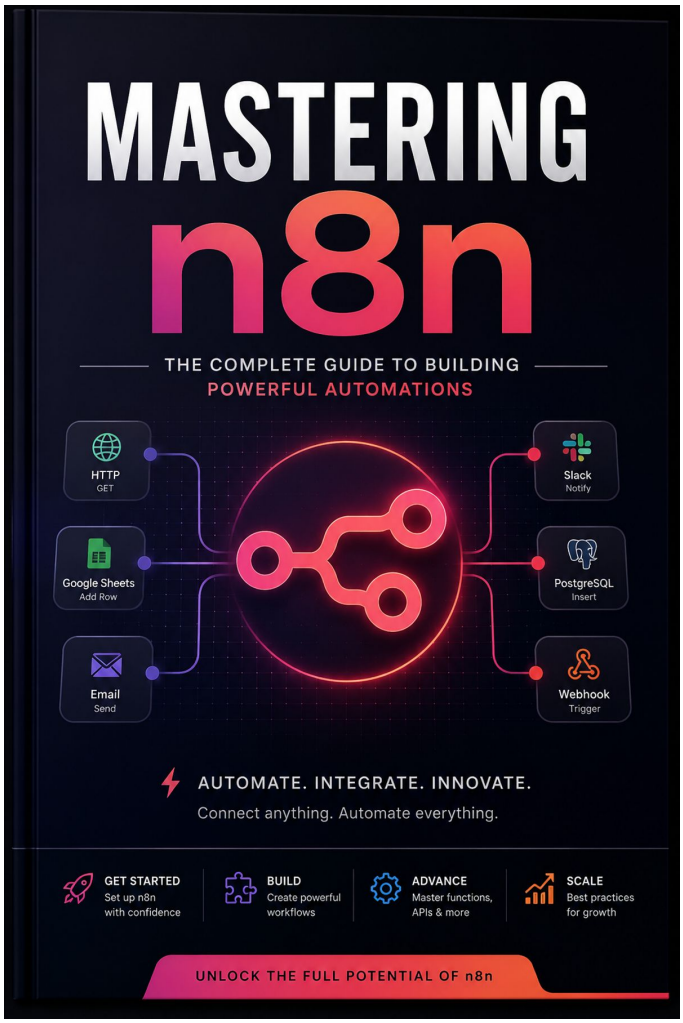

Mastering n8n

If Directus is the first thing I stand up on a new project, n8n is the second. It's the workflow automation tool I reach for whenever something needs to talk to something else — webhooks, scheduled syncs, API orchestration, and increasingly, AI agents that actually do work instead of just chatting.

The pitch is simple: a visual canvas where each node is a step, the data flows down the chain, and you can drop into real JavaScript any time the visual builder runs out of road. Self-host it on a single box, point it at Postgres and Redis, and you've got a durable automation engine you fully own — no per-task pricing, no vendor watching your data go by.

This is the guide I'd hand a developer who's tired of gluing services together by hand: how to self-host n8n properly with Docker and queue mode, how the data model and expressions actually work, how to handle errors like you mean it, how to build real AI agents with the LangChain nodes, and how to wire it up to Directus so the two cover each other's weak spots.

This is a living document and will be updated as n8n updates.

Roger Stringer · rogerstringer.com

June 19, 2026

Contents

What n8n Actually Is

Most people meet n8n as "open-source Zapier," and like most one-line pitches, it's true enough to be useful and wrong enough to be misleading. Zapier is a hosted product you rent. n8n is a workflow engine you *run* — a Node.js app you self-host, point at your own database, and own end to end.

The model is dead simple once it clicks. A **workflow** is a graph of **nodes**. One node is a **trigger** (a webhook fires, a schedule ticks, a row changes). Every node after it is a step that receives data, does something, and passes data on. The data moving between nodes is a list of **items** — and that item model is the single most important thing to understand about n8n, so it gets its own chapter later. The canvas is visual, but the moment the visual builder runs out of road you drop into a **Code node** and write real JavaScript (or Python). That escape hatch is why I reach for it over the no-code tools: I never hit a wall I can't code my way through.

Here's where it actually earns its place in my stack. n8n is the glue layer. Directus owns my data and gives me an API; n8n is what reaches *out* — calling third-party APIs, reacting to webhooks, running nightly syncs, orchestrating five services into one coherent process, and lately, running AI agents that call tools and get real work done. It ships with 400+ integration nodes so I'm not writing yet another OAuth dance for the Nth SaaS API, but the HTTP Request node means I can talk to anything with an endpoint regardless.

A few things worth knowing up front:

- **It's fair-code, not open source.** n8n ships under the Sustainable Use License. You can self-host and modify it freely for your own internal business use, and — importantly — you're now explicitly allowed to do *paid consulting* building n8n workflows for clients. What you can't do is rebrand it and resell it as your own hosted SaaS. Files with `.ee.` in the name are Enterprise-licensed and not covered. For an indie dev or a team running their own automations, you're completely fine. The terms have shifted over the years, so check the current license page rather than trusting any blog post (including this one) on the fine print.
- **Self-host or cloud, your choice.** There's an n8n Cloud if you'd rather not run it, but this guide is about self-hosting — that's the whole point for me. You own the data, there's no per-task metering, and a $5–10 VPS handles a surprising amount.
- **It's stateful and durable.** Unlike a stateless function, n8n persists every execution to a database. You can open any run, see exactly what data each node received and emitted, and replay it. That observability is worth a lot when something breaks at 2am.
- **AI is now first-class.** Since the 2.0 line, n8n ships native LangChain-based nodes — an AI Agent node, chat models, memory, vector stores, the lot. It's gone from "automation tool" to "the orchestration layer I build agents on." We'll spend two full chapters there.

