Holen Ventures

AGENTS & WORKFLOWS · BUILD GUIDE

Build a Knowledge Agent You Can Trust

A Slack bot that answers questions from your documents and your live data, cites every claim, says "I don't know" instead of guessing, handles follow-up questions in the thread, and gets every answer checked by a second model before it's sent. Plus the eval harness that catches regressions when you change anything. Runs in n8n, on free tiers.

Henrik Holen

June 2026 · ~2 hours · Advanced

holenventures.com

Contents

WHAT YOU'LL BUILD, STEP BY STEP

A KNOWLEDGE AGENT YOU CAN ACTUALLY RELY ON

A knowledge agent gives your team one place to ask: it answers from your documents and your live data, instead of someone digging through folders or interrupting whoever knows. Under the hood it's **RAG** (retrieval-augmented generation): documents become searchable vectors, a question retrieves the closest chunks, a model answers from them.

This version adds the protection a plain RAG bot lacks. It refuses questions the documents can't answer, cites a source for every claim, labels live data as live, and has a second model check each answer against its sources before it's sent. A small test suite catches regressions when you change anything later.

What it does (and doesn't do)

The agent answers questions from two kinds of knowledge:

- **Documents.** Policies, internal docs, product documentation, anything static enough to embed in a vector store. Answers cite the source file in brackets after every claim.
- **Live data.** A database or API holding facts that change daily: customers, orders, tickets. The agent queries it through a locked-down tool and labels every fact it returns with a "Live data:" prefix, so you can always tell document answers from live lookups.

Questions come in through two doors: @mention it in Slack (it replies in a thread) or POST to a webhook (it returns JSON, which is how the test suite talks to it). Within a thread it follows the conversation, so after "what plan is Acme on?" you can ask "and who owns them?" and get the right account. Every interaction lands in a log table with the retrieval score, the citations, and the verdict of the faithfulness check, and that same table is the memory the follow-ups read from.

It does not answer from the model's own memory. If retrieval comes back weak and the question isn't a live-data lookup, it says so and names a human to ask. If the drafted answer contains claims the sources don't support, a judge model blocks it and flags a human instead.

Before you start

Services you'll need

SERVICE	WHAT IT DOES	FREE TIER
n8n	Runs both workflows	Yes (self-hosted) or cloud free tier
Neon (neon.com)	Postgres with pgvector: the vector store, the live-data table, the log	Yes, comfortably covers this build
OpenAI	Embeddings (text-embedding-3-small)	Pay-as-you-go, under a cent per full ingestion
OpenRouter	The agent, the intent check, and the judge	Pay-as-you-go, cents per day of heavy use
Slack	Where people ask questions	Yes
Google Drive	Optional document source (PDFs, Docs, Word files)	Yes
Python 3	The eval harness	Local, needs only `requests`

Any Postgres with the pgvector extension works in Neon's place. One database carries all three tables, so there's only one database credential to manage.

Pick a test corpus before your real one

Build against a public document set first, then swap in your own. This build used eight pages of the GitLab employee handbook, which gave it real prose across varied topics.

Choose a model that can tool-call

The agent decides when to query the live-data tool, so it needs a model tuned for tool calling. The same advice from the marketing agent build applies: pick a model that scores well on function-calling benchmarks (a current Qwen flagship such as `qwen/qwen3.7-max` is a solid default), and verify on OpenRouter that `tools` appears in its `supported_parameters`. The intent check and the judge don't tool-call, so they can be any cheap model, but running all three on one model keeps the build simple.

Set it up, step by step

Nine steps from an empty database to a Slack bot with its own test suite. About two hours end to end.

