odooAI

# Odoo AI use-case library: 8 ready-to-deploy examples

Eight concrete, production-tested AI integrations for Odoo ERP — each with the exact trigger, model call, data flow, and a realistic before/after so you can ship the first one this week, not "someday."

## How to read this library (and the 3 questions that decide if AI belongs in your Odoo)

Most "AI in ERP" content is vapor: vague promises about "intelligent automation" with nothing you can actually build. This library is the opposite. Every one of the eight use cases below maps to a real Odoo trigger (a button, an incoming email, a record-create event, a scheduled action), a real model call, and a real place the output lands (a field, a chatter message, a draft record). You can copy the pattern into a `server action` or a small custom module and have it running in a sandbox the same afternoon.

Before you build anything, run each candidate through three filters. First, **frequency**: does this task happen often enough that automating it saves real hours? Classifying 5 tickets a week is not worth it; classifying 500 is. Second, **tolerance for being wrong**: AI is probabilistic, so the best use cases are *draft-and-review*, not *auto-commit*. Drafting a reply a human approves is safe; auto-posting a journal entry to your ledger is not. Third, **structured output**: if you need the AI's answer to flow back into Odoo fields, you must force JSON output and validate it — never parse free-form prose into your database.

A practical architecture note that applies to all eight: keep the AI call in one reusable Python helper (one method, one place that holds your API key in `ir.config_parameter` or an env var, one place that handles timeouts and retries). Every use case then becomes a thin caller of that helper. This is the DRY principle that turns eight one-off hacks into one maintainable capability — and it is what lets you reuse the same plumbing across every Odoo instance you run.

## Use case 1 — Auto-triage and route incoming Helpdesk / CRM emails

**The pain:** Every inbound email to support@ or sales@ lands in one undifferentiated queue. A human reads each one, decides if it's a bug, a billing question, a feature request, or a hot sales lead, sets priority, and assigns a team. On 200 emails a day that is roughly an hour of pure sorting, and the SLA clock is already ticking while messages sit unread.

**The build:** Hook a server action onto `helpdesk.ticket` creation (or `crm.lead` for sales). When a record is created from an incoming mail, send the subject + body to the model with a system prompt like: "You are a triage classifier. Return ONLY JSON: {category, priority, suggested_team, sentiment, summary_one_line}. Categories are exactly: bug, billing, how_to, feature_request, complaint, sales." Parse

the JSON, then set `ticket.team_id`, `ticket.priority`, and write the one-line summary to a custom field or the chatter. Negative-sentiment + "complaint" can auto-escalate priority to Urgent.

**Before/after:** Before, a tier-1 agent spends ~20 seconds per ticket triaging and misroutes ~15% of them, forcing reassignment. After, tickets arrive pre-tagged, the agent opens already-sorted queues, and the AI's one-line summary lets them decide in 3 seconds whether to grab it. The honest caveat: classification will be ~90-95% right, so keep the human as the final assignee confirmation for the first month and log every correction — those corrections become few-shot examples you paste back into the prompt to push accuracy higher.

## Use case 2 — Draft customer reply suggestions in chatter (human approves, never auto-sends)

**The pain:** Agents retype the same answers all day — "how do I reset my password," "where is my invoice," "what's your lead time." Each reply is 2-4 minutes of writing the same thing in slightly different words, and quality drifts when people are tired.

**The build:** Add a "Suggest reply" button (a server action) on the ticket form. When clicked, gather context: the full message thread from `mail.message`, the customer's name and language (`partner_id.lang`), and — critically — relevant snippets from your knowledge base (a few canned-response records or your docs). Send all of that to the model: "Draft a reply in {lang}, polite and concise, using ONLY the facts in the provided context. If the context is insufficient, say what info is missing instead of inventing an answer." Post the draft as an internal note (`message_post` with `subtype='note'`) so the agent reviews, edits one line, and sends.

**Before/after:** Before, average handle time on a routine ticket is 4 minutes of writing. After, the draft is 80% ready on arrival; the agent spends 45 seconds reviewing and personalizing. The two non-negotiable safety rails: (1) never auto-send — the agent always clicks send, which keeps a human accountable for tone and correctness; (2) the prompt must forbid invention ("use ONLY the provided context") to prevent the model from confidently stating wrong policy or fake order numbers. This is the single highest-ROI, lowest-risk use case for most service teams — start here if you're unsure where to begin.

## Use case 3 — Extract structured data from PDF invoices and bills

**The pain:** Vendor bills arrive as PDFs and scanned images. Someone keys in the vendor, date, invoice number, line items, tax, and total by hand into `account.move`. It's slow (3-6 minutes a bill), error-prone, and it's the classic job nobody wants. Odoo's built-in OCR exists but is a paid per-document add-on and still needs cleanup.