The rest of this guide is about running it like you mean it: self-hosting on Docker, scaling with queue mode, hardening for production, actually understanding the data model and expressions, handling errors properly, building AI agents, and wiring it up to Directus. If you've read Mastering Directus (https://rogerstringer.com/guides/mastering-directus), this is the other half of how I build — the two tools cover each other's blind spots, and there's a whole chapter on running them together.

Self-Hosting with Docker

n8n self-hosts beautifully with Docker, and for a single-instance setup you can be running in about five minutes. The one decision that trips people up later is the database, so let's get that right from the start.

Don't use SQLite in production

Out of the box, n8n will happily run on SQLite. It's fine for kicking the tires on your laptop. It is *not* fine for production — execution data grows fast, SQLite locks under concurrency, and you'll regret it the first time you want to scale. Use **Postgres** from day one. If you're already running Directus, you may even be able to share the Postgres server (separate database, same instance).

A real single-instance setup

Here's a `docker-compose.yml` I'd actually deploy — n8n plus its own Postgres:

```
services:
  postgres:
    image: postgres:16
    restart: always
    environment:
      POSTGRES_USER: n8n
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: n8n
    volumes:
      - n8n_pgdata:/var/lib/postgresql/data
    healthcheck:
      test: ['CMD-SHELL', 'pg_isready -U n8n']
      interval: 10s
      timeout: 5s
      retries: 5

  n8n:
    image: docker.n8n.io/n8nio/n8n:latest
    restart: always
    depends_on:
      postgres:
        condition: service_healthy
    ports:
      - '5678:5678'
    environment:
      DB_TYPE: postgresdb
      DB_POSTGRESDB_HOST: postgres
      DB_POSTGRESDB_DATABASE: n8n
      DB_POSTGRESDB_USER: n8n
      DB_POSTGRESDB_PASSWORD: ${POSTGRES_PASSWORD}
      N8N_HOST: n8n.example.com
      N8N_PROTOCOL: https
      WEBHOOK_URL: https://n8n.example.com/
      N8N_ENCRYPTION_KEY: ${N8N_ENCRYPTION_KEY}
      GENERIC_TIMEZONE: America/Vancouver
      N8N_RUNNERS_ENABLED: 'true'
    volumes:
      - n8n_data:/home/node/.n8n

volumes:
  n8n_pgdata:
  n8n_data:
```

Drop a `.env` next to it:

```
POSTGRES_PASSWORD=$(openssl rand -hex 24)
N8N_ENCRYPTION_KEY=$(openssl rand -hex 24)
```

Then `docker compose up -d` and you're live on port 5678.

The settings that matter

A few environment variables are non-negotiable and easy to miss:

- **`N8N_ENCRYPTION_KEY`** — this encrypts all your stored credentials. **Set it explicitly and back it up.** If n8n generates one for you and you later lose that volume, every saved credential becomes undecryptable garbage. This is the single most common self-hosting disaster. Treat this key like a database password.
- **`WEBHOOK_URL`** — n8n needs to know its own public URL to generate correct webhook addresses. If you're behind a reverse proxy or tunnel, set this or your webhooks will hand out `localhost` URLs.
- **`N8N_HOST`** / **`N8N_PROTOCOL`** — used for building URLs and the editor. Match your real domain.
- **`GENERIC_TIMEZONE`** — Schedule triggers use this. If you don't set it, your "every day at 9am" cron runs in UTC and you spend a confused afternoon.
- **`N8N_RUNNERS_ENABLED`** — turns on task runners, which execute Code nodes in a separate process. This is the recommended default going forward; turn it on.

Coolify makes this even easier

If you've read The Boring Stack (https://rogerstringer.com/guides/the-boring-stack), you know I lean on Coolify for single-VPS deploys. n8n is a one-click service in Coolify — it provisions Postgres, wires up the env, and handles TLS via the built-in proxy. I still set `N8N_ENCRYPTION_KEY` myself and write it down somewhere safe. The Compose above is what's happening under the hood either way; understanding it means you can debug it when Coolify's abstraction leaks.

Updating

Updating is `docker compose pull && docker compose up -d`. n8n runs database migrations automatically on boot. Two rules: **pin a major version** rather than floating on `:latest` so a breaking change doesn't land on you unannounced, and **back up your Postgres database and your encryption key before any upgrade**. Read the release notes for major bumps — the 1.x to 2.x transition, for instance, changed real behavior.

Queue Mode: Scaling with Workers and Redis

The single-container setup from the last chapter runs every workflow inside the main process. That's the **regular** execution mode, and it's fine until it isn't. One heavy workflow can starve the editor. A spike of webhooks can pile up. A crash takes everything down at once. When you outgrow it, the answer is **queue mode**.

How queue mode works

Queue mode splits n8n into three roles connected by Redis:

- **The main instance** runs the UI, registers triggers, and receives webhooks. It does *not* execute workflows itself. When a workflow needs to run, it pushes a job onto Redis.
- **Redis** is the message broker — the queue itself.
- **Worker processes** watch Redis, pick up jobs, execute the workflow, and report results back.

The payoff: you add workers to add throughput, and because each worker is its own process, a slow or failing workflow on one worker doesn't block the others or the editor. This is the same shape as any real job queue — and if you've read Background Jobs & Queues (https://rogerstringer.com/guides/background-jobs-and-queues), it'll feel familiar, because it's the same idea applied to n8n itself.

