine-readable. The models just need to be taught to read it. This would transform the entire ecosystem: server authors would be rewarded for writing rich metadata because agents would actually use it, creating a virtuous cycle that the current training gap prevents.

6. Scoped Write Permissions for MCP Apps

As MCP servers evolve from read-only data sources toward interactive applications, write operations become inevitable. But the current specification offers no mechanism to restrict a write tool to a specific interaction context. Today, if a server exposes a POST tool, any connected agent can call it at any time — the only protection is either generating short-lived tokens server-side or adding an isApprovedByUser boolean parameter that the agent can simply set to true without actual user approval. Neither is real security.

The production implementation has designed and validated a workaround: a two-tool pattern where the agent calls a model-visible "open" tool that mints a server-side WriteIntent, and the actual mutation is performed by a separate "commit" tool registered with `visibility: ["app"]` — hidden from the agent, callable only by the MCP App form. The intent is user-bound, resource-scoped, one-shot, time-limited, and validated with typed Zod schemas per write operation. Both calls are logged. This creates a real security boundary despite the specification not supporting one natively.

The specification should formalize this pattern: a way to scope write tools to specific MCP apps, so that a tool marked as write-only-via-app can only be invoked through a validated form interaction, not through freeform agent reasoning. The `ext-apps` extension already supports `visibility: ["app"]`, but this is advisory — the underlying `tools/call` endpoint remains callable regardless. Making write scoping a protocol-level guarantee rather than a host-level hint would let servers safely expose bounded write operations — creating a time booking, approving an invoice, updating a status — without giving the agent unbounded write access. The schema enforces what can be written; the app scope enforces when and how. This is the missing primitive that would allow Rich MCP Domain Servers to evolve into Rich MCP Domain Apps.

These six recommendations share a common theme: the MCP specification is designed for transport, but the real challenge is knowledge delivery and safe interaction. The protocol excels at connecting agents to tools. It does not yet help agents understand what those tools

mean, or safely act on what they understand. Addressing these gaps at the framework level would raise the floor for every server in the ecosystem — not just the ones built by teams willing to invest 40 hours in metadata engineering.

09 How to Apply This Monday Morning

This is not a framework to install. It is a method to follow.

Prerequisites, stated honestly. The build times quoted below assume TypeScript proficiency, direct access to a domain expert who knows the data daily, AI-assisted development throughout, and willingness to iterate on metadata for weeks after the initial deploy. Without these — especially the domain expert — the pattern still works but takes longer, produces lower-confidence metadata, and compounds more slowly. The 40-hour first build is what one developer achieved; it is not a benchmark.

1. **Pick your data source.** Your company's ERP, database, or API—anything with structured data where the domain is complex enough that AI can't figure it out from the schema alone. The more obscure or industry-specific the domain, the higher the value of this pattern.
2. **Build a basic MCP server.** Use the MCP TypeScript SDK. Expose your data through standard tools. This commodity layer takes hours, not days. An open-source example server provides a standalone starting point using Dutch building data — an extract of the building profile tool described in this paper, without the production infrastructure but with the metadata patterns intact [28].
3. **Run the five-phase pattern.** Have the AI research MCP best practices, generate a guidelines document, iterate over your actual data to discover domain knowledge, write it back into tool descriptions with typed input/output schemas, and set up basic logging for the feedback architecture. The guidelines document becomes your team's reference for consistent metadata quality across all tools.
4. **Validate with domain experts.** Review the TODO markers. Each one is a question for someone who knows the business. Each answer makes the system permanently smarter.
5. **Run, observe, improve.** Deploy to power users. Analyze tool call logs weekly. Identify where agents struggle. Update metadata. The system compounds.

> *In the author's implementation, the first server (ERP, 11 tools)—from blank MCP server to domain-intelligent assistant—took one developer roughly 40 hours, with no prior MCP server experience. The 40 hours were achieved with TypeScript proficiency, domain access, and AI-assisted development throughout — the same AI-first approach the pattern itself advocates. The rough breakdown: ~1 day to set up a proof-of-concept on a single tool, ~1 day to implement the Entra ID security layer with OAuth flow, ~1 day to add more tools and enrich the approach across connectors, and ~2 days to improve metadata in collaboration with domain experts resolving TODO markers.*
>
> *The pattern accelerates with reuse. The second server (BIM, 7 tools) took 1–1.5 days including metadata improvement — the handler factory, description block structure, and feedback architecture carried over directly. The third server (fleet, 12 tools) took half a day, building on all the foundation already in place. Subsequent servers (energy monitoring, construction standards, visualization, games) each took 0.5–1 day. Seven servers with 52 tools were built by one developer in approximately 8 weeks of cumulative effort, with each server benefiting from the patterns established by its predecessors. The intelligence lives in tool descriptions, not in infrastructure. The investment is time and domain expertise, not budgets and vendor contracts.*

10 The Bigger Picture

MCP is becoming infrastructure. The protocol tripled its enterprise adoption in twelve months, with 76% of AI solutions now purchased rather than built internally [18]. As MCP servers become standard integration points, the competitive advantage will not be having a server—it will be having a server that understands your domain.