Step 1: Set up the database

Create a Neon project (Postgres 18 ships pgvector 0.8.1). Copy the pooled connection string, then run this schema with psql or the Neon SQL editor:

```
CREATE EXTENSION IF NOT EXISTS vector;

-- document_vectors is created by n8n's PGVector node on first insert.
-- Do not pre-create it.

CREATE TABLE IF NOT EXISTS crm_contacts (
  id            serial PRIMARY KEY,
  name          text NOT NULL,
  company       text NOT NULL,
  email         text NOT NULL,
  plan          text NOT NULL CHECK (plan IN ('trial', 'starter', 'business')),
  mrr           numeric NOT NULL DEFAULT 0,
  owner         text NOT NULL,
  last_contact  date,
  notes         text
);

CREATE TABLE IF NOT EXISTS qa_log (
  id          serial PRIMARY KEY,
  asked_at    timestamptz NOT NULL DEFAULT now(),
  channel     text NOT NULL,
  question    text NOT NULL,
  answer      text,
  abstained   boolean NOT NULL DEFAULT false,
  top_score   numeric,
  citations   text,
  faithful    boolean,
  judge_notes text,
  thread_ts   text
);

CREATE INDEX IF NOT EXISTS qa_log_thread_ts_idx
  ON qa_log (thread_ts) WHERE thread_ts IS NOT NULL;
```

`crm_contacts` is the stand-in live-data source: seed it with a handful of fake customers (name, company, plan, MRR, owner, a note each) so the lookup tool has something to find. In a real deployment this table is whatever system holds your changing facts, see Customising. `qa_log`

records every question the agent ever handles, which becomes your tuning data, and it doubles as the agent's conversation memory: `thread_ts` keys each row to its Slack thread (Step 6 explains how).

In n8n, create a Postgres credential from the connection string parts. Skip the SSH tunnel option (that's for databases behind bastion hosts, Neon is direct TLS) and set SSL to `require`.

Step 2: Set up the Slack bot

Go to api.slack.com/apps and create a new app. Name it something like "Knowledge Agent" and select your workspace.

1. Under **OAuth & Permissions**, add bot token scopes `app_mentions:read` and `chat:write`.
2. Install the app, copy the **Bot User OAuth Token**, and create a Slack credential in n8n with it (the plain "Slack API" type, not OAuth2).
3. Invite the bot to the channel where people will ask questions.
4. **Don't configure Event Subscriptions yet.** The request URL comes from the workflow you import in the next step, and there's a gotcha about when to paste it (Step 7).

Step 3: Import the workflows

This build is two workflows. Both are downloadable from the web version of this guide at **holenventures.com/agents-and-workflows/knowledge-agent**:

- **KA Query** (knowledge-agent.json): the agent. Two triggers, the thread memory, the retrieval and trust layer, the live-data tool, the judge, the reply routing, the log.
- **KA Ingestion** (knowledge-agent-ingestion.json): turns documents into vectors. Two source branches: a URL list and a Google Drive folder.

In n8n, **Workflows → Import from File**, once per file. Both come with credentials stripped: after import, open each node that shows a credential warning and bind your own (Postgres everywhere, OpenAI on the embeddings nodes, OpenRouter on the three model nodes, Slack on the trigger and reply nodes, Google Drive on the Drive nodes if you use that branch).

Step 4: Configure ingestion

The ingestion workflow runs on demand, whenever the corpus or the chunking settings change. It has two branches off one manual trigger; use either or both.

The URL branch

The **Corpus File List** code node holds your document list: one URL plus one `source` label per document. The label becomes the citation, so keep it short and recognisable. Swap in your own markdown or plain-text URLs.

The Google Drive branch

Create a Google Drive credential in n8n (on n8n Cloud it's a click-through OAuth, no Google Cloud console needed). Then open **List Drive Folder** and set the folder ID from your folder's URL. Everything in that folder gets ingested:

- **Native Google Docs** are exported as markdown on download, the cleanest of the three.
- **PDFs** are routed through n8n's Extract From File node, then treated as text.
- **Word, EPUB, CSV and plain text** go through the loader's binary mode with format auto-detection.