The setup

You add a Redis service, flip `EXECUTIONS_MODE` to `queue`, and run one or more containers in **worker** mode. The main and worker containers share the *same* image, the *same* Postgres, the *same* Redis, and critically the *same* `N8N_ENCRYPTION_KEY` — otherwise workers can't decrypt credentials.

```
services:
  redis:
    image: redis:7-alpine
    restart: always
    volumes:
      - n8n_redis:/data

  n8n-main:
    image: docker.n8n.io/n8nio/n8n:latest
    restart: always
    ports:
      - '5678:5678'
    environment:
      EXECUTIONS_MODE: queue
      QUEUE_BULL_REDIS_HOST: redis
      QUEUE_HEALTH_CHECK_ACTIVE: 'true'
      DB_TYPE: postgresdb
      DB_POSTGRESDB_HOST: postgres
      DB_POSTGRESDB_PASSWORD: ${POSTGRES_PASSWORD}
      N8N_ENCRYPTION_KEY: ${N8N_ENCRYPTION_KEY}
      WEBHOOK_URL: https://n8n.example.com/
    depends_on: [postgres, redis]

  n8n-worker:
    image: docker.n8n.io/n8nio/n8n:latest
    restart: always
    command: worker
    environment:
      EXECUTIONS_MODE: queue
      QUEUE_BULL_REDIS_HOST: redis
      DB_TYPE: postgresdb
      DB_POSTGRESDB_HOST: postgres
      DB_POSTGRESDB_PASSWORD: ${POSTGRES_PASSWORD}
      N8N_ENCRYPTION_KEY: ${N8N_ENCRYPTION_KEY}
    depends_on: [postgres, redis]
    deploy:
      replicas: 2
```

That `command: worker` is the whole trick — same image, started in worker mode. Scale workers with `docker compose up -d --scale n8n-worker=4` or the `replicas` key.

Webhooks at scale

Under heavy inbound load you can also run dedicated **webhook processor** instances (`command: webhook`) so that receiving webhooks never competes with the editor. For most setups you won't need this until you're well into production volume — but it's there, and it's the same pattern: another container, same shared backing services.

How many workers, and how big?

Start with two workers and watch. The lever inside each worker is **concurrency** (`N8N_CONCURRENCY_PRODUCTION_LIMIT`) — how many executions one worker runs in parallel. More concurrency per worker uses more RAM; more workers uses more everything but isolates failures better. My rough rule: bump concurrency first because it's cheap, add worker containers when a single worker's CPU or memory is the ceiling. n8n itself is not especially hungry — most of your memory goes to the *payloads* moving through workflows, so workflows that haul large files or big arrays are what actually drive sizing.

Do you even need this?

Be honest about scale. A solo founder running a few dozen automations does **not** need queue mode — the single container is simpler and one less thing to break. Reach for queue mode when you're running enough volume that executions queue up, when one workflow's load is hurting the editor, or when you want the resilience of isolated workers. Don't cargo-cult the architecture; earn it.

Hardening for Production

Getting n8n running is easy. Running it so you can sleep at night takes a handful of deliberate choices. None of this is exotic — it's the same hygiene any self-hosted service needs — but n8n has a few sharp edges worth calling out.

Put it behind a reverse proxy with TLS

Never expose n8n's port directly. Terminate TLS at a reverse proxy — Caddy, Traefik, nginx, or Coolify's built-in proxy — and forward to n8n internally. Caddy is the least fuss:

```
n8n.example.com {
    reverse_proxy n8n:5678
}
```

That one block gets you automatic Let's Encrypt certs and renewal. Make sure `N8N_PROTOCOL=https` and `WEBHOOK_URL` use the public HTTPS address so generated webhook URLs are correct.

Lock down the editor

The n8n editor is the keys to the kingdom — it holds every credential and can call anything. Owner accounts are protected by login, but a few extra layers matter:

- **Turn on user management** and use strong, unique passwords; enable MFA where available.
- Consider **not exposing the editor to the public internet at all**. If only you and your team use it, put the editor behind a VPN or IP allow-list at the proxy, and expose *only* the `/webhook/` paths publicly. Your webhooks need to be reachable; your editor does not.
- Keep the instance **updated** — security fixes land in releases, and running months behind is a real risk for an internet-facing app.

Protect your webhooks

A public webhook URL is an open door unless you guard it. n8n's webhook node supports **header auth** and **basic auth** — use them. For incoming webhooks from services that sign their payloads (Stripe, GitHub, etc.), verify the signature in a Code node before doing anything else. Treat every inbound webhook payload as hostile until you've validated it.

Back up the right things

Three things must be backed up, and people routinely forget the second:

1. **The Postgres database** — your workflows, credentials (encrypted), and execution history.
2. **The `N8N_ENCRYPTION_KEY`** — without it, the encrypted credentials in that database are useless. Back it up *separately* from the database, in a password manager or secrets store. I cannot overstate how many people learn this the hard way.
3. **Your workflow definitions** — ideally in git, which we'll cover in the source-control chapter. A database backup is your safety net; git is your version history.

Tame execution data

Every execution gets written to the database, and by default n8n keeps a lot of it. Left alone, your Postgres database balloons. Prune it:

```
EXECUTIONS_DATA_PRUNE=true
EXECUTIONS_DATA_MAX_AGE=336      # hours; 336 = 14 days
EXECUTIONS_DATA_PRUNE_MAX_COUNT=10000
```

Tune retention to how much forensic history you actually need. You can also set workflows to **save only error executions** in their settings, which keeps the noise down while preserving the runs you'll actually want to inspect. For successful high-frequency workflows, saving every run is usually just expensive logging.

Resource limits and timeouts

Set `EXECUTIONS_TIMEOUT` so a runaway workflow can't hang forever, and put memory limits on your containers so one bad payload can't take down the host. If you're in queue mode, this isolation is already better — a worker can die and restart without touching the editor. Pair that with `restart: always` and Docker will bring crashed containers back on its own.

The Data Model: Items and the Chain

If you only deeply understand one thing about n8n, make it this: **data flows between nodes as an array of items.** Almost every confusing moment you'll have — "why did my node run five times?", "why is my expression undefined?", "why did I get one result when I expected ten?" — traces back to not having internalized the item model. So let's nail it.