The production implementation described in this paper is no longer a set of disconnected tools — it functions as a complete agentic AI platform. Seven servers with cross-system joins, shared authentication, interactive visualizations, and a common feedback architecture create a unified interface across five backend systems. Non-technical employees — service coordinators, project controllers, energy advisors — use it daily to query operational data, visualize trends, and make decisions informed by AI-interpreted data. A service coordinator asks about maintenance schedules and sees interactive tables with condition scores and equipment details. A project controller compares budgeted versus actual costs and sees the margin visualized in a chart. An energy advisor assesses a building's sustainability profile by combining government building data with smart meter consumption and weather context — across three APIs, in one conversation. These are not demos. They are production workflows that already influence how the organization allocates resources, prioritizes maintenance, and advises clients.

The Rich MCP Domain Server sits between expensive infrastructure and cheap wrappers. It is cheap to build, fast to deploy, and agent-agnostic. But there is a stronger reason to invest now: rich metadata is future-proof. Every improvement in foundation models will make a Rich MCP Domain Server work better, not worse — better models reason more effectively over structured metadata, follow cross-references more reliably, and handle multi-tool sequences with less confusion. The metadata you write today compounds with every model generation. A Level 1 API wrapper gains the least from a smarter model — there's nothing for the reasoning to latch onto. A Level 6 Rich Domain App gains the most.

> *The pattern is open. The method is documented. What matters now is who applies it first.*

References

[1] Challapally, A. et al. The GenAI Divide: State of AI in Business 2025. MIT Project NANDA, 2025. Based on 150 interviews, 350 employee surveys, and 300+ public AI deployment analyses.

[2] S&P Global Market Intelligence. 2025 Enterprise AI Survey. Survey of 1,000+ enterprises across North America and Europe. Reports 42% of companies abandoned most AI initiatives in 2025.

[3] Ryseff, J., De Bruhl, B.F., and Newberry, S.J. The Root Causes of Failure for Artificial Intelligence Projects and How They Can Succeed. RAND Corporation, RR-A2680-1, 2024.

[4] Informatica. CDO Insights 2025 Survey. Identifies top obstacles to AI success: data quality and readiness (43%), lack of technical maturity (43%), shortage of skills (35%).

[5] Gartner. Generative AI Pilot Forecast 2025. Predicts 50%+ of generative AI projects abandoned at pilot stage due to poor data quality.

[6] Smink, W. et al. Designing and Evaluating a Generative AI-Powered Chatbot for Enterprise Software Support. Journal of Systems and Software, January 2026.

[7] Peliqan. Text-to-SQL for ERP Data. Product documentation, 2025.

[8] Microsoft. Copilot for Dynamics 365. 2025. SAP. Joule: Generative AI Assistant. 2025. Oracle. NetSuite AI. 2025.

[9] K2view. GenAI Data Fusion Platform. Technical documentation, 2025.

[10] Anthropic. Model Context Protocol Specification. modelcontextprotocol.io, 2024–2026. Donated to the Linux Foundation under the Agentic AI Foundation, December 2025.

[11] Model Context Protocol. SEP-1382: Tool Annotations and Best Practices. GitHub specification proposal, status: dormant, 2025.

[12] Steyvers, M. and Peters, M.A.K. Metacognition in Large Language Models: A Survey. Sage Journals, 2025.

[13] OpenAI. Self-Evolving Agents Cookbook. OpenAI documentation, 2025. Post-hoc evaluation and prompt optimization from production traces.

[14] Dicklesworthstone. Ultimate MCP Server: Autonomous Refiner. GitHub, 2025. Development-time tool simulating agent failures.

[15] Courchaine, C. et al. Metacognitive State Vectors for LLMs. The Conversation, February 2026.

[16] McKinsey & Company. The State of AI: How Organizations Are Rewiring to Capture Value. March 2025.