> ● THE PDF PATH THAT KILLS THE WHOLE RUN
>
> The vector store's own binary loader cannot parse PDFs on n8n Cloud. It fails with `DOMMatrix is not defined` (its PDF parser needs a browser API the server runtime doesn't have), and because a failing node aborts the entire execution, one PDF in the folder took down the whole corpus reload, including the unrelated URL branch. The imported workflow already contains the two fixes: PDFs detour through Extract From File into the text path, and the Drive insert nodes run with "On Error: Continue" so an unparseable file is skipped instead of failing the whole run. If you rebuild from scratch, carry both fixes over.

Chunking

The splitter nodes are set to **800 characters per chunk with 100 overlap**, markdown-aware so cuts prefer heading boundaries. These two numbers are the main retrieval-quality dial in the build, and Step 9 demonstrates what happens when they're wrong. Leave them alone until the eval harness exists.

> ● TWO INGESTION GOTCHAS
>
> **Don't pre-create the vectors table.** The PGVector node creates `document_vectors` itself on first insert, with columns `id`, `text`, `metadata`, `embedding`.
>
> **Citations are built from the source metadata.** Each loader stamps chunks with a `source` value. A corpus ingested without it produces answers that cite nothing.

Step 5: Run ingestion and verify by row count

Execute the ingestion workflow, then check the database, not the n8n execution status:

```
SELECT count(*) FROM document_vectors;

SELECT metadata->>'source' AS source, count(*)
FROM document_vectors
GROUP BY 1 ORDER BY 2 DESC;
```

You want a sensible chunk count (the eight handbook files here, roughly 430 KB, made 875 chunks) and every source present. The second query is also how you confirm a Drive file actually made it in.

● INGESTION CAN CRASH THE N8N INSTANCE, AFTER THE WORK IS DONE

On a small n8n Cloud instance, embedding-heavy runs repeatedly finished their inserts and then knocked the instance over: the execution shows status "crashed", the instance restarts, and it's back within a minute or two. Reducing the insert nodes' embedding batch size from 200 to 50 (already set in the imported workflow) helps the run complete fast, but the restart can still happen. So verify ingestion by row count in the database. An execution marked "crashed" with all rows present did its job.

Re-ingestion is truncate-and-reload: `TRUNCATE document_vectors;`, then run the workflow again. It's simple, correct, and fine up to thousands of documents, and incremental updates are a straightforward extension (see Going further).

Step 6: Configure the agent

The query workflow is where the trust layer lives. Bind the credentials, then go through these settings one at a time.

Conversation memory

Memory has to happen before retrieval, not after. A follow-up like "and who owns them?" embeds to nothing in particular, scores low against every chunk, and the threshold gate refuses it before any model runs. So the workflow rewrites follow-ups into standalone questions first, and the trust layer treats the rewritten question like any other.

The memory itself is the log table. Each Slack thread is a session: the thread timestamp is the key, a Postgres node reads the thread's last four turns back from `qa_log`, and a small LLM call (temperature 0) rewrites the latest message as a standalone question, so "and who owns them?" becomes "Who owns the Acme account?". A first message has no history, so it skips the rewrite entirely. The condensed question is what goes through retrieval, the gate, the agent, and the log, which means the next follow-up builds on resolved questions rather than ambiguous ones.

Two design points worth keeping if you rebuild this:

- **Durable beats in-process.** n8n has a Simple Memory node that holds history in instance memory, and Step 5's gotcha already showed how casually this instance restarts. Reading memory back from the database survives restarts and keeps the workflow itself stateless.
- **Threads isolate users.** Several people asking at once is the normal case for a team bot, not the edge case. Keying memory on the thread timestamp means concurrent conversations can't bleed into each other; two threads are just two keys in the same table.

The webhook door gets the same behaviour by passing a `threadTs` field in the body. Leave it out and the door is stateless, which keeps the eval harness's single-turn cases independent of each other.

The threshold gate

Retrieval returns the top 5 chunks with a score per chunk, and a code node decides deterministically whether the question is answerable before any LLM gets involved.