An item is `{ json, binary }`

Each item is an object with two parts: a `json` property (the structured data) and an optional `binary` property (files — images, PDFs, attachments). When people say "the data," they usually mean `item.json`. A node receives an array of items and outputs an array of items.

```
[
  { "json": { "id": 1, "name": "Ada" } },
  { "json": { "id": 2, "name": "Linus" } }
]
```

That's two items. This detail matters enormously because of the next point.

Nodes run once per item

Most nodes execute **once for every item they receive**. Hand an HTTP Request node 10 items and it fires 10 requests — one per item — automatically. You don't write a loop; the item array *is* the loop. This is the single most powerful and most surprising thing about n8n. Want to process 200 records? Get them into 200 items and every downstream node just handles them.

The corollary: if a node runs more times than you expected, it's because it received more items than you thought. If it runs *once* when you wanted many, you probably have a single item containing an array, and you need to **split it out** first.

Splitting and aggregating

Two nodes you'll use constantly:

- **Split Out** — takes one item holding an array field and turns it into one item per array element. The classic case: an API returns `{ "results": [...] }` as a single item, and you Split Out on `results` to get one item per result.
- **Aggregate** — the reverse. Collapses many items back into one (e.g. to build a single payload, or to send one summary email instead of 50).

Getting fluent at moving between "many items" and "one item with an array" is most of the skill.

Referencing data: the expression basics

Inside any field you can click into **expression mode** and pull values from the item with `{{ }}` syntax:

- `{{ $json.name }}` — a field from the current item.
- `{{ $json["first name"] }}` — bracket syntax when a key has spaces.
- `{{ $('NodeName').item.json.id }}` — reach back to a *specific earlier node's* output. This is how

you grab data from three steps ago, not just the immediately previous node.
- `{{ $now }}`, `{{ $today }}` — built-in date helpers.
- `{{ $env.SOME_VAR }}` — environment variables (handy for keeping config out of workflows).

The `$('NodeName')` reference is the one that unlocks real workflows. Data doesn't only flow straight down — you can pull from any earlier node by name, which is how you correlate, say, the original webhook payload with the result of an API call you made two steps later.

Pinned data and the manual run

When you're building, **execute the workflow** (or a single node) and n8n shows you the exact items at every step in the side panel. You can **pin** a node's output so you're not hammering a live API on every test run. This tight inspect-and-iterate loop — see real data, adjust the expression, run again — is how you actually build in n8n. Don't write a whole workflow blind and hope; build it one node at a time, watching the items flow.

Internalize "array of items, once per item, split and aggregate to reshape, reference by node name" and n8n stops being mysterious. Everything else is just nodes.

Expressions and the Code Node

The visual builder gets you most of the way. Expressions and the Code node get you the rest. This is the line between people who find n8n limiting and people who find it can do anything — and as a developer, you live on the right side of that line.

Expressions: small logic, inline

An expression is a snippet of JavaScript inside `{{ }}` that resolves to a value. Use them for the small stuff right where you need it:

```
{{ $json.email.toLowerCase().trim() }}
{{ $json.total > 100 ? 'priority' : 'standard' }}
{{ $json.tags.join(', ') }}
{{ $now.minus({ days: 7 }).toISO() }}
```

That last one uses **Luxon**, which n8n bundles for date math — `$now` and `$today` are Luxon DateTime objects, so `.plus()`, `.minus()`, `.toFormat()` all work. There's a JMESPath helper (`$jmespath()`) for digging through nested JSON, and a pile of built-in string/array/date helper methods n8n adds on top of vanilla JS. When an expression returns `undefined`, 90% of the time it's a typo in a field name or you're referencing the wrong node — check the actual item in the panel.

The Code node: when logic gets real

When the transformation is more than a one-liner, drop a **Code node**. You get full JavaScript (or Python, via Pyodide), and you choose the mode:

- **Run Once for All Items** — the code runs a single time and receives *all* items. Use this when you need to look across the whole batch: dedupe, sort, aggregate, reshape the set.
- **Run Once for Each Item** — the code runs per item, like a built-in node. Use this for per-record transforms.

A "run once for all items" Code node looks like this:

```
// `items` is the incoming array; return an array of items
const seen = new Set();
const deduped = [];

for (const item of items) {
  const key = item.json.email.toLowerCase();
  if (!seen.has(key)) {
    seen.add(key);
    deduped.push(item);
  }
}

return deduped;
```

The iron rule: **return an array of objects shaped like `{ json: {...} }`.** Return the wrong shape and the next node gets confused. n8n is forgiving about wrapping bare objects, but be explicit and you'll never wonder why downstream broke.

What you can and can't do in there

Code nodes run in a sandboxed environment. With **task runners** enabled (the `N8N_RUNNERS_ENABLED` flag from the Docker chapter), they execute in a separate process — better isolation and they can't take down the main instance. A few practicalities:

- You can `require` a curated set of built-in modules; arbitrary npm packages aren't available by default (there's an allow-list env var if you self-host and really need one).
- For HTTP calls, prefer the **HTTP Request node** over `fetch` in code — you get retries, auth handling, and pagination for free, and the request shows up in the execution log.
- `$input.all()`, `$input.first()`, `$('NodeName').all()` are how you reach data inside a Code node — same node-reference idea as expressions, just method form.

My rule of thumb

Reach for an expression first — it keeps the logic visible on the canvas. Reach for a Code node when the logic is genuinely a function: looping with state, complex branching, building a non-trivial payload. Don't turn the whole workflow into one giant Code node, though — at that point you've thrown away the observability and the per-node replay that made n8n worth using. The sweet spot is a visual workflow with code where code earns its place. This is the same 70/30 instinct from The 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer): let the tool do the boring 70%, and spend your code on the 30% that's actually yours.

