The Ultimate Guide to Deploying n8n Community Edition

Shannon Atkinson, House of Loops

September 2026

- 6. Monitoring and Troubleshooting
- 7. Performing Updates
- 8. Advanced Scenarios
- 9. Troubleshooting Common Issues
- Chapter 19: Kubernetes: Ingress and Let's Encrypt
- 1. Install Cert-Manager
- 2. Deploy an Ingress Controller
- 3. Configure Let's Encrypt ClusterIssuers
- 4. Configure DNS
- 5. Create Ingress Resource with TLS
- 6. Observe Certificate Issuance
- 7. Verify HTTPS Access
- 8. Custom Error Pages and Advanced Annotations
- 9. Troubleshooting Common Issues
- Chapter 20: Kubernetes: Scaled Queue Mode
- 1. Deploy Redis as a StatefulSet
- 2. Configure Environment Secrets
- 3. Deploy the Trigger Deployment
- 4. Deploy the Worker Deployment with HPA
- 5. Testing and Verification
- 6. Backup and Restore
- 7. Troubleshooting Common Issues
- Chapter 21: Deploying with Helm
- 1. Scaffold the Helm Chart
- 2. Define Chart Metadata (`Chart.yaml`)
- 3. Configure Default Values (`values.yaml`)
- 4. Create the Secret (`templates/secrets.yaml`)
- 5. Template the Deployments (`templates/deployment.yaml`)
- 6. Service, PVC, and Ingress Templates
- 7. Package and Install the Chart
- 8. Upgrades and Rollbacks
- 9. Dependencies and Subcharts
- 10. Hooks for Backups and Migrations
- 11. Linting and Testing
- 12. Publishing Your Chart
- 13. Troubleshooting Common Issues
- Chapter 22: AWS ECS Fargate
- 1. Build and Push Docker Image to ECR
- 2. Define IAM Roles & Policies
- 3. Create an ECS Cluster
- 4. Provision ElastiCache for Redis
- 5. Decide How Binary Data Persists
- 6. Create the Main Task Definition
- 7. Create the Worker Task Definition
- 8. Configure the Application Load Balancer
- 9. Deploy the ECS Services
- 10. Enable Service Auto Scaling for Workers
- 11. Configure CloudWatch Logs & Metrics
- 12. Manage Secrets with AWS Secrets Manager
- 13. Troubleshooting Common Issues
- Chapter 23: Google Cloud Run
- 1. Containerize n8n and Push to Artifact Registry
- 2. Provision Cloud SQL for PostgreSQL
- 3. Create a Service Account and Grant IAM Roles

- 5. Branching Strategies
- 6. Tagging and Releasing
- 7. Workflow Metadata
- 8. Secrets and Credentials
- 9. Merging and Conflict Resolution
- 10. This Chapter vs. n8n's Paid Source Control and Environments
- 11. Troubleshooting Common Issues
- Chapter 33: Upgrading to n8n 2.0
- 1. Why 2.0 Matters
- 2. Pre-Flight Checklist
- 3. Using the Built-In Migration Report
- 4. The Breaking-Change Checklist
- 5. Behavior Changes
- 6. Security
- 7. Data
- 8. Configuration and Environment
- 9. CLI and Workflow
- 10. External Hooks
- 11. Release Channels
- 12. Performing the Upgrade
- 13. Rollback Plan
- 14. Troubleshooting Common Issues
- Chapter 34: Preparing for n8n 3.0
- 1. The headline change: Docker-based self-hosting only
- 2. Migrating off npm, npx, and PM2
- 3. Removed nodes and their replacements
- 4. Execute Workflow node: old behavior removed
- 5. AI Agent node: version 1 and its modes removed
- 6. `$getPairedItem` removed
- 7. Tighter security defaults
- 8. Retired capabilities
- 9. Finding affected workflows today
- 10. A 90-day timeline
- 11. Troubleshooting Common Issues
- Appendix A: Legacy Installs with Node.js, npm, and PM2 (until n8n 3.0)
- 10. Troubleshooting Common Issues
- Appendix B: Environment Variable Reference for n8n 2.x
- Conclusion: Your Journey with n8n

Table of Contents

Preface

Welcome to **The Ultimate Guide to Deploying n8n Community Edition**, second edition. The first edition, written in the spring of 2025 against n8n 1.89, walked from a laptop install to multi-region Kubernetes. Since then n8n shipped 2.0 (December 2025), moved the Code node onto isolated task runners, replaced the activate toggle with publishing, tightened a dozen security defaults, and announced that 3.0 (October 2026) will support Docker-based self-hosting only. This edition is written against **n8n 2.38** and is arranged so that nothing in it stops working when 3.0 arrives.

Three things changed in how the book is built:

- **Docker first.** Every chapter starts from one reference Docker Compose stack: PostgreSQL 18, a pinned n8n image, and the task-runner sidecar. You learn it once, on your laptop, and carry the same file to a VPS, a cloud VM, Swarm, or Kubernetes. The npm and PM2 methods live on in Appendix A for people maintaining an existing install, with a clear cutoff date.
- **Correct configuration, not plausible configuration.** The first edition carried a few settings that had already been removed from n8n, and some that never existed. This edition uses only variables you can find in the n8n documentation, and Appendix B lists each one with its 2.x default.
- **Two chapters on major versions.** Upgrading to 2.0 and preparing for 3.0 each get a full chapter with the breaking changes, how to tell whether you are affected, and what to do.

The book is free, and it is the companion to the House of Loops community on Skool, where the workflows built in these chapters are shared and where the author answers questions. If a step does not work as written, that is the place to say so.

Introduction to n8n Community Edition

What is n8n?

n8n is a source-available workflow automation tool that sits between the no-code tools and writing everything yourself. Each node in a workflow does one job: call an API, query a database, transform data, run a piece of JavaScript or Python, or hand a task to an AI agent. n8n offers:

- A **visual editor** where workflows are built by connecting nodes, with expressions for the parts that need code.
- **Over 400 built-in integrations** and more than a thousand community nodes, plus the HTTP Request node for everything else.
- **AI nodes** for agents, tools, memory, and vector stores, and an MCP server so other agents can call your workflows.
- **Self-hosting** under a fair-code licence, so the data and the credentials stay on infrastructure you control.

Key features of the Community Edition

- **Persistent workflows and credentials** in SQLite or, for anything beyond a laptop, PostgreSQL. Credentials are encrypted with a key you own.
- **Queue mode** that splits n8n into a main instance, webhook processors, and worker processes coordinated through Redis, so throughput scales horizontally.
- **Task runners** that execute Code nodes in an isolated sidecar, which since 2.0 is the only barrier between user-supplied code and your database.
- **Publishing with versions**, workflow history, and a CLI for exporting, importing, and publishing workflows from a pipeline.
- **A REST API and webhooks**, Prometheus metrics, and native OpenTelemetry tracing.
- **User management** with an owner account, admins, members, and two-factor authentication. Single sign-on, projects with fine-grained roles, external secrets, and Git-backed source control are paid features and are labelled as such where they come up.