> ● THE SCORE IS A DISTANCE, NOT A SIMILARITY
>
> The PGVector node's `score` is a cosine distance: **lower means more similar**. Read it as a similarity and the abstention logic fires backwards, confidently answering nonsense and refusing good questions. The gate converts with `1 - score` so the threshold reads naturally. On the handbook corpus, answerable questions landed around 0.65 to 0.81 similarity and nonsense around 0.21 to 0.34, so the 0.5 threshold has comfortable margin on both sides. Check your own corpus's spread (ask one real question and one absurd one, look at `top_score` in the log) before trusting the default.

The intent check

Questions for the live-data tool ("what plan is Acme on?") score low against the document corpus, because the entities they mention aren't in the documents. Without a fix, the threshold gate kills the questions the tool exists to answer, before the agent ever runs. An early version of this build shipped with that bug, and only end-to-end testing caught it.

The fix sits on the abstain branch: a one-question LLM call (temperature 0) that decides whether the question is about live data, and routes it to the agent anyway if so. The system prompt is where you describe what your live source holds, currently:

```
Decide whether the question asks about a specific customer, company account, contact, or CRM
data (plans, MRR, account owner, trials, renewal, contact details). Reply with exactly YES or
NO. Nothing else.
```

Swap the noun list for your domain: "orders, shipments, stock levels" or "tickets, SLAs, assignees". Both ways this check can fail end in a refusal, never an invented answer: a wrong NO gets the abstention message, a wrong YES makes the agent run, find nothing, and say the documents don't cover it.

The lookup tool

The agent's one tool, `live_data_lookup`, is whatever connects to your live system. This build queries a Postgres table, but the slot takes any n8n tool node: an HTTP Request tool against an internal API, an MCP Client node pointed at a server that exposes the data, or a native integration node for your CRM or ticketing system. Here it's a parameterised SQL search where the model supplies only the search term and the query is fixed, so the model cannot write arbitrary SQL:

```
SELECT name, company, email, plan, mrr, owner, last_contact, notes
FROM crm_contacts
WHERE name ILIKE '%' || $1 || '%' OR company ILIKE '%' || $1 || '%'
LIMIT 5;
```

Whatever the integration, keep the shape: **a locked-down lookup, a model-supplied parameter, and "Live data:" labels on everything it returns**.

The agent prompt

The Answer Agent's system message enforces four rules:

```
Today's date is {{ $now.toFormat('yyyy-MM-dd') }} ({{ $now.toFormat('cccc') }}).

You answer questions using ONLY the document excerpts provided in the user message. Rules:
1. After every factual claim, cite its source file in square brackets, e.g. [values/_index.md].
2. If the excerpts answer the question, even indirectly, answer from them. Refuse only when
they genuinely don't: if the question is not about a specific customer, contact, or account,
reply "The documents don't cover that.", plus one short cited sentence on any related facts the
excerpts do contain. Never include that phrase when you answer using the live_data_lookup tool.
3. Use the live_data_lookup tool ONLY when the question asks about a specific customer,
contact, or account. Prefix every fact that came from the tool with "Live data:". Never present
tool data as document content.
4. Be concise. No preamble.
```

Rule 2's exact phrase matters: the judge exempts it by string match, so if you reword it, reword it in both places. The date line exists because models have no clock (the marketing agent build hit the same problem).

The faithfulness judge

After the agent answers, a second, independent LLM call (temperature 0, fresh context) grades the answer against the same excerpts:

```
You are grading whether an answer is faithful to its source excerpts. Every claim in the answer
must be supported by the excerpts, except lines prefixed "Live data:" (those came from a
database tool and are exempt) and the exact phrase "The documents don't cover that." Reply on
the first line with exactly FAITHFUL or UNFAITHFUL, then one sentence of reasoning.
```

Faithful means supported by the provided sources, not "true in the world". An answer the model happened to know from training data still fails, and should: once answers stop tracing to your documents, you can't tell where anything came from, and the citations stop meaning anything. A failed verdict routes to an escalation message, the draft is withheld, and the log keeps the evidence. With a tuned threshold this fires rarely, which is why the check has to be automatic; nobody keeps reviewing answers that are almost always fine.