Error Handling, Retries, and Idempotency

A workflow that only works when every API is up, every payload is well-formed, and the network never blips is a demo, not a system. The gap between the two is error handling, and n8n gives you good tools — if you actually use them. Most people don't until something silently fails for a week.

The Error Trigger and a global error workflow

The first thing to build is a dedicated **error workflow**. Create a new workflow whose first node is the **Error Trigger**. It receives details of any failed execution — which workflow, which node, the error message, a link to the run. Wire it to a Slack or email alert:

> Workflow *"Nightly Stripe Sync"* failed at node *"HTTP Request"* — `429 Too Many Requests`. View execution (…)

Then, in each important workflow's **Settings → Error Workflow**, point it at this one. Now every failure, anywhere, pages you with context instead of vanishing into the execution log. Build this *once*, point everything at it, and you've turned silent failure into a notification. This alone puts you ahead of most n8n setups I've seen.

Node-level retries

Many failures are transient — a timeout, a rate limit, a brief 503. Each node has, under **Settings**, retry options: turn on **Retry On Fail**, set the number of attempts (3–5 is the production standard — *never* infinite), and a wait between tries. Where you can, use **exponential backoff**: 1s, then 2s, then 4s, rather than hammering a struggling service every second. Add a little jitter so a hundred workflows that failed at once don't all retry in lockstep and create a thundering herd.

Continue vs. stop

By default a node error stops the workflow. Sometimes that's right. Sometimes you want the batch to keep going and deal with the failures separately. Each node can be set to **Continue (using error output)**, which gives the node a second output branch for failed items. Route the good items down the happy path and the failed ones to a dead-letter destination — a table, a Slack channel, a "to retry" queue. This is how you process 1,000 records and don't lose all of them because record 437 had a malformed email.

Idempotency: the one that bites you

Here's the trap. A workflow charges a customer, then fails on the *next* node (sending the receipt). It retries from the top. Now you've charged them twice. Retries and idempotency are inseparable: the moment you retry, you must make sure re-running can't duplicate side effects.

The fixes, in rough order of how often I use them:

- **Idempotency keys.** Many APIs (Stripe, PayPal, SendGrid) accept an `Idempotency-Key` header — send a stable key (e.g. derived from the order ID) and the API itself dedupes. This is the cleanest option; use it whenever the upstream supports it.
- **Check-before-write.** Before creating a record, query whether it already exists by a natural key. n8n's

Remove Duplicates node and a lookup step cover a lot of cases.
- **Mark progress.** Write a status flag (`processed: true`) so a re-run skips what's already done. Configure the node's retry to *skip on success* so a partially-succeeded batch doesn't redo its successes.

Think of it as: retries recover from transient errors, idempotency makes recovery *safe*. You need both. If you've read Background Jobs & Queues (https://rogerstringer.com/guides/background-jobs-and-queues), this is the same gospel — at-least-once delivery means design for duplicates — applied inside n8n.

Make failures visible

Finally: set important workflows to **save error executions** (Settings), even if you skip saving successful ones. When the alert fires, you want to open the exact failed run, see the precise item that broke and the data it carried, fix it, and replay. That loop — alert, inspect, fix, replay — is the whole point of running a durable engine instead of fire-and-forget scripts.

Practical Recipes

Enough theory. Here are workflows I actually build, described tightly enough that you can reproduce them. They're deliberately boring — the boring ones are the ones that earn their keep running quietly for years.

1. Webhook → validate → store

The bread-and-butter shape. A **Webhook** trigger receives a POST. First node after it is a validation step — a Code node or **IF** node that checks the payload is well-formed and (if the sender signs requests) verifies the signature. Bad payloads route to a 400 response; good ones continue to a **Create/Update** node that writes to your database, then a Webhook **Respond** node returns 200. Guard the webhook with header auth (see the hardening chapter). This is how you receive form submissions, Stripe events, GitHub hooks — anything that pushes to you.

2. Scheduled sync between two systems

A **Schedule** trigger fires (say, every 15 minutes). An HTTP Request or service node pulls records changed since the last run — store a `last_synced_at` timestamp somewhere (a single-row table, or n8n's static data) and pass it as a filter so you only fetch the delta. **Split Out** the results into items, transform each, then upsert into the destination with **Remove Duplicates** or an idempotency check so re-runs are safe. Wire it to your error workflow. This pattern replaces an astonishing number of brittle cron scripts.

3. Fan-out API orchestration

One event needs to touch five services. A new signup, for example: create the CRM contact, add them to the email list, post to a Slack channel, provision their workspace, send a welcome email. Lay these out as sequential nodes (or parallel branches where order doesn't matter), passing the user data down the chain and referencing the original payload with `{{ $('Webhook').item.json... }}`. Set each external call to continue-on-error with its failures routed to a dead-letter branch, so a flaky CRM doesn't block the welcome email. This is the orchestration n8n is *made* for — the kind of multi-service glue that's miserable to write and maintain by hand.

4. Batch processing with rate limits

You have 5,000 records to push to an API that allows 10 requests/second. Get them into items, then use the **Loop Over Items** (batching) node to process in chunks, with a small **Wait** between batches to stay under the limit. Turn on node retries with backoff for the inevitable 429s. Aggregate the results at the end and send yourself a one-line summary: *"4,981 synced, 19 failed — see attached."*

5. The human-in-the-loop approval

Not everything should be fully automatic. A workflow drafts something — a refund, a published post, an outbound email — then **pauses** and sends an approval request (Slack button, or n8n's Wait-for-webhook / form). A human clicks approve or reject, and the workflow resumes down the matching branch. This is the safety valve for automations with real consequences: let the machine do the assembly, keep a person on the trigger.