Why deploy n8n yourself?

1. **Data ownership.** Credentials, execution data, and binary files never leave your environment.
2. **Cost.** The Community Edition has no per-user or per-execution fee; you pay for the machine it runs on.
3. **Control.** You choose the database, the proxy, the secrets store, the scaling model, and the upgrade cadence.
4. **Security posture.** You can put n8n behind your own identity provider, network policies, and audit pipeline.
5. **Fit.** From a Raspberry Pi to a multi-region Kubernetes fleet, the same software runs everywhere in this book.

Versions used in this edition

Component	Version	Notes
n8n	2.38.1	Pin this tag on the `n8nio/n8n` image. `stable` is the rolling alias.
Task runners	2.38.1	`n8nio/runners`; must match the n8n version exactly.
PostgreSQL	18	16 and 17 also work.

Component	Version	Notes
Redis	7	Queue mode only.
Docker Compose	v2	`docker compose`, no `version:` key.
n8n 3.0	October 2026	Docker-only; see the chapter on preparing for 3.0.

When a newer n8n release is out, change the two pinned tags, read the release notes for migrations, and everything else in this book still applies.

How the book is organised

1. **Local and small deployments.** Your laptop with Docker Compose, a deeper look at Docker itself, ARM boards, and task runners.
2. **VPS and cloud VMs.** The reference production stack on Ubuntu, reverse proxies and HTTPS, the four big clouds, tunnels for private instances, and platform-as-a-service options.
3. **Containers and Kubernetes.** Compose with PostgreSQL, queue mode, Swarm, four Kubernetes chapters, Helm, and the managed container services.
4. **Operations.** High availability, multi-region, backup and disaster recovery, CI/CD and GitOps, observability, security hardening, and workflow versioning.
5. **Major versions.** Upgrading to 2.0 and preparing for 3.0.
6. **Appendices.** Legacy npm and PM2 installs, and the environment variable reference.

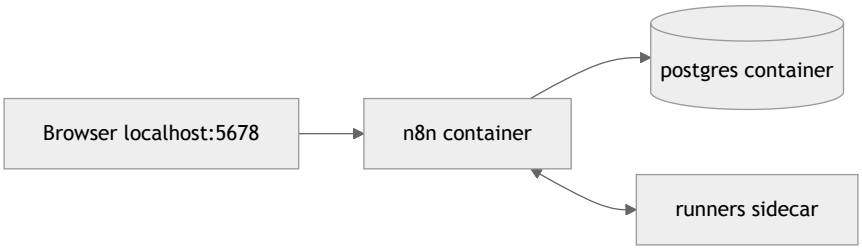
Chapters refer to each other by title rather than number so that you can read in any order.

Chapter 1: n8n on Your Laptop with Docker Compose

Overview

This chapter gets a working n8n on your own machine in about fifteen minutes. You will install Docker, write two small files, start three containers, create your owner account, and run a workflow that proves the whole stack works end to end.

The stack you build here is the same stack the rest of the book builds on: PostgreSQL for storage, the `n8nio/n8n` image for the editor and the execution engine, and the `n8nio/runners` sidecar that executes every Code node. Later chapters change the hostname, the proxy, and the number of containers. They do not change the shape of this file.



Diagram

Prerequisites

- A laptop with at least 4 GB of free RAM and 5 GB of free disk.
- macOS 13 or later, Windows 10/11 with WSL2, or a Linux distribution with a current kernel.
- A terminal you are comfortable typing in.
- Administrator rights on the machine, to install Docker.

No Node.js. No `npm`. Nothing on your machine but Docker.

1. Why Docker, and Only Docker

n8n 3.0 ships in **October 2026** and supports **Docker-based self-hosting only**. The npm and `npx n8n` install paths are already deprecated in the current 2.x line and will not carry forward.

Even before that deadline, Docker is the honest choice for a beginner:

- **The runtime is fixed.** The npm path still works today, but only on Node.js 20.19 through 24. Get that wrong and you get failures that look like n8n bugs. The image pins the runtime for you.
- **Task runners need a second process.** Since n8n 2.0, every Code node runs inside a task runner, not inside the main process. The supported production mode is `external`, which means a separate `n8nio/runners` container. That is a Compose file, not an `npm install`.
- **What you build here is what you deploy.** The file in this chapter is the file you copy to a VPS in the Ubuntu VPS with Docker Compose chapter. Nothing has to be relearned.

Two rules for the whole book. The command is `docker compose` with a space, never the old `docker-compose` binary. And every image is pinned to an exact version, never `latest`. Compose v2 is the only supported version, and the `version:` key at the top of a Compose file is obsolete — you will not see it here.

---

2. Install Docker

2.1 macOS

Download Docker Desktop from https://docs.docker.com/desktop/setup/install/mac-install/ and pick the build for your chip. Apple Silicon and Intel each have their own installer.

Drag it to Applications, launch it, and accept the terms. When the whale icon in the menu bar stops animating, Docker is running. Confirm from a terminal:

```
docker --version
docker compose version
```

You want Docker 24 or newer and Compose v2.x. If `docker compose version` errors but `docker --version` works, Docker Desktop is out of date; update it before continuing.

Apple Silicon needs no special flags. The n8n images are multi-arch and run natively on `arm64`.

2.2 Windows with WSL2

On Windows you run Docker Desktop with the WSL2 backend, and you keep your project files inside the Linux filesystem.

1. Open PowerShell as Administrator and install WSL2 with Ubuntu:

   ```
   wsl --install -d Ubuntu
   ```

   Reboot when prompted, then finish the Ubuntu first-run setup (username and password).

2. Install Docker Desktop from https://docs.docker.com/desktop/setup/install/windows-install/. In **Settings → General**, confirm **Use the WSL 2 based engine** is checked. In **Settings → Resources → WSL Integration**, enable integration for your Ubuntu distribution.

3. From now on, work inside the Ubuntu shell, not PowerShell. Open it from the Start menu or run `wsl` in a terminal, then verify:

   ```
   docker --version
   docker compose version
   ```

Put the project folder inside WSL, not on `/mnt/c`. Your Windows drives are mounted at `/mnt/c`, and that path works — but every file read crosses a translation layer between Windows and Linux. Bind mounts on `/mnt/c` are slow enough to make container startup and workflow execution visibly sluggish. Create your project under your WSL home directory (`~/`, which is `/home/<you>`) instead. Everything in this chapter uses named volumes, which live inside WSL regardless, so the only file that matters is the Compose file itself.