● YOUR EDITS DON'T REACH PRODUCTION UNTIL YOU PUBLISH

Editing a node in the n8n editor changes the draft. The production webhook keeps running the previously published version until you publish again. A prompt fix that "didn't work" usually worked fine, in a version production isn't running. Check which version is live before debugging the prompt.

Step 7: Test the doors

Activate the workflow, then test the webhook door first, it has no Slack dependencies:

```
curl -X POST https://YOUR-INSTANCE/webhook/ka-eval \
  -H 'Content-Type: application/json' \
  -d '{"question": "YOUR IN-CORPUS QUESTION"}'
```

Run four questions and check each behaviour:

1. A question your documents answer → an answer with bracketed citations and `"faithful": true`.
2. An absurd question → the abstention message, `"abstained": true`, low `topScore`.
3. A live-data question ("what plan is [seeded customer] on?") → a "Live data:" answer with no document citations.
4. A mixed question → cited document content and labelled live data, cleanly separated.
5. A follow-up → re-send the live-data question with `"threadTs": "test-1"` added to the body, then POST `{"question": "and who owns that account?", "threadTs": "test-1"}`. The response's `question` field shows the standalone rewrite the agent actually answered.

Then wire Slack: open the Slack Trigger node, copy its Production webhook URL, paste it into your Slack app's Event Subscriptions request URL, subscribe to `app_mention`, and **press Save**. @mention the bot and you should get a threaded, cited reply. @mention it again inside that thread with a follow-up and it answers against the conversation.

● THE SLACK URL DIES EVERY TIME THE WORKFLOW IS REBUILT

Re-importing or programmatically replacing the workflow regenerates the Slack trigger's webhook ID, the old URL goes dead, and mentions silently stop arriving (the webhook door keeps working, because its URL is path-based). If mentions stop: copy the trigger's current URL and re-paste it into Slack. If you edit workflows through an AI assistant over n8n's MCP API, batch every structural edit and re-point Slack once at the end. And remember Slack's settings page does nothing until you press Save, an unsaved Events URL looks the same as a broken workflow.

One behaviour to know about: **high retrieval similarity does not mean the answer is present.** In this build, "what are the company values?" scored 0.785 similarity but retrieved chunks that discuss the values without listing them, and the agent correctly said the excerpts don't cover it. Rephrased as "list the six core values by name", retrieval found the enumeration chunk and the agent answered with the full list. The threshold catches off-topic questions; the agent's excerpts-only rule catches on-topic questions the retrieved chunks happen not to answer. You need both layers.

Step 8: Build the eval harness

A knowledge agent is non-deterministic, so the only way to know a change didn't break it is a fixed set of questions with known correct behaviour, run before and after every change.

The golden set

Create `golden-set.json` with 15 to 20 cases across four kinds:

```
{
  "cases": [
    { "kind": "answerable",
      "question": "A question your documents answer",
      "expected": "The gold answer, taken from the documents",
      "source": ["the-file-it-should-cite.md"] },
    { "kind": "abstain",
      "question": "Which conference room has the espresso machine?",
      "expected": null, "source": null },
    { "kind": "live",
      "question": "What plan is [seeded customer] on?",
      "expected": "The answer from your seed data",
      "source": null },
    { "kind": "followup",
      "turns": ["What plan is [seeded customer] on?",
                "and when does their trial end?"],
      "expected": "The trial end date from your seed data",
      "source": null }
  ]
}
```

Write the answerable cases from your actual corpus, never from memory. The abstain cases should be plausible questions your documents don't cover. The live cases come from your seed data. The followup cases send their turns through one thread key and grade only the final answer; the second turn only passes if the rewrite resolved it correctly.