6. Scheduled report / digest

A cron fires each morning, queries your data (sales, signups, errors, whatever you watch), formats a tidy summary — and here's the modern twist — optionally pipes the numbers through an AI node to write a plain-English narrative, then posts it to Slack or email. I run a few of these; they're the cheapest way to stay on top of a system without logging into five dashboards. (If you want this *outside* n8n too, my own setup leans on scheduled agent tasks — see Building Your Agentic OS (https://rogerstringer.com/guides/building-your-agentic-os).)

Notice the through-line: get data into items, reshape, act per item, handle errors, summarize. Once those moves are muscle memory, most "can n8n do X?" questions answer themselves — X is just these recipes recombined.

Building AI Agents

This is where n8n stopped being "automation software" for me and became something closer to an agent runtime. Since the 2.0 line, n8n ships native LangChain-based nodes, and the centerpiece is the **AI Agent** node. If you've built agents from scratch — the tool-calling loop, the message history, the stop condition — you'll recognize exactly what this node is doing, just with the plumbing handled.

What the AI Agent node actually is

The AI Agent node is a node that runs a reasoning loop. You plug things *into* it from underneath — these are "cluster nodes," sub-nodes that attach to the agent rather than sitting in the main flow:

- **A Chat Model** (required) — the brain. OpenAI, Anthropic, Google Vertex, Mistral, a local model via **Ollama**, or anything OpenAI-compatible. Self-hosting + Ollama means you can run agents with zero per-token cost and zero data leaving your box, which is a big deal for a lot of use cases.
- **Memory** (optional) — so the agent remembers the conversation. Window Buffer Memory keeps the last N messages; Postgres Chat Memory persists across sessions.
- **Tools** (optional, but the whole point) — things the agent can *do*.

Internally the agent runs a function-calling / ReAct loop: the model reasons, decides whether to call a tool, n8n executes that tool, feeds the result back, and the loop continues until the model produces a final answer or hits a step cap. You don't write that loop — you assemble its parts on the canvas.

Tools are where it gets real

A chatbot that can only talk is a toy. An agent that can *act* is useful, and in n8n a "tool" can be:

- A **built-in tool** node (HTTP Request as a tool, a calculator, a SerpAPI search, Wikipedia, etc.).
- A **sub-workflow** exposed as a tool — this is the killer feature. Any n8n workflow you can build becomes a capability the agent can call. Want the agent to be able to "look up a customer's orders"? Build a workflow that does exactly that, expose it as a tool with a clear description, and the agent will call it when the conversation calls for it.

That sub-workflow-as-tool pattern is the bridge between everything in the earlier chapters and AI. All those recipes — the database lookups, the API orchestration — become tools an agent can wield. You're not choosing between "automation" and "AI agent"; the automation *is* the agent's hands.

A concrete build: a support triage agent

- **Trigger:** a Chat trigger (or a webhook from your support inbox).
- **AI Agent** with an Anthropic or OpenAI chat model and a system prompt: *"You are a support triage assistant. Classify the issue, look up the customer, and either answer from the knowledge base or escalate."*
- **Tools:** a "look up customer" sub-workflow (queries your DB), a "search knowledge base" tool (the RAG setup from the next chapter), and an "escalate to human" sub-workflow (posts to Slack with a button).
- **Memory:** Postgres Chat Memory keyed by conversation ID, so a back-and-forth holds context.

The agent reasons over each message, calls the tools it needs, and only escalates when it should. That's a genuinely useful system, assembled visually, running on infrastructure you own.

The instinct to keep

Two cautions from actually running these. First, **the description of each tool is doing the prompting** — a vague tool description means the model calls the wrong tool at the wrong time. Write tool descriptions like you're writing function docs for a junior dev. Second, **agents are non-deterministic; your guardrails shouldn't be.** Keep irreversible actions (refunds, deletes, sends) behind a deterministic check or a human-in-the-loop approval, exactly like the recipe from the last chapter. Let the agent decide *what* to do; keep a hard gate on the things it would be expensive to get wrong. That balance — give the model real capability, keep humans on the dangerous levers — is the same philosophy running through Building Your Agentic OS (https://rogerstringer.com/guides/building-your-agentic-os).

RAG and Vector Stores

An agent is only as good as what it knows. Out of the box, a chat model knows its training data and nothing about *your* docs, your products, your policies. **Retrieval-Augmented Generation** fixes that: you store your knowledge as embeddings in a vector database, and at query time you retrieve the most relevant chunks and hand them to the model as context. n8n makes both halves — ingestion and retrieval — buildable as workflows.

The two workflows you need

RAG is always two jobs, and it helps to think of them separately:

1. Ingestion (write side). Take your source documents, split them into chunks, embed each chunk, and store the vectors. In n8n: a trigger (manual, or a Directus webhook when a doc changes), a node to load the document, a **Text Splitter** to chunk it, an **Embeddings** node (OpenAI, Cohere, or a local model), and a **Vector Store** node in *insert* mode. Run this whenever your knowledge changes.

2. Retrieval (read side). At query time, embed the user's question, search the vector store for the nearest chunks, and feed them to the model. In n8n this is usually a **Vector Store** node attached to your AI Agent as a **tool** (often via the "Vector Store Question Answer" / retriever tool), so the agent can decide to search the knowledge base when it needs to.

Picking a vector store

n8n supports the usual suspects: **Pinecone** and **Qdrant** (dedicated vector DBs), **Supabase** and **PGVector** (Postgres with the pgvector extension), and an **in-memory** store for testing. My bias, predictably, is toward the boring, self-hostable option: if you're already running Postgres for n8n and Directus, **PGVector** means one less moving part — your vectors live right next to your data. Pinecone and Qdrant are excellent when you outgrow that or want a purpose-built engine, but don't add a new database to your stack on day one if Postgres will do. Start with the in-memory store to prove the workflow, then swap in PGVector.