2.3 Linux

Do not install Docker Desktop on Linux. Install Docker Engine plus the Compose plugin, using Docker's convenience script or your distribution's instructions at https://docs.docker.com/engine/install/.

```
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh
```

Then add yourself to the `docker` group so you do not need `sudo` for every command:

```
sudo usermod -aG docker $USER
newgrp docker
```

```
docker compose version
```

Log out and back in if `newgrp` does not take effect in new terminals.

---

3. Create the Project Folder and Secrets

Everything lives in one folder. Two files, both of which you will reuse in later chapters.

```
mkdir -p ~/n8n
cd ~/n8n
```

Windows users: this must be your WSL home, so run it in the Ubuntu shell.

Generate the two secrets now, because you will paste them into `.env` in a moment:

```
openssl rand -hex 32   # N8N_ENCRYPTION_KEY
openssl rand -hex 32   # N8N_RUNNERS_AUTH_TOKEN
```

`N8N_ENCRYPTION_KEY` encrypts every credential n8n stores. Generate it once and never change it. If you lose it, the credentials in your database are unreadable — the workflows survive a restore, the credentials do not.

`N8N_RUNNERS_AUTH_TOKEN` is the shared secret between the n8n container and the runners sidecar. Both containers must present the same value or the runner will not connect and every Code node will fail.

Now create `.env`, pasting your two generated values:

```
# Generate with: openssl rand -hex 32
N8N_ENCRYPTION_KEY=replace-with-64-hex-characters
N8N_RUNNERS_AUTH_TOKEN=replace-with-a-long-random-string

POSTGRES_USER=n8n
POSTGRES_PASSWORD=replace-me
POSTGRES_DB=n8n

# Laptop values. Later chapters replace these with a real hostname.
N8N_HOST=localhost
GENERIC_TIMEZONE=America/Los_Angeles
```

Set `GENERIC_TIMEZONE` to your own zone. It decides when Schedule Trigger nodes fire.

Two notes on `.env` syntax, because n8n 2.0 upgraded its dotenv parser: a `#` starts a comment, and values containing backticks must be quoted. Your generated hex strings contain neither, so plain values are fine.

If this folder is ever a git repository, add `.env` to `.gitignore` before you commit anything.

---

4. Write `compose.yml`

Create `compose.yml` in the same folder:

```yaml
services:
  postgres:
    image: postgres:18
    restart: unless-stopped
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    volumes:
      - postgres_data:/var/lib/postgresql
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER} -d ${POSTGRES_DB}"]
      interval: 10s
      timeout: 5s
      retries: 5

  n8n:
    image: n8nio/n8n:2.38.1
    restart: unless-stopped
    ports:
      - "5678:5678"
    environment:
      N8N_ENCRYPTION_KEY: ${N8N_ENCRYPTION_KEY}
      DB_TYPE: postgresdb
      DB_POSTGRESDB_HOST: postgres
      DB_POSTGRESDB_PORT: 5432
      DB_POSTGRESDB_DATABASE: ${POSTGRES_DB}
      DB_POSTGRESDB_USER: ${POSTGRES_USER}
      DB_POSTGRESDB_PASSWORD: ${POSTGRES_PASSWORD}
      N8N_HOST: ${N8N_HOST}
      N8N_PROTOCOL: http
      WEBHOOK_URL: http://localhost:5678/
      N8N_EDITOR_BASE_URL: http://localhost:5678/
      GENERIC_TIMEZONE: ${GENERIC_TIMEZONE}
      TZ: ${GENERIC_TIMEZONE}
      N8N_RUNNERS_MODE: external
      N8N_RUNNERS_BROKER_LISTEN_ADDRESS: 0.0.0.0
      N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
      N8N_NATIVE_PYTHON_RUNNER: "true"
      N8N_DEFAULT_BINARY_DATA_MODE: filesystem
      EXECUTIONS_DATA_PRUNE: "true"
      EXECUTIONS_DATA_MAX_AGE: 336
      N8N_DIAGNOSTICS_ENABLED: "false"
    volumes:
      - n8n_data:/home/node/.n8n
      - n8n_files:/home/node/.n8n-files
    depends_on:
      postgres:
        condition: service_healthy

  runners:
    image: n8nio/runners:2.38.1
    restart: unless-stopped
    environment:
      N8N_RUNNERS_TASK_BROKER_URI: http://n8n:5679
      N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
      N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT: 15
    depends_on:
      - n8n

volumes:
  postgres_data:
  n8n_data:
  n8n_files:
```

What the less obvious settings do:

- `N8N_PROTOCOL: http`, `WEBHOOK_URL`, and `N8N_EDITOR_BASE_URL` are the laptop values. Every chapter that puts n8n behind a proxy switches these to `https` and a real hostname and adds

`N8N_PROXY_HOPS`.

- `N8N_RUNNERS_MODE: external` with `N8N_RUNNERS_BROKER_LISTEN_ADDRESS: 0.0.0.0` tells the n8n container to open its task broker on port 5679 so the sidecar can reach it over the Compose network. The other mode, `internal`, runs the runner as a child process and is not for production; use `external` from the start so nothing changes later. (`N8N_RUNNERS_ENABLED` was required on 1.x and is deprecated from 2.0 — leave it out.)
- `N8N_NATIVE_PYTHON_RUNNER: "true"` enables the native Python runner. n8n 2.0 removed the old Pyodide-based Python in favour of this.
- `EXECUTIONS_DATA_PRUNE` with `EXECUTIONS_DATA_MAX_AGE: 336` deletes execution history older than 14 days, so your laptop's Postgres volume does not grow forever.
- `N8N_DIAGNOSTICS_ENABLED: "false"` turns off telemetry.

Both images are pinned to `2.38.1`, and they must always match. A runners image on a different version than n8n is unsupported.