The runner

The runner is about fifty lines of Python and needs no framework. It POSTs each question to the webhook door, grades answerable and live cases with an LLM judge, checks abstain cases by the `abstained` flag, verifies citations against the expected source, and exits non-zero on any failure. The setup and the two calls:

```
#!/usr/bin/env python3
import json, os, sys, time
from pathlib import Path
import requests

WEBHOOK_URL = os.environ["KA_WEBHOOK_URL"]
KEY = os.environ["OPENROUTER_API_KEY"]
JUDGE_MODEL = "qwen/qwen3.7-max"
GOLDEN = Path(__file__).resolve().parent / "golden-set.json"

def ask(question, thread_ts=None):
    payload = {"question": question}
    if thread_ts:
        payload["threadTs"] = thread_ts
    r = requests.post(WEBHOOK_URL, json=payload, timeout=180)
    r.raise_for_status()
    return r.json()

def judge(question, expected, answer):
    prompt = (
        "You are grading a knowledge agent's answer.\n"
        f"Question: {question}\nGold answer: {expected}\nAgent answer: {answer}\n"
        "Does the agent answer convey the same key facts as the gold answer? "
        "Grade on the COMPLETE answer: if the key facts appear anywhere in it, "
        "it passes, even if the answer also contains a caveat. Ignore phrasing "
        "differences and extra correct detail. An omitted year counts as "
        "matching when the day and month agree. "
        "First line: exactly PASS or FAIL. Second line: one sentence why."
    )
    r = requests.post("https://openrouter.ai/api/v1/chat/completions",
        headers={"Authorization": f"Bearer {KEY}"},
        json={"model": JUDGE_MODEL, "temperature": 0,
              "messages": [{"role": "user", "content": prompt}]},
        timeout=180)
    r.raise_for_status()
    verdict = r.json()["choices"][0]["message"]["content"].strip()
    return verdict.splitlines()[0].strip().upper().startswith("PASS"), verdict
```

And the grading logic plus the loop:

```
def grade(case):
    if case["kind"] == "followup":
        # Fresh thread key per run so old history can't leak in.
        thread_ts = f"eval-{int(time.time())}"
        for turn in case["turns"]:
            result = ask(turn, thread_ts)
            time.sleep(2)  # let the log commit before the next turn reads it
        question = case["turns"][-1]
    else:
        result = ask(case["question"])
        question = case["question"]
    if case["kind"] == "abstain":
        if result.get("abstained") is True:
            return True, "abstained correctly"
        if "don't cover that" in (result.get("answer") or ""):
            return True, "agent-level refusal"
        return False, "answered instead of abstaining"
    if result.get("abstained"):
        return False, "abstained on an answerable question"
    if result.get("faithful") is False:
        return False, "failed the workflow's own faithfulness check"
    ok, note = judge(question, case["expected"], result.get("answer", ""))
    if ok and case["kind"] in ("answerable", "followup") and case.get("source"):
        sources = case["source"] if isinstance(case["source"], list) else [case["source"]]
        cited = " ".join(result.get("citations", [])).lower()
        if not any(s.lower() in cited for s in sources):
            return False, f"wrong or missing citation, wanted one of {sources}"
    return ok, note.splitlines()[-1][:140]

def main():
    cases = json.loads(GOLDEN.read_text())["cases"]
    failures = 0
    for case in cases:
        try:
            ok, note = grade(case)
        except requests.RequestException as exc:
            ok, note = False, f"request failed: {exc}"
        label = case.get("question") or " / ".join(case["turns"])
        print(f"[{'PASS' if ok else 'FAIL'}] ({case['kind']}) {label[:58]} | {note}")
        failures += 0 if ok else 1
        time.sleep(1)
    print(f"\n{len(cases) - failures}/{len(cases)} passed")
    sys.exit(1 if failures else 0)

if __name__ == "__main__":
    main()
```

Run it red first

Before trusting a green run, prove the harness can fail: temporarily change one abstain case's question to something the documents answer, run it, and confirm a FAIL and a non-zero exit. If it can't fail, it isn't testing anything.