The chunking is the hard part

Everyone obsesses over which vector DB to use. In practice, retrieval quality lives or dies on **chunking and embeddings**, not the store. Chunks that are too big bury the relevant sentence in noise; too small and they lose context. Split on natural boundaries (paragraphs, headings) rather than blind character counts where you can, keep a little overlap between chunks so a thought isn't severed mid-sentence, and store useful metadata (source, title, last-updated) alongside each vector so you can filter and cite. When an agent gives a confidently wrong answer, the culprit is almost always retrieval handing it bad chunks — not the model.

Keep it fresh

The trap with RAG is staleness. You ingest your docs once, ship it, and six months later the agent is confidently quoting a refund policy you changed in March. This is exactly where n8n earns its keep: wire the *ingestion* workflow to a webhook so that whenever a document updates in your source of truth — Directus, Notion, a Google Drive folder — the corresponding vectors get re-embedded automatically. Your knowledge base stays in sync with your real content because the same tool that runs the agent also runs the pipeline that feeds it. That closed loop, ingestion and retrieval living in one system you own, is the whole argument for doing RAG in n8n instead of stitching together three SaaS products.

Directus and n8n Together

I run Directus and n8n side by side on more or less every project, and it's not an accident — they're complementary in a way that's almost suspicious. Directus is the system of record with a great API and an admin UI; n8n is the orchestration engine that reaches out to the rest of the world. Each is weak exactly where the other is strong. This chapter is the pairing, end to end.

The division of labor

Directus owns your **data, schema, permissions, and admin experience**. It has its own automation engine — **Flows** — which is excellent for in-app, data-event-driven logic that you want logged right next to your content (auto-slugs, status-change emails, simple validation). I covered Flows in Mastering Directus (https://rogerstringer.com/guides/mastering-directus), and the rule I gave there still holds: *if the automation is about my Directus data and triggered by my Directus events, it's a Flow; if it's complex multi-service orchestration, long-running, or needs a hundred pre-built integrations, it's n8n.*

n8n owns the **outbound, the multi-step, and the AI**. Directus shouldn't be trying to run a rate-limited batch sync to five APIs or a RAG pipeline. n8n shouldn't be your database. Keep each doing what it's good at and the seams stay clean.

Wiring 1: Directus Flow !ħ8n webhook

The most common connection. Something happens in Directus, and you want heavy lifting done elsewhere. In Directus, a Flow with an **event trigger** (item created/updated) fires a **Request** operation that POSTs to an n8n **Webhook** trigger. Directus hands off the changed record; n8n takes it from there — enriches it, fans it out to other services, runs it past an AI agent, whatever. This keeps Directus's own automation lean and lets n8n be the muscle. Secure the webhook with header auth and have the Flow send the matching header.

Wiring 2: n8n !Directus API

The other direction, just as common. An n8n workflow needs to read or write your data. Directus gives you a clean REST and GraphQL API plus a static token, so in n8n you either use a Directus community node or just the **HTTP Request** node pointed at `https://your-directus/items/collection` with a bearer token credential. Now any n8n workflow — a scheduled report, an AI agent's tool, an inbound webhook handler — can query and update your real data. This is how the agent from the AI chapter "looks up a customer": it's an HTTP Request to Directus under the hood.

Wiring 3: shared Postgres

Because both can sit on the same Postgres server (separate databases), some patterns get very tidy. The **PGVector** store from the RAG chapter can live in the same Postgres instance as your Directus content. Your n8n ingestion workflow reads documents from Directus via its API and writes embeddings to pgvector — one database server, one backup, one thing to run.

A complete example: content !þublished !ṡyndicated

Put it all together. An editor sets an article's status to `published` in the Directus admin. A Directus Flow fires a webhook to n8n. n8n picks it up and: re-embeds the article into the RAG store (so the support agent

now knows about it), posts a summary to Slack, generates social copy with an AI node, schedules it out, and pings a frontend rebuild. The editor did one thing — flip a status — in the tool built for editors. Everything downstream happened in the tool built for orchestration. That's the whole philosophy: the right tool holding each responsibility, connected by a webhook and an API token. It's also, not coincidentally, the Hypermedia Stack (https://rogerstringer.com/guides/the-hypermedia-stack) backend with its automation layer made explicit.

Credentials, Environments, and Source Control

Once n8n is doing real work, you hit the questions every serious tool eventually raises: where do secrets live, how do I separate staging from production, and how do I not lose my workflows? n8n has answers, with a couple of caveats worth knowing before you commit to a way of working.

Credentials are encrypted, centralized, and reusable

In n8n you create a **credential** once — an API key, an OAuth connection, a database login — and reference it from any node that needs it. They're stored encrypted in the database, keyed by that `N8N_ENCRYPTION_KEY` I keep harping on. Two habits that pay off: name credentials clearly (`Stripe — Production`, not `Stripe2`), and **never paste a secret directly into a node field** where it'd be saved in plaintext in the workflow JSON — always go through a credential. For self-hosters who want secrets to live outside n8n entirely, there's **external secrets** support (Vault, AWS/GCP secret managers) on the enterprise tier; for most setups, encrypted credentials plus a backed-up key is plenty.

Config that changes per environment

Things that differ between staging and production — a base URL, a Slack channel, a feature flag — shouldn't be hardcoded in nodes. Two clean options:

- **Environment variables** read in expressions: `{{ $env.API_BASE_URL }}`. Set them differently per instance. Simple, works on community edition.
- **Variables** (the built-in key-value store) for values you want to manage in the UI rather than redeploy to change. (Some of this is gated to paid tiers — check what your edition includes.)

The principle is the same one from the Context Engineering (https://rogerstringer.com/guides/context-engineering) and config discipline I bang on about elsewhere: keep the environment-specific stuff *out* of the logic, so the same workflow runs unchanged everywhere and only the surrounding config differs.