**The build:** When a bill attachment lands (or via a "Parse bill" button), pass the PDF/image to a multimodal model with a strict schema: "Extract into JSON: {vendor_name, vendor_vat, invoice_number, invoice_date, currency, lines:[{description, qty, unit_price, tax_rate}], untaxed_total, tax_total, total}. Dates as YYYY-MM-DD. If a field is not present, use null — never guess." Match `vendor_name` / `vendor_vat` against

existing `res.partner` records, create the draft `account.move` with extracted lines, and — this is mandatory — flag it **Draft** with a chatter note: "AI-extracted, verify totals before posting."

**Before/after:** Before, accounts payable keys every bill manually at 4 min each; on 300 bills/month that's 20 hours. After, ~85-90% of fields land correctly and the clerk's job becomes verify-and-correct at ~1 minute each — a 70%+ time cut. The hard rule: the AI never posts to the ledger. Extraction creates a draft; a human checks that `untaxed_total + tax_total = total` (validate this arithmetically in code and reject the extraction if it doesn't reconcile) and then posts. Financial data is exactly where probabilistic output must stay behind a human gate.

## Use case 4 — Natural-language product and inventory search

**The pain:** Salespeople and warehouse staff know what a customer wants in plain words — "the stainless 12mm bolt we sell to the marine clients" — but Odoo search needs exact SKUs, internal references, or category filters. People scroll, guess keywords, and ask colleagues, burning minutes per lookup and occasionally quoting the wrong part.

**The build:** Two-tier approach. The simple version: a search box where the user types natural language, the AI converts it to an Odoo domain — "return JSON Odoo domain for product.product matching this request," e.g. `[['name','ilike','bolt'],['x_material','=','stainless'],['x_size','=','12mm']]` — which you execute with `search_read`. The robust version for large catalogs: precompute embeddings of each product's name + description + attributes, store them (pgvector on your Postgres, or a small vector store), and do semantic similarity search so "marine fastener" finds stainless bolts even without the word "marine" in the record. Return the top 5 with stock-on-hand from `stock.quant`.

**Before/after:** Before, a catalog lookup is 60-90 seconds of scrolling and keyword roulette, with occasional wrong-part quotes. After, the rep types one sentence and gets the right 5 candidates with live stock in 5 seconds. Start with the domain-translation version (it's a one-day build and needs no embeddings infrastructure); only move to embeddings when your catalog is large enough or your customers' language is fuzzy enough that keyword matching genuinely fails. This is also a strong reusable asset — the same embedding pipeline powers product search across every catalog-heavy deployment.

## Use case 5 — Generate product descriptions, SEO copy, and translations at scale

**The pain:** A new catalog of 500 products needs website descriptions, short summaries for quotations, SEO meta titles/descriptions, and translations into the 3 languages you sell in. Doing this by hand is weeks of copywriting work, it's the kind of task that gets perpetually postponed, and inconsistent tone makes the webshop look amateur.

**The build:** A scheduled action (`ir.cron`) or a batch button that loops over products missing descriptions. For each, feed the model the product name, attributes, category, and any spec data: "Write a 60-word benefit-led webshop description, a 155-char SEO meta description, and a 60-char SEO title. Tone: professional, no hype. Return JSON." Then for translation, loop the result through Odoo's translation layer

— write to the field with the right language context (`with_context(lang='de_DE')`) so the model-term translations populate correctly. Process in batches of 20-50 with a rate-limit pause to stay inside API limits and control cost.

**Before/after:** Before, a copywriter does maybe 15-20 products a day; 500 products is two-plus weeks. After, the batch runs overnight and a human reviews/approves in a day. The realistic caveats: (1) generated copy needs a human skim for factual claims (don't let it invent certifications or specs the product doesn't have — feed it only real attribute data and tell it not to add features); (2) for SEO, spot-check that meta descriptions read naturally rather than keyword-stuffed; (3) machine translation is excellent for webshop body copy but get a native speaker to check anything legal or pricing-related. Even with review, this is a 10x throughput gain on a task that otherwise never gets finished.

## Use case 6 — Sales summaries and "explain this report" for managers

**The pain:** A manager opens the Sales or Inventory dashboard and sees numbers, but "what changed and why" takes manual digging. Weekly status updates to leadership are written by hand by someone who first has to pull the figures, eyeballing pivot tables for the story. That's an hour a week of a senior person's time, every week.