---

5. Start the Stack

```
docker compose up -d
```

The first run downloads three images, so give it a minute or two. Then check what is running:

```
docker compose ps
```

You want all three services in state `running`, with `postgres` showing `healthy`. Watch n8n come up:

```
docker compose logs -f n8n
```

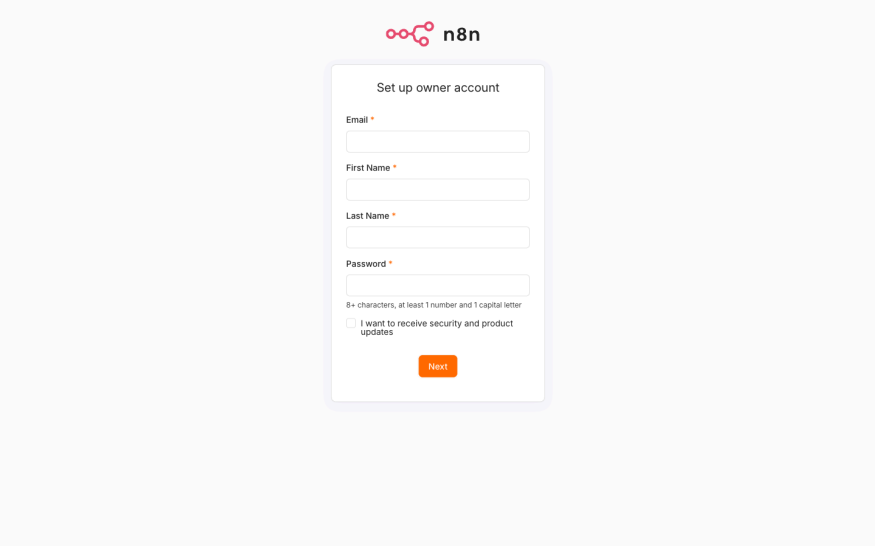
Wait for the line reading `Editor is now accessible via: http://localhost:5678/`. Press `Ctrl+C` to stop following the logs — that stops the log stream, not the containers.

---

6. Create Your Owner Account

Open http://localhost:5678 in a browser.

The first screen asks you to set up an owner account: email, first and last name, and a password. This is the account system built into n8n — there is no separate gate in front of the editor, and the first person to reach a fresh instance becomes the owner. Fill it in immediately, before anything else on your network can reach the port.

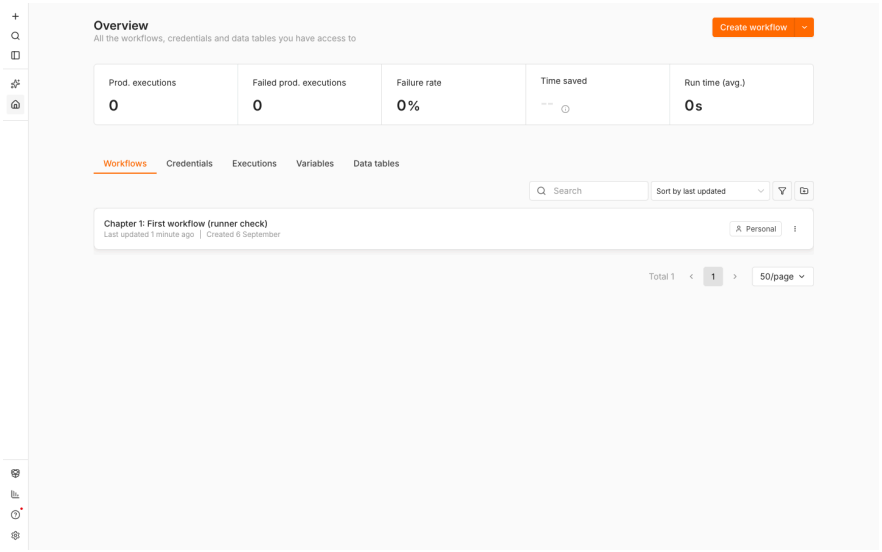


The owner-account form n8n shows on first visit. There is no basic auth in front of it; this account is the authentication.

The owner can later invite other people from **Settings** → **Users** with the roles owner, admin, and member, and can turn on two-factor authentication from the personal settings menu. Single sign-on and Projects with role-based access control are paid-plan features, not part of Community Edition.

---

7. A Two-Minute First Workflow



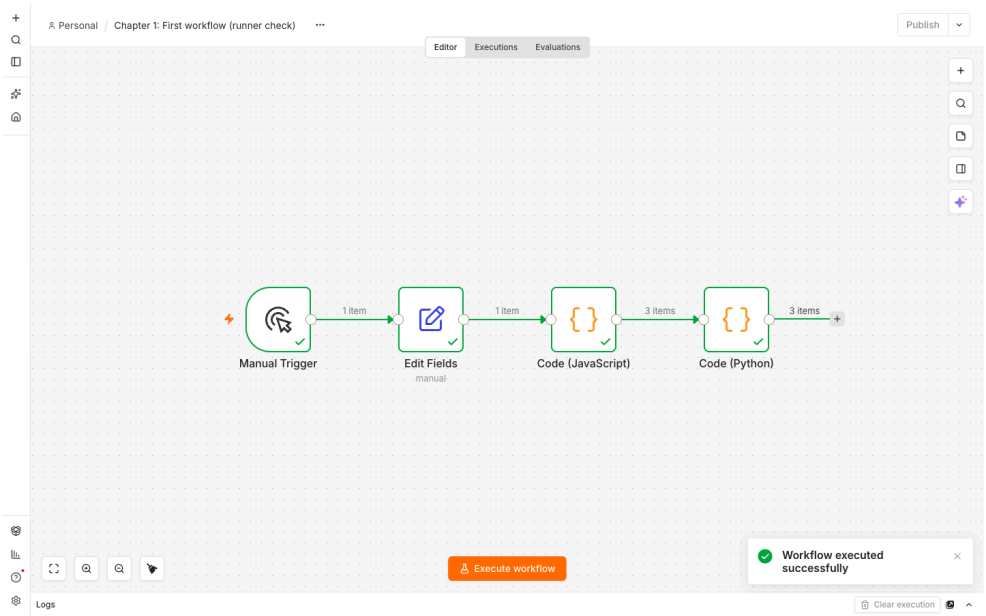
The workflows overview after the owner account exists. Everything in this book starts from here.

This exists to prove one specific thing: that the runners sidecar is connected. If a Code node runs, everything else is wired correctly.

1. Click **Create Workflow**.
2. Add a **Manual Trigger** node. (The old Start node was removed in 2.0; Manual Trigger replaces it.)
3. Add an **Edit Fields (Set)** node after it. Add a string field named `message` with the value `hello from n8n`.
4. Add a **Code** node after that. Leave the language as JavaScript and replace the body with:

```
const items = $input.all();
return [{ json: { echoed: items[0].json.message, itemCount: items.length } }];
```

5. Click **Execute workflow**. The Code node should return your message and a count of `1`.



6. Now change the Code node's language to **Python (Native)** and replace the body with one line:

```
return [{"json": {"engine": "python", "message": _input.first()["json"]["message"]}}]
```

7. Execute again. Same result, different runtime.

Both runs went to the `runners` container. If step 5 fails with a message about no runner being available, jump to the troubleshooting section — the auth token is almost always the cause.

Save the workflow. It persists in Postgres, so it survives restarts.

---

8. Where the Data Lives, and What to Back Up

The stack declares three named volumes. Docker manages them; they are not folders inside `~/n8n`.

```
docker volume ls
docker volume inspect n8n_postgres_data
```

Compose prefixes volume names with the project name, which defaults to the folder name — so `n8n_postgres_data` if your folder is `n8n`.

