Odoo MCP Technical Overview (for Developers)

How to expose Odoo's ORM, business logic, and data to LLM agents through the Model Context Protocol — architecture, transport, tool design, authentication, and production hardening.

What MCP Is and Why It Fits Odoo

The Model Context Protocol (MCP) is an open JSON-RPC 2.0 protocol that lets an LLM client (Claude Desktop, an IDE agent, a custom orchestrator) call *tools*, read *resources*, and use *prompts* exposed by a server. For an Odoo developer, the mental model is simple: an MCP server is a thin adapter that turns Odoo's existing capabilities — `search_read`, `create`, `write`, workflow methods, reports — into a typed, self-describing API an LLM can discover and invoke at runtime. The LLM never touches the database directly; it only sees the tools you choose to publish.

This fits Odoo unusually well because Odoo already has a uniform RPC surface (XML-RPC and JSON-RPC over `/web/dataset/call_kw`) and a strongly-typed model layer. You are not inventing an API — you are wrapping one that already enforces access rights, record rules, and business constraints. The MCP server becomes a translation layer between MCP's `tools/list` + `tools/call` semantics and Odoo's `execute_kw(model, method, args, kwargs)` calls.

The three MCP primitives map cleanly: **Tools** = callable Odoo actions (create a lead, confirm a sale order, run a server action); **Resources** = read-only context the model can pull (an invoice as text, a product catalog, a report PDF); **Prompts** = reusable templated instructions (e.g. "draft a dunning email for overdue invoice X"). Most Odoo MCP servers start with tools only and add resources once the read patterns stabilize.

Reference Architecture

A typical deployment has four layers. (1) **MCP client / host** — the LLM app that speaks MCP. (2) **MCP server** — a standalone Python process (most commonly built with the official `mcp` Python SDK / FastMCP) that declares tools and translates calls. (3) **Odoo connector** — inside the MCP server, an `xmlrpc.client.ServerProxy` against `/xmlrpc/2/common` and `/xmlrpc/2/object`, or a `requests` session against the JSON-RPC endpoint. (4) **Odoo** itself — unchanged, behind its normal access-control stack.

You generally should *not* embed the MCP server inside the Odoo worker process. Keeping it as a sidecar process (its own container in your `docker-compose.yml`, alongside the Odoo and Postgres services) means it can be restarted, scaled, and secured independently, and an LLM bug can never take down Odoo HTTP workers. The MCP server holds no business logic of its own — that stays in Odoo modules where it belongs and stays reusable across clients.

For transport, MCP supports two main options. **stdio** is used when the client launches the server as a subprocess (ideal for local/desktop: Claude Desktop spawns `python odoo_mcp.py` and pipes JSON-RPC

over stdin/stdout). **Streamable HTTP** (the current spec transport, superseding the older HTTP+SSE) is used for remote/multi-user deployments where the server runs as a long-lived service behind a reverse proxy. Pick stdio for a single developer's machine; pick HTTP for a shared server like a VPS-hosted environment.

Defining Tools That Wrap the Odoo ORM

The core work is tool design. Each tool needs a name, a description the LLM reads to decide when to call it, and a JSON Schema for its arguments. With FastMCP this is mostly automatic from Python type hints. A minimal generic example:

- `@mcp.tool()` `def search_records(model: str, domain: list, fields: list[str], limit: int = 20) -> list:` — wraps `execute_kw(model, 'search_read', [domain], {'fields': fields, 'limit': limit})`.
- `def create_record(model: str, values: dict) -> int:` — wraps `create` and returns the new id.
- `def call_method(model: str, method: str, ids: list[int]) -> dict:` — for workflow actions like `action_confirm` on `sale.order`.

A generic `search_records` tool is powerful but dangerous: it gives the LLM the entire ORM. The more robust pattern for production is **narrow, business-named tools** — `create_crm_lead(name, email, phone, description)`, `get_overdue_invoices(partner_id)`, `confirm_quotation(order_id)`. Narrow tools have better descriptions, validate inputs explicitly, are far easier to permission, and the LLM chooses them more reliably because the intent is encoded in the name. Treat the generic tool as a power-user escape hatch, not the default surface.

Write tool descriptions for the model, not for humans. Include what the tool does, when to use it, the meaning of each argument, and crucially what it does *not* do ("This only drafts; it does not send."). Return structured, compact data — return the fields the model needs, not the full record. Every extra field is tokens and a chance for the model to hallucinate downstream. Map Odoo's `(id, name)` many2one tuples into readable strings so the model doesn't have to guess.

Authentication, Sessions, and Multi-Company

Odoo authentication in the connector is two-step over XML-RPC: `common.authenticate(db, login, api_key, {})` returns a `uid`, then every `object.execute_kw` call passes `db, uid, api_key, model, method, ...`. Use an **API key** (Settings > Users > API Keys), never a raw password, and never the `admin` superuser. Create a dedicated service user (e.g. `mcp_bot`) with exactly the groups it needs. This is your single most important security control: the LLM inherits this user's access rights and record rules, so Odoo's own ACLs become your guardrail.

For multi-user MCP servers you have a design decision: **shared service identity** (one Odoo user for all MCP calls — simple, but you lose per-user audit and record-rule isolation) versus **identity passthrough** (the MCP server maps the authenticated MCP client to a specific Odoo user/API key). Passthrough is correct for any deployment where different humans drive the agent, because Odoo's record rules and

`res.company` filtering then apply per real user. Store the per-user API keys in the MCP server's secret store, keyed by the MCP session's identity.