[17] PubNub. Build vs Buy for MCP Servers. Technical analysis, 2025.

[18] Menlo Ventures. 2025 State of Generative AI in the Enterprise. December 2025.

[19] Holmes, E. MCP is dead. Long live the CLI. ejholmes.github.io, 2026. Argues CLIs provide equivalent value to MCP with less friction for developer workflows.

[20] Various. Skills as MCP alternatives. Developer community posts, Medium, 2026. Skills as self-contained markdown instruction sets that load on-demand.

[21] Cramer, D. MCP, Skills, and Agents. cra.mr, 2026. Counterargument that MCP, skills, and CLIs solve fundamentally different problems.

[22] jngiam. MCPs, CLIs, and Skills: when to use what? Blog post, 2026. Balanced analysis of complementary tool primitives.

[23] Golverdingen, D. MCP Server Metadata Guide — Rich Domain Server Patterns. Internal technical documentation, 2026. Operational metadata guide documenting validated patterns from production deployment across seven MCP servers: AFAS Profit (11 tools), Autodesk AEC Data Model (7 tools), Trips in the Cloud (12 tools), Warmtebouw Duurzaam (10 tools), Ketenstandaard (4 tools), Utility (5 tools), and Warmtebouw Games (3 tools).

[24] Wen, Y. et al. Model Context Protocol (MCP) Tool Descriptions Are Smelly! Towards Improving AI Agent Efficiency with Augmented MCP Tool Descriptions. arXiv:2602.14878, February 2026. Analysis of 856 tools across 103 MCP servers finding 97.1% contain description "smells" including unstated limitations, missing usage guidelines, and opaque parameters.

[25] Anthropic. The Complete Guide to Building Skills for Claude. https://resources.anthropic.com/hubfs/The-Complete-Guide-to-Building-Skill-for-Claude.pdf, 2026. Introduces skills as a client-side knowledge layer for MCP integrations, using markdown instruction files to teach Claude workflows and best practices for connected tools.

[26] Model Context Protocol. MCP Apps: Bringing UI Capabilities to MCP Clients. blog.modelcontextprotocol.io, January 26, 2026. First official MCP extension enabling servers to deliver interactive UI components (dashboards, forms, visualizations) directly in the conversation.

[27] Kadaster. Basisregistratie Adressen en Gebouwen (BAG) API and EP-Online Energielabel API. Public government APIs providing building characteristics (year, floor area, intended use, coordinates) and energy performance data (label, certification method, CO2 emissions). bag.basisregistraties.overheid.nl and ep-online.nl.

[28] Golverdingen, D. MCP Building Profile NL — Open-Source Example. GitHub, 2026. https://github.com/DaveGold/mcp-building-profile-nl. Standalone extract of the building profile tool described in this paper, using Dutch building data (Kadaster BAG and EP-Online). Demonstrates the core metadata patterns (structured descriptions, typed schemas, field annotations, behavioral blocks) without the production handler factory, security layer, cross-tool references, or feedback architecture.

About the Author

David Golverdingen is a software engineer with 18+ years of experience connecting complex systems to the people who use them. He started in industrial automation — programming PLCs, SCADA, and robotic controls for manufacturing and horticultural machinery — before moving into software engineering and data visualization in 2013, where his real-time monitoring work for energy and manufacturing clients earned a "Best in Show" award at the OSIsoft Visualization Hackathon. Seven years of freelance consulting followed, delivering senior frontend and full-stack engineering for de Volksbank, KLM/Air France, Allianz, Rijkswaterstaat (via Technolution), and Nederlandse Spoorwegen. The common thread: turning complex business requirements into applications that work at scale.

In September 2025 he joined Warmtebouw — a 300–400 person Dutch installation company — leading the development team. There, the gap between what AI could theoretically do with enterprise data and what it actually did became the central problem. The MCP servers described in this paper are the result: production infrastructure that lets entire teams query live ERP, BIM, fleet, energy, and building data through natural language. The pattern emerged not from AI research but from 18+ years of translating business problems into software solutions and turning industry trends into systems that people actually use.

About this paper. Written with the assistance of Claude (Anthropic) and curated, reviewed, and validated by the author. Implementation patterns and production observations reflect direct hands-on experience; AI assisted with research synthesis, structural organization, and prose. The domain knowledge and conclusions are the author's own.

David Golverdingen · MCP Architecture & Enterprise AI Engineering · April 2026 LinkedIn: https://www.linkedin.com/in/davidgolverdingen/