- `postgres_data` — workflows, credentials, executions, users. Everything that matters.
- `n8n_data` — `/home/node/.n8n`, which holds instance settings and filesystem-mode binary data.
- `n8n_files` — `/home/node/.n8n-files`, the only host path nodes may read or write by default, because n8n 2.0 sets `N8N_RESTRICT_FILE_ACCESS_TO` to that directory.

Two things need backing up: the Postgres volume and `N8N_ENCRYPTION_KEY`. One without the other is useless. Restore the database without the key and every stored credential decrypts to garbage.

Dump the database to a file in your project folder:

```
docker compose exec -T postgres pg_dump -U n8n n8n > n8n-backup-$(date +%F).sql
```

Restore into a fresh stack, with the same `.env` in place:

```
docker compose exec -T postgres psql -U n8n -d n8n < n8n-backup-2026-09-05.sql
```

Keep a copy of `N8N_ENCRYPTION_KEY` somewhere that is not the laptop — a password manager entry is fine. The Backup, Restore, and Disaster Recovery chapter covers backups properly.

---

9. Updating n8n

Updating means editing one thing: the pinned tag, on **both** images.

```
# in compose.yml, change n8nio/n8n:2.38.1 and n8nio/runners:2.38.1
# to the new version, keeping the two identical
docker compose pull
docker compose up -d
```

Compose recreates only the containers whose image changed. Your volumes, and therefore your workflows and credentials, are untouched. Database schema migrations run automatically when the new n8n container starts; watch `docker compose logs -f n8n` and let them finish.

Back up before a major version bump. If you would rather track releases than pick versions, the tag `stable` exists and always points at the current stable release — but then you no longer know what you are running, which is why the rest of this book pins.

---

10. Stopping and Removing

Stop the stack, keeping all data:

```
docker compose down
```

Start it again with `docker compose up -d`. Nothing is lost.

Delete everything, including the volumes:

```
docker compose down -v
```

`-v` removes the named volumes. Your workflows, credentials, and users are gone and cannot be recovered without a backup. There is no confirmation prompt.

---

11. Why Not SQLite Here

n8n runs perfectly well on SQLite, and for a single-user laptop it would be one less container. This book uses Postgres anyway.

The reason is continuity. Postgres is the production database for n8n — it is what queue mode requires, what every VPS and Kubernetes chapter uses, and what you would eventually migrate to. If you start on SQLite you learn one file, then throw it away and learn another. Starting on Postgres costs about 300 MB of RAM on your laptop and saves you a migration.

If you do choose SQLite for a throwaway experiment, remove the `postgres` service and all six `DB_*` variables; n8n falls back to SQLite in `/home/node/.n8n`. Note that SQLite now runs on the pooled driver only, tuned with `DB_SQLITE_POOL_SIZE`: the docs list the default as `0` (rollback journal), so set `DB_SQLITE_POOL_SIZE=2` explicitly to get the pooled WAL driver. MySQL and MariaDB are not options at all — support was dropped in 2.0.

---

12. Troubleshooting Common Issues

Port 5678 is already in use. `docker compose up -d` fails with `address already in use`. Find the offender, or just move n8n to another port by changing the mapping to `"5679:5678"` and setting `WEBHOOK_URL` and `N8N_EDITOR_BASE_URL` to `http://localhost:5679/` to match. The right side of the mapping is the port inside the container and does not change.

```
docker compose down
lsof -i :5678       # macOS and Linux
```

Everything is slow on Windows. Almost always a project folder on `/mnt/c`. Move it into your WSL home directory and run `docker compose up -d` again. Check with `pwd` inside the Ubuntu shell: you want `/home/<you>/n8n`, not `/mnt/c/Users/...`. The same applies to any bind mount you add later.

"Your n8n server is configured to use a secure cookie" when opening it from another device. You reached the editor over plain HTTP on a non-localhost address, such as `http://192.168.1.20:5678`. n8n refuses to send its session cookie over an insecure origin. You can turn that check off by adding to the n8n service:

```
      N8N_SECURE_COOKIE: "false"
```

Understand the tradeoff: the session cookie now travels unencrypted across your network, and anyone on that network can capture it and become you. It is acceptable for a few minutes of testing on a home LAN. It is not acceptable on shared Wi-Fi and never in production — for real access from other machines, use HTTPS behind a proxy (Reverse Proxy and HTTPS: Caddy, Nginx, and Traefik) or a tunnel (Exposing a Private Instance: Cloudflare Tunnel, ngrok, and SSH).

Code nodes fail, or the runner never connects. Check the sidecar's logs:

```
docker compose logs runners
```

An authentication or handshake error means `N8N_RUNNERS_AUTH_TOKEN` does not match between the two services. Both read it from the same `.env` variable, so the usual causes are a stray quote, a trailing space, or editing `.env` without recreating the containers. Compose only re-reads `.env` on `up`:

```
docker compose up -d --force-recreate
```

If the logs show a connection refused to `http://n8n:5679`, confirm `N8N_RUNNERS_BROKER_LISTEN_ADDRESS: 0.0.0.0` is set on the n8n service — the broker binds to localhost otherwise and the sidecar cannot reach it.

Apple Silicon. Nothing to do. `n8nio/n8n` and `n8nio/runners` are multi-arch images with native `arm64` builds. If you see a "platform mismatch" warning, you have added a `platform:` line somewhere — remove it. Do not force `linux/amd64`; emulation is slower and buys nothing.

n8n restarts in a loop. Read `docker compose logs n8n` from the top. The two common causes are Postgres credentials in `.env` that disagree with an existing `postgres_data` volume (the volume keeps the password from first creation), and a changed `N8N_ENCRYPTION_KEY`. For the first, either restore the original password or `docker compose down -v` and start clean.

Further reading

- Install with Docker Compose: https://docs.n8n.io/deploy/host-n8n/install-options/install-using-docker-compose
- Set up task runners: https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners

- Deployment environment variables: https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/deployment
- n8n 3.0 breaking changes: https://docs.n8n.io/changelog/v30-breaking-changes
- n8n 2.0 breaking changes: https://docs.n8n.io/changelog/v20-breaking-changes

---

Chapter 2: Docker Deep Dive: docker run, Volumes, Secrets, and Air-Gapped Installs

Overview

“n8n on Your Laptop with Docker Compose” got you a working instance with a `compose.yml` file and a named volume. This chapter goes deeper into the Docker pieces that chapter glossed over: raw `docker run` invocations, named volumes versus host bind mounts, the UID 1000 permission problem bind mounts create, Docker secrets, resource limits, and running n8n with no internet access at all. None of it is required to run n8n — it matters once you outgrow the laptop setup: multiple environments, an untrusted host, or a server with no outbound network access.