Staging vs. production

Don't develop on the instance that's running your live automations. Run a **separate staging instance** — same Compose, different box or at least different database — build and test there, then promote. n8n's enterprise tiers have formal *environments* and git-backed promotion; on community edition you do it with discipline and the source-control approach below. Either way, the rule is: the production instance is for running, not for editing live.

Get your workflows into git

This is the one people skip and regret. Workflows are JSON — they belong in version control, not trapped in a database you hope you backed up. Options, roughly in order of effort:

- **Manual export.** Each workflow can be exported to a JSON file from the UI. The n8n CLI can bulk-export all of them (`n8n export:workflow --all --output=...`). Commit the files. Crude but it works, and it's scriptable into a nightly job.
- **API export.** A meta n8n workflow that calls n8n's own API on a schedule, dumps every workflow to JSON, and commits to a git repo. Very on-brand: n8n backing up n8n.
- **Git-based source control** is a first-class feature on the enterprise tier, with proper push/pull of workflows and credentials between environments. If you're at that scale, it's the cleanest path.

Whatever you choose, the goal is a git history of your automation logic: diffable, reviewable, restorable. When a workflow that worked last week is suddenly broken, `git diff` should tell you what changed. That's table stakes for treating these as software — which, once they're running your business, they are.

Don't forget: the database is still the source of truth for *state*

Git holds your workflow *definitions*. It does **not** hold execution history, credentials (those are encrypted in Postgres), or the static data workflows accumulate. So git is your version control; the Postgres backup plus the encryption key (from the hardening chapter) is your disaster recovery. You need both — they cover different failures.

Real-World Patterns and Gotchas

The stuff that doesn't fit neatly into a chapter but that I'd want a teammate to know before they're three workflows deep. Scar tissue, mostly.

The encryption key is everything

I've said it three times and I'll say it once more, because it's the disaster I see most. **Back up `N8N_ENCRYPTION_KEY` separately from your database.** Lose it and every stored credential is unrecoverable ciphertext, even with a perfect database backup. Put it in your password manager the day you set up the instance.

Memory and large payloads

n8n holds the items flowing through a workflow in memory. A workflow that pulls 50,000 records into one big array, or hauls large binary files item by item, will eat RAM and can OOM the process. Mitigations: page through large datasets and process in batches (Loop Over Items) rather than loading everything at once; for big files, stream or hand off to object storage instead of carrying binaries through every node; and in queue mode, remember the *worker's* memory is the ceiling, not the main instance's. If a workflow gets killed mid-run, payload size is the first thing to check.

"Why did it run twice?" — trigger gotchas

- **Test vs. production webhook URLs are different.** The editor's "Test" URL only listens while you're actively testing; the production URL works when the workflow is *active*. Sending a real webhook to the test URL (or vice versa) is a classic "why isn't it firing" hour.
- **Activating a workflow is what registers its triggers.** A saved-but-inactive workflow does nothing on a schedule or webhook. Toggle it active.
- **Schedule triggers use `GENERIC_TIMEZONE`.** If your nightly job runs at the wrong hour, your timezone env var is wrong (or unset, defaulting to UTC).

Expressions returning `undefined`

The everyday papercut. Almost always one of: a misspelled field name, referencing the *previous node* when the data is actually two nodes back (use `{{ $('NodeName')... }}`), or the item structure isn't what you assumed. Don't guess — open the node's input panel and look at the actual JSON. The data is right there; trust it over your memory of what the API "should" return.

Versioning and breaking changes

n8n moves fast. Nodes get new versions, and major releases (the 1.x to 2.x jump especially) can change behavior. Pin a major version in your image tag, read release notes before upgrading, and test upgrades on staging first. The flip side: don't fall *years* behind either, because security fixes and the good AI nodes only land in current releases. Steady, deliberate updates beat both "never touch it" and "always `:latest`."

Community nodes: useful, with a caveat

There's a rich ecosystem of community nodes for services n8n doesn't cover natively. They're genuinely handy — and they're third-party code running in your instance. Vet them like any dependency: check the source, the maintenance, the download count. For something security-sensitive, the plain **HTTP Request node** against the service's API is often the more trustworthy path even if it's a little more work.

When n8n is the wrong tool

The honest part. n8n is fantastic glue, but it's not a general-purpose application runtime. If you find yourself building something with dozens of Code nodes, intricate state machines, and logic that would genuinely be clearer as a normal codebase — that's the signal to write an actual service and let n8n *call* it, not host it. The same judgment from Build vs. Buy in the AI Era (https://rogerstringer.com/guides/build-vs-buy-in-the-ai-era) applies inside n8n itself: use it for what it's great at — event-driven orchestration, integration, agent assembly — and don't contort it into being your whole backend. Knowing where the tool ends is part of mastering it.

About Roger

I'm Roger Stringer — I build things, break them, and write up what I learned so you don't have to learn it the hard way. These Field Guides come straight out of that work.

Working on something bigger? I take on a handful of fractional CTO (https://rogerstringer.com/guides/the-fractional-cto-field-guide) engagements — helping founders and teams set technical direction, build AI-powered workflows, and actually ship the hard parts. If you're wrestling with the kind of problem this guide covers and want someone in your corner who's done it before, that's exactly what I help with. Drop me a line. (mailto:roger.stringer@hey.com)

And if a guide helped, got something wrong, or you just want to compare notes, I'd love to hear from you:

- **Email** — roger.stringer@hey.com (mailto:roger.stringer@hey.com)
- **X** — @freekrai (https://x.com/freekrai)
- **GitHub** — github.com/freekrai (https://github.com/freekrai)
- **LinkedIn** — linkedin.com/in/rogerstringer (https://www.linkedin.com/in/rogerstringer)

New guides go up as I hit problems worth documenting — follow along wherever suits you.