**The build:** A scheduled action that runs every Monday: query the week's data via `read_group` (sales by team, top products, won/lost opportunities, overdue invoices, stock-outs), assemble it into a compact structured payload, and send to the model: "You are a sales operations analyst. Given this data, write a 5-bullet executive summary: what moved, notable wins/risks, and one recommended action. Be specific with numbers. Do not invent figures beyond the data given." Email the summary, post it to a Discuss channel, or write it to a dashboard note. Bonus: add an "Explain" button on any report that sends the visible aggregated data and returns a plain-English readout.

**Before/after:** Before, the weekly leadership update is an hour of pulling-and-writing, and ad-hoc "why did the North region drop?" questions interrupt the analyst. After, the summary is drafted automatically with real numbers and the manager edits rather than authors. Critical guardrail: **only ever send aggregated figures you computed in Odoo** — never ask the model to do the arithmetic, because it will occasionally get sums wrong. The AI's job is narration and pattern-spotting on numbers you already trust, not calculation. Keep the math in Postgres; keep the prose in the model.

## Use case 7 — Internal ops copilot over your own SOPs and Odoo data (RAG)

**The pain:** New hires and even veterans constantly ask "how do we handle a return?", "what's the approval flow for a discount over 15%?", "which warehouse ships EU orders?" The answers live in scattered docs, in someone's head, or in undocumented Odoo configuration. Onboarding is slow and the same questions get re-answered forever — the opposite of knowledge compounding.

**The build:** A retrieval-augmented chatbot, ideally surfaced right inside Odoo (a Discuss bot, or a simple custom action/widget). Step 1: collect your sources — SOP documents, process notes, your Odoo

Knowledge articles, key config descriptions — chunk them and store embeddings (pgvector on the same Postgres your Odoo runs on keeps infra simple). Step 2: on each question, embed it, retrieve the top relevant chunks, and send them with the question: "Answer using ONLY the provided company documents. Cite which document each fact came from. If the answer isn't in the documents, say 'Not documented — ask [owner].'" That last instruction is what stops the bot from confidently inventing policy.

**Before/after:** Before, a procedural question costs the asker 5-15 minutes (find the doc or interrupt a colleague) and the colleague's time too. After, an instant cited answer, and "not documented" responses become a backlog of SOPs you actually need to write — the system tells you where your knowledge gaps are. This is the use case with the most compounding leverage: every doc you add makes it smarter. Caveat: keep the source documents current, because a RAG bot is only as accurate as what you feed it — stale SOPs produce confidently stale answers.

## Use case 8 — Anomaly flags on transactions (alert, don't block) + your deployment playbook

**The pain:** Duplicate vendor bills, a price 10x the usual, a suspicious round-number expense, an order shipping to a brand-new address for a long-standing customer — these slip through because nobody audits every transaction. By the time finance catches a duplicate payment, the money is gone.

**The build:** A scheduled action that runs nightly over recent `account.move`, `purchase.order`, or `hr.expense` records. First do the cheap deterministic checks in plain SQL/ORM — exact duplicate (same vendor + amount + date), amount far outside this vendor's historical range, missing reference. Then for borderline cases, send a compact description to the model: "Given this transaction and the vendor's typical pattern, rate fraud/error risk 0-10 and explain in one sentence. Return JSON {risk, reason}." Anything above a threshold gets an `activity` assigned to the finance lead and a chatter flag — it does **not** block or alter the transaction. The defining principle: **alert, never act** — AI flags, humans decide.

**Now ship one, not eight.** Pick the single use case with the highest frequency-times-pain (for service teams that's #2 or #3; for catalog-heavy businesses, #4 or #5) and finish it end-to-end before touching another. The repeatable build pattern: (1) prototype the prompt until JSON output is reliable on 20 real examples from your data; (2) wrap the call in one shared helper method with timeout, retry, and key management; (3) wire it to a trigger in a **staging database restored from a production backup** — never build against live; (4) run it draft/review for two weeks, logging every human correction; (5) feed corrections back as few-shot examples, then promote. Steps 1-3 are a day or two of work per use case once the shared helper exists.

**Three rules that keep this safe.** *Cost:* log token usage per call and set a monthly cap — batch jobs especially can surprise you. *Privacy:* decide consciously what customer/financial data leaves your server; for sensitive workloads consider a model with a no-training data policy or a self-hosted option, and strip PII you don't need to send. *Trust:* every AI output that touches money or customers stays draft-and-review until the correction rate is low enough that your team stops double-checking. Build the shared helper once, and each subsequent use case becomes a thin, fast, reusable addition rather than a new project from scratch.

**Pick one use case where your team wastes the most hours weekly, build it as a single server action in a staging database, measure the time saved over two weeks, then roll it to production. Want a tailored shortlist for your own Odoo instance? Reply with your Odoo version, modules in use, and the three most repetitive manual tasks your team complains about — and you'll get a prioritized deployment plan with effort estimates.**

---