Prerequisites

- Docker Engine installed, ideally after working through “n8n on Your Laptop with Docker Compose.”
- Comfortable at a terminal; this chapter is command-line only.
- `openssl` for generating keys (ships with macOS and most Linux distributions).

1. docker run with Named Volumes and –env-file

A named volume is Docker-managed storage; you never see its host path directly, and Docker sets its ownership correctly for you. Create one and use `--env-file` instead of stacking `-e` flags:

```
docker volume create n8n_data
```

`.env`:

```
N8N_ENCRYPTION_KEY=<paste output of: openssl rand -hex 32>
GENERIC_TIMEZONE=America/Los_Angeles
```

```
docker run -d \
  --name n8n \
  --env-file .env \
  -p 127.0.0.1:5678:5678 \
  -v n8n_data:/home/node/.n8n \
  --restart unless-stopped \
  n8nio/n8n:2.38.1
```

`--env-file` keeps secrets out of your shell history, unlike a stack of `-e` flags. Generate the key once and keep it; lose it, and existing credentials become unreadable.

2. Host Bind Mounts and the UID 1000 Permission Issue

A bind mount points a container path at a specific host directory instead of Docker-managed storage. Take the `docker run` command from section 1 and replace the named volume with a host path:

```
mkdir -p ~/n8n_data
# -v ~/n8n_data:/home/node/.n8n   instead of   -v n8n_data:/home/node/.n8n
```

The `n8nio/n8n` image runs as the `node` user, UID 1000, not root. If `~/n8n_data` is created by root (common when a setup script runs under `sudo`), n8n can't write to it and fails to start, with permission errors in `docker logs n8n`. Fix the ownership before starting it:

```
sudo chown -R 1000:1000 ~/n8n_data
```

Named volumes don't have this problem because Docker creates them with the right ownership on first write. Prefer named volumes unless you specifically need to browse or edit the files from the host.

3. GENERIC_TIMEZONE and Container Clock

Containers default to UTC. If a workflow uses a Cron trigger or displays timestamps, add both variables:

```
-e GENERIC_TIMEZONE=America/Los_Angeles -e TZ=America/Los_Angeles
```

`GENERIC_TIMEZONE` controls n8n's own scheduling; `TZ` controls the Node.js process and any shell commands a Code node runs.

4. Docker Secrets with _FILE Variables

Any environment variable ending in `_FILE` tells n8n to read the value from a file path instead of the variable itself, keeping secrets out of `docker inspect` and CI logs. The two most useful are `N8N_ENCRYPTION_KEY_FILE` and `DB_POSTGRESDB_PASSWORD_FILE`.

Create the secret files with restrictive permissions:

```
mkdir -p ~/n8n_secrets
openssl rand -hex 32 > ~/n8n_secrets/encryption_key
chmod 600 ~/n8n_secrets/encryption_key
```

Mount the file read-only and point the `_FILE` variable at the mounted path, adding these two flags to the `docker run` command from section 1 in place of `--env-file`:

```
-v ~/n8n_secrets/encryption_key:/run/secrets/n8n_encryption_key:ro \
-e N8N_ENCRYPTION_KEY_FILE=/run/secrets/n8n_encryption_key \
```

In a `compose.yml`, the equivalent uses a top-level `secrets:` block, which Compose mounts read-only at `/run/secrets/<name>` automatically — the same mechanism Docker Swarm uses for its managed secrets store, and the practical equivalent on a single host without Swarm:

```
services:
  n8n:
    environment:
      N8N_ENCRYPTION_KEY_FILE: /run/secrets/n8n_encryption_key
      DB_POSTGRESDB_PASSWORD_FILE: /run/secrets/db_password
    secrets: [n8n_encryption_key, db_password]

secrets:
  n8n_encryption_key: { file: ./secrets/encryption_key }
  db_password: { file: ./secrets/db_password }
```

5. Resource Limits

On a shared host, cap what n8n and its runner can consume so one runaway workflow doesn't starve everything else. Add `--memory 1g --cpus 1.5` to any `docker run` command above. In `compose.yml`, the equivalent keys are `mem_limit` and `cpus`, which work without Swarm — the `deploy.resources` key, by contrast, is Swarm-only and is silently ignored by plain `docker compose up`:

```
services:
  n8n:
    image: n8nio/n8n:2.38.1
    mem_limit: 1g
    cpus: 1.5
```

If n8n is killed unexpectedly under load, check `docker inspect n8n --format='{{.State.OOMKilled}}'` before assuming it's a bug in a workflow.

6. Multi-Container Stacks with Postgres and Redis

Wiring Postgres and Redis by hand with `docker run` and a user-defined network (`docker network create n8n-net`, then `--network n8n-net` on each container) works for a quick test, but past two or three containers it's error-prone to reproduce. The full stack — Postgres, n8n, workers, and Redis for queue mode — is what "Docker Compose in Queue Mode: Main, Workers, Webhooks, and Runners" builds properly. Use `docker run` for single-container experiments; switch to Compose once you're running more than one.

7. Air-Gapped Installs

On a host with no outbound internet access, pull and save the images on a connected machine first. You need both the n8n image and the task runners sidecar image, at the same version:

```
docker pull n8nio/n8n:2.38.1
docker pull n8nio/runners:2.38.1
docker save n8nio/n8n:2.38.1 n8nio/runners:2.38.1 -o n8n-images.tar
```

Copy the tarball to the air-gapped host, then load it:

```
docker load -i n8n-images.tar
```

`docker load` restores the exact tags, so `docker run` and `compose.yml` files need no changes. Save and load `postgres:18` and `redis:7-alpine` the same way if you need them too.

8. Backing Up Named Volumes

Because named volumes don't expose a host path, back them up by running a throwaway container that mounts the volume alongside a host directory:

```
docker run --rm \
  -v n8n_data:/data \
  -v "$(pwd)/backups":/backup \
  alpine \
  sh -c "tar czf /backup/n8n_data_$(date +%F).tar.gz -C /data ."
```

Restore into a fresh or emptied volume the same way, swapping the command for `sh -c "tar xzf /backup/n8n_data_2026-09-05.tar.gz -C /data"`.

`n8n_data` holds workflows and credentials, but the credentials are only readable with the matching `N8N_ENCRYPTION_KEY`. Back up the key separately — a password manager, not the same tarball.