Multi-company is a classic trap (and a recurring source of subtle Odoo bugs). The `uid`'s allowed companies and the `allowed_company_ids` context determine what records are visible and which company stamps new records. Pass an explicit context — `execute_kw(..., kwargs={'context': {'allowed_company_ids': [company_id], 'lang': 'en_US'}})` — rather than relying on the user's default. Always set `lang` explicitly too; otherwise translated fields and reports come back in whatever the service user's language happens to be, which silently breaks output for multi-language databases.

Resources, Prompts, and Reports

Resources expose read-only context addressed by URI. A natural Odoo scheme is `odoo://res.partner/42` or `odoo://account.move/1015`, where the server resolves the URI to a `read` call and returns a clean text/markdown rendering. Resources differ from tools in intent: the client/user typically attaches a resource to the context deliberately, whereas tools are invoked autonomously by the model. Use resources for stable reference data the model should ground on — a customer's profile, an open invoice, a BOM — and tools for actions and dynamic queries.

Odoo's report engine is a high-value resource. You can expose a tool like `get_invoice_pdf(move_id)` that calls the report action (`account.report_invoice`) via `execute_kw` on `ir.actions.report` and returns the rendered document, or render the QWeb report to HTML/text for the model to read inline. This lets an agent answer "what's on invoice INV/2026/0042" by reading the actual rendered document rather than reconstructing it from raw fields — which avoids the model misformatting tax lines or totals.

Prompts are server-defined, parameterized message templates the client can list and fill. For Odoo, ship prompts that encode your operational playbooks: a "dunning email" prompt that takes `invoice_id` and produces a draft, or a "sales follow-up" prompt seeded with the lead's history. This is where reusable, productizable value lives — the same prompt library can be shipped across every client's MCP server, turning one-off agent instructions into a standardized asset.

Error Handling and Returning Useful Failures

Odoo raises typed exceptions that you must translate into MCP tool errors the LLM can act on. `AccessError` (no permission), `ValidationError` and `UserError` (business-rule violations, surfaced to the user), `MissingError` (record gone), and the generic `except_orm` / `XmlRpcFault` wrappers over RPC. Catch these in each tool and return a structured error message — for example, on `UserError` return the message text so the model can relay or correct ("Cannot confirm: customer has no delivery address"), rather than letting a raw fault traceback reach the model.

Distinguish **recoverable** from **fatal** errors in the response. A `ValidationError` usually means the model passed bad arguments and should retry with corrections — say so explicitly. An `AccessError` means the service user lacks rights — the model must not retry; it should report the limitation. Encode this hint in the error string. Set MCP's `isError` flag on tool results so the client treats it as a failure rather than silently feeding a fault string back as if it were data.

Validate before you call. Many Odoo errors are avoidable: check required fields, resolve display names to ids before write operations, and confirm a record exists with a cheap `search_count` before acting. Cheap pre-flight validation in the MCP server produces clearer errors than letting them surface deep in Odoo's `_check_constraint` stack, and it saves a round-trip plus a confused retry from the model.

Safety: Write Actions, Confirmation, and Scope Control

The hard problem with LLM + ERP is destructive or irreversible actions. Default to read-only and make writes opt-in. Practical controls: (1) **Separate read and write tools** and gate write tools behind a server config flag or a distinct, more-privileged MCP session, so a read-only agent literally cannot mutate data. (2) **Human-in-the-loop confirmation** for irreversible steps — model the action as "draft" plus a separate "confirm" tool, e.g. `create_quotation` leaves it in draft and a human (or an explicitly authorized step) calls `confirm_quotation`.

Constrain scope with Odoo's own mechanisms rather than logic in the MCP server. Record rules on the service user, field-level access (`ir.model.fields` groups), and a tightly scoped set of allowed models give you defense in depth that survives even if a tool is misused. A useful belt-and-suspenders pattern is an **allowlist of (model, method)** pairs in the MCP server config: even a generic `call_method` tool then refuses anything not on the list, so the LLM cannot reach `unlink` or arbitrary server actions you never intended.

Rate-limit and bound every query. Force a default and maximum `limit` on search tools so a model can't pull 200k records (and blow your context window and Odoo's memory). Add timeouts on the RPC client. Log every tool call with the resolved Odoo user, model, method, and arguments — both for audit and because it is the only way to debug why an agent did something unexpected three steps ago.

Deployment, Testing, and Productization

For local development, register the server with the client (e.g. Claude Desktop's config: command `python`, args pointing at your script, env vars for `ODOO_URL`, `ODOO_DB`, `ODOO_USER`, `ODOO_API_KEY`). The **MCP Inspector** (`npx @modelcontextprotocol/inspector`) is the fastest way to exercise `tools/list` and `tools/call` by hand before involving an LLM — verify each tool's schema and behavior against a test database, not production. For remote, run the HTTP-transport server as its own container behind your reverse proxy with TLS, and treat its API keys as production secrets.

Test against a throwaway Odoo database, never live data, when validating write tools — a copied DB on a non-production port lets you confirm `create` / `write` / `action_confirm` tools behave before any agent touches real records. Build a small fixture-based test suite that calls each tool through the connector and asserts on the returned shape; this catches schema drift when you upgrade Odoo versions (field renames between majors — e.g. removed `detailed_type` in 19 — will silently break tools otherwise).

The leverage play: an Odoo MCP server is highly repeatable across clients. Factor it as a parameterized template — connection from env, a base set of generic CRUD tools, plus a per-client module of business-named tools and a shared prompt library. The connector, error handling, auth, and safety controls are written once and reused; only the business tools differ per engagement. That turns "build an AI

integration" from custom consulting into a standardized, deployable product with recurring-revenue potential.

> **Start with a read-only stdio MCP server against a test database, wrap three to five business-named tools, validate them in the MCP Inspector, then add write tools behind confirmation — and keep the connector generic so you can redeploy it for the next client in an afternoon.**