Expect to calibrate the golden set, not the agent

The first full run here scored 13 of 16, and all three failures were in the test set, not the system: a fact that lives in several corpus files failed a single-source citation expectation (accept a list of valid sources), a gold answer demanded a detail retrieval hadn't surfaced (trim gold answers to what one retrieval supports), and the judge failed "June 22" for omitting the year (judges need explicit leniency rules). Three things to get right:

- **Pin the judge's temperature to 0.** Without it, the same answer passed one run and failed the next. The harness must be the most deterministic thing in the system.
- **Accept either refusal layer for abstain cases**: the threshold abstention or the agent's "documents don't cover that". Both are correct refusals with different causes.
- **Read failure notes before blaming your changes.** Mid-build, the n8n instance had a brief outage and the harness reported sixteen request failures. An outage and a regression look identical from the outside, which is why the runner reports per-case errors instead of crashing.

Step 9: Break it on purpose

Once the harness is green, open the ingestion workflow's splitter nodes and set chunk size to 150 with 0 overlap. These are deliberately poor values which will make the workflow break. Truncate the vectors table, re-ingest (the corpus splits into five times as many fragments), and run the evals.

This build went from 16/16 to **10/16**, and the failures weren't evenly spread:

- **Six of eight document questions failed**: refusals, partial answers, and one fact truncated mid-sentence because a book title straddled a chunk boundary with no overlap to stitch it.
- **All five abstain cases still passed.** Refusing correctly doesn't depend on chunk quality.
- **All three live-data cases still passed.** The lookup tool never touches the vector store.

The way it fails tells you why. Document answers got worse, while refusals and tool answers stayed the same, so the vector store is the problem. That's why the golden set spans categories instead of being sixteen variations of one question. Restore 800/100, truncate, re-ingest, and confirm green before moving on.

Customising for your deployment

- **Your documents.** Point the Drive branch at the folder where your documents already live, or swap the URL list. You can also add a trigger to the Drive branch, so ingestion fires when a file in the folder changes instead of manually.
- **Your live source.** Replace the lookup tool's SQL with your table, swap the Postgres tool for an HTTP Request tool against an internal API, or use an MCP Client node if the system has an MCP server. Keep the shape: locked-down lookup, model-supplied parameter, "Live data:" labels. Then update the intent-check noun list and the agent prompt's rule 3 to match.
- **The refusal wording.** The abstention message ("I don't have that in the documents. Ask …") and the fallback contact are plain fields in the Abstain Reply node. Name a real person; "ask a human"

answers are only useful if they say which human.

- **The threshold.** 0.5 similarity worked on an 875-chunk prose corpus. Sparser or more technical corpora shift the score distributions, so check `top_score` in the log for a week and move the line to where your real questions separate from nonsense.
- **Scanned PDFs don't work.** The PDF extractor reads embedded text; a scanned page is an image and yields nothing. OCR is its own build, put it in front of ingestion if you need it.

Going further

- **An email door.** The two triggers converge on one normalisation node, and a third door plugs in at the same place: an email trigger (IMAP or Gmail) that pre-shapes sender and subject the way the Slack door does, with replies routed back by email.
- **Incremental re-ingestion.** Truncate-and-reload is fine until the corpus is large. The `source` metadata enables the upgrade: delete where source matches, re-ingest only the changed file.
- **Graft the eval pattern onto your other agents.** The golden set, the red-first check, and the judge-with-leniency-rules transfer to any non-deterministic workflow, and give you an answer to "did my change break anything?" instead of a guess.
- **Watch the log.** `qa_log` accumulates abstention rates, score distributions, and judge verdicts. A rising abstention rate means the corpus has drifted from what people actually ask, which is the signal to expand it.

Build something with this?

This is one of a series of build guides. If you want help setting up something similar, or just want the weekly letter on building in the AI era, here's where to go.

MORE BUILDS	holenventures.com/agents-and-workflows
THE LETTER	holenventures.substack.com
WORK WITH ME	holenventures.com/work-with-me
EMAIL	henrik@holenventures.com