9. Troubleshooting Common Issues

Issue	Cause	Remedy
Permission denied on startup	Bind-mounted directory not owned by UID 1000	`sudo chown -R 1000:1000 <host path>`, or use a named volume
Credentials unreadable after a restore	`N8N_ENCRYPTION_KEY` differs from the one used at backup time	Restore the original key; a lost key means unrecoverable credentials
`_FILE` variable reads empty	Secret file not mounted at the path the `_FILE` variable points to	Confirm the mount path and the variable's value match exactly
Container killed under load	Memory limit too low for the workload	Raise it, or confirm with `docker inspect --format='{{.State.OOMKilled}}'` first
`docker load` succeeds but `run` still pulls	Tag in your run command doesn't match the loaded tag	Run `docker images` and compare the tag string exactly

Further reading

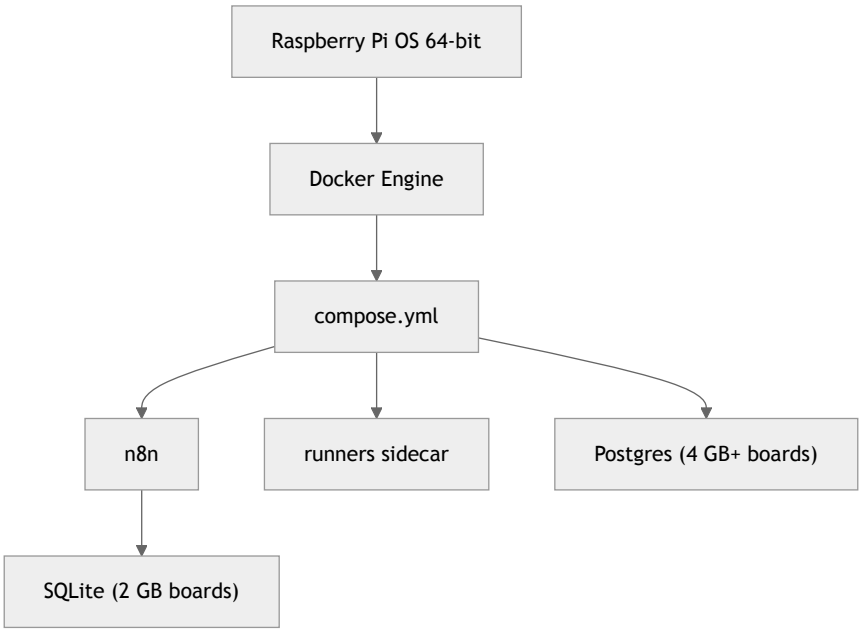
- https://docs.n8n.io/deploy/host-n8n/install-options/install-using-docker-compose
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/deployment

---

Chapter 3: Raspberry Pi and ARM Boards

Overview

n8n's Docker image is multi-arch: the same `n8nio/n8n:2.38.1` tag works on a Raspberry Pi as it does on an x86 server, and Docker pulls the right variant automatically. There's no separate `-arm64` tag to remember. This chapter covers what's actually Pi-specific: a mandatory 64-bit OS, sizing your stack to the board's RAM, the task runners sidecar's memory cost, auto-start on boot, backups, and the two failure modes unique to SD-card hardware.



Diagram

Prerequisites

- Raspberry Pi 4 or 5 (Pi 3 and earlier have 1 GB RAM, too little for anything but the lightest workloads)
- **Raspberry Pi OS, 64-bit.** The 32-bit build cannot run current Docker images for n8n; verify with `uname -m`, which must report `aarch64`, not `armv7l`.
- A high-endurance SD card or, better, a USB-attached SSD
- SSH access and basic Linux CLI familiarity

1. Preparing the OS

```
sudo apt update && sudo apt upgrade -y
```

Confirm the architecture before going further:

```
uname -m
# must print: aarch64
```

If it prints `armv7l`, you're on the 32-bit build and need to reflash with the 64-bit Raspberry Pi OS image before continuing.

For headless setups, enable SSH and Wi-Fi through `raspi-config`:

```
sudo raspi-config
```

2. Installing Docker Engine

The official convenience script detects ARM and installs the correct build. It also installs the Compose v2 plugin, so there's no separate Compose install step.

```
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh
sudo usermod -aG docker $USER
```

Log out and back in for the group change to take effect, then verify:

```
docker --version
docker compose version
docker run hello-world
```

3. Sizing Your Stack to the Board's RAM

The task runners sidecar (`n8nio/runners`) that executes Code nodes is a second container running alongside n8n; budget for it, not just for n8n itself.

- **4 GB RAM or more**: run the full reference stack — Postgres, n8n, and the runners sidecar. This is the setup to use if the Pi is doing anything beyond a hobby project.
- **2 GB RAM**: skip Postgres. Use SQLite with a small connection pool, and keep the runners sidecar — Code nodes still need it, and `internal` runner mode is not recommended for anything long-running:

```
DB_SQLITE_POOL_SIZE=2
```

Below 2 GB, expect to fight the OOM killer regardless of configuration; add swap (see Troubleshooting) or move to a 4 GB board.

4. Defining Your Compose Stack

```
mkdir ~/n8n-pi && cd ~/n8n-pi
nano compose.yml
```

Find the Pi's LAN address with `hostname -I`, then write `.env`:

```
# Generate with: openssl rand -hex 32
N8N_ENCRYPTION_KEY=replace-with-64-hex-characters
N8N_RUNNERS_AUTH_TOKEN=replace-with-a-long-random-string

POSTGRES_PASSWORD=replace-me
GENERIC_TIMEZONE=America/Los_Angeles

# The Pi's LAN IP, e.g. 192.168.1.42 — from: hostname -I
N8N_HOST=<pi-ip>
```

```
openssl rand -hex 32   # N8N_ENCRYPTION_KEY
openssl rand -hex 32   # N8N_RUNNERS_AUTH_TOKEN
```

4 GB+ board, with Postgres:

```
services:
  postgres:
    image: postgres:18
    restart: unless-stopped
    environment:
      POSTGRES_USER: n8n
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: n8n
    volumes:
      - postgres_data:/var/lib/postgresql
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U n8n -d n8n"]
      interval: 10s
      timeout: 5s
      retries: 5

  n8n:
    image: n8nio/n8n:2.38.1
    restart: unless-stopped
    ports:
      - "5678:5678"
    environment:
      N8N_ENCRYPTION_KEY: ${N8N_ENCRYPTION_KEY}
      DB_TYPE: postgresdb
      DB_POSTGRESDB_HOST: postgres
      DB_POSTGRESDB_DATABASE: n8n
      DB_POSTGRESDB_USER: n8n
      DB_POSTGRESDB_PASSWORD: ${POSTGRES_PASSWORD}
      N8N_HOST: ${N8N_HOST}
      N8N_PROTOCOL: http
      WEBHOOK_URL: http://${N8N_HOST}:5678/
      N8N_EDITOR_BASE_URL: http://${N8N_HOST}:5678/
      N8N_SECURE_COOKIE: "false"
      GENERIC_TIMEZONE: ${GENERIC_TIMEZONE}
      TZ: ${GENERIC_TIMEZONE}
      N8N_RUNNERS_MODE: external
      N8N_RUNNERS_BROKER_LISTEN_ADDRESS: 0.0.0.0
      N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
      N8N_NATIVE_PYTHON_RUNNER: "true"
      N8N_DEFAULT_BINARY_DATA_MODE: filesystem
      EXECUTIONS_DATA_PRUNE: "true"
      EXECUTIONS_DATA_MAX_AGE: 336
    volumes:
      - n8n_data:/home/node/.n8n
      - n8n_files:/home/node/.n8n-files
    depends_on:
      postgres:
        condition: service_healthy
```

```
  runners:
    image: n8nio/runners:2.38.1
    restart: unless-stopped
    environment:
      N8N_RUNNERS_TASK_BROKER_URI: http://n8n:5679
      N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
      N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT: 15
    depends_on:
      - n8n

volumes:
  postgres_data:
  n8n_data:
  n8n_files:
```

`N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT` (seconds) idles the runner's internal process down after no task has run, which matters more here than on a server — it's the difference between the runner sidecar sitting at rest and holding memory continuously. Lower it to reclaim RAM between sparse Code node executions; raise it if workflows call Code nodes in bursts and the shutdown/restart cycle is adding latency.

A Pi on your LAN is reached at `http://<pi-ip>:5678`, plain HTTP, not through a hostname or a certificate — that's why `N8N_PROTOCOL` is `http` here instead of the `https` the reference stack uses, and why `N8N_SECURE_COOKIE` has to be `"false"`. n8n normally marks its session cookie `Secure`, which browsers refuse to send back over a plain HTTP connection; setting it to `"false"` gives up that protection so the editor's login actually works on the LAN. The cookie is sent in the clear either way at that point, so this only matters against someone who can already observe your LAN traffic — for most home networks that's a reasonable trade, but it's not a setting to carry onto anything internet-facing. The way to remove the trade-off entirely is to put a reverse proxy with a real (or self-signed, trusted-on-your-devices) TLS certificate in front of the Pi and switch back to `https`; see "Reverse Proxy and HTTPS: Caddy, Nginx, and Traefik" for how to set one up.

2 GB board: drop the `postgres` service and its `depends_on`, and replace the `DB_POSTGRESDB_*` block in `n8n`'s environment with:

```
      N8N_ENCRYPTION_KEY: ${N8N_ENCRYPTION_KEY}
      DB_SQLITE_POOL_SIZE: 2
      N8N_HOST: ${N8N_HOST}
      N8N_PROTOCOL: http
      WEBHOOK_URL: http://${N8N_HOST}:5678/
      N8N_EDITOR_BASE_URL: http://${N8N_HOST}:5678/
      N8N_SECURE_COOKIE: "false"
      GENERIC_TIMEZONE: ${GENERIC_TIMEZONE}
      TZ: ${GENERIC_TIMEZONE}
      N8N_RUNNERS_MODE: external
      N8N_RUNNERS_BROKER_LISTEN_ADDRESS: 0.0.0.0
      N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
      N8N_NATIVE_PYTHON_RUNNER: "true"
      N8N_DEFAULT_BINARY_DATA_MODE: filesystem
      EXECUTIONS_DATA_PRUNE: "true"
      EXECUTIONS_DATA_MAX_AGE: 336
```

5. Launching the Stack

```
docker compose up -d
docker compose ps
docker compose logs -f n8n
```

Wait for `postgres` to report `healthy` (4 GB variant) before worrying about `n8n`'s state.

6. Accessing n8n

```
http://<raspberry-pi-ip>:5678
```

The first visit prompts you to create the owner account — there's no default login to change.

7. Auto-Start on Boot

You don't need a custom systemd unit for this. `get-docker.sh` already enables Docker's own systemd service, so it comes up on every boot:

```
sudo systemctl is-enabled docker
# enabled
```

Combined with `restart: unless-stopped` on every service in `compose.yml`, that's sufficient: Docker starts on boot, and each container with that restart policy comes back up with it, in the state it was left in. Confirm after a real reboot, not just `docker compose restart`:

```
sudo reboot
# after it comes back:
docker compose -f ~/n8n-pi/compose.yml ps
```

8. Backups and Restore

```
docker run --rm \
  -v n8n-pi_n8n_data:/data \
  -v ~/n8n-pi/backups:/backup \
  alpine sh -c "tar czf /backup/n8n_data_$(date +%F).tar.gz -C /data ."
```

Restore into a stopped stack the same way, reversing `tar czf` to `tar xzf`. If you're running Postgres, back it up too — see "Docker Compose with PostgreSQL" for the `pg_dump` approach, which is more reliable than archiving the raw data directory.

For a full-device recovery point, image the entire SD card or SSD from another Linux machine with `dd`, with the Pi powered off.

9. Offline / Air-Gapped Deployment

Same process as any other host — see “Docker Deep Dive: docker run, Volumes, Secrets, and Air-Gapped Installs” for the full explanation. On the connected machine:

```
docker pull n8nio/n8n:2.38.1
docker pull n8nio/runners:2.38.1
docker pull postgres:18
docker save n8nio/n8n:2.38.1 n8nio/runners:2.38.1 postgres:18 -o n8n-pi-images.tar
```

Copy the tarball to the Pi and load it:

```
docker load -i n8n-pi-images.tar
```

10. Troubleshooting Common Issues

Problem	Cause	Solution
Image won't pull, or runs but crashes immediately	32-bit OS	Confirm `uname -m` reports `aarch64`; reflash with the 64-bit image if not
Out of memory / containers killed	RAM oversubscribed by Postgres + runners on a 2 GB board	Drop to SQLite with `DB_SQLITE_POOL_SIZE=2`, or move to a 4 GB+ board
Persistent volume permission denied	Bind-mounted directory not owned by UID 1000	`sudo chown -R 1000:1000 <path>`, or use a named volume instead
n8n and the editor feel sluggish, high disk wait	SD card wear from constant small writes (SQLite, execution data, logs)	Move the `n8n_data` volume, and especially Postgres's `postgres_data` volume, to a USB-attached SSD; SD cards degrade under this write pattern faster than under normal file storage
Postgres volume corrupted after a power loss	SD card write caching combined with an ungraceful shutdown	Move Postgres to a USB SSD; add a UPS, or always shut down with `docker compose down` before pulling power
Stack doesn't survive a reboot	`restart: unless-stopped` missing from a service, or Docker's systemd unit disabled	Add the restart policy to every service; `sudo systemctl enable docker`
Runner sidecar using more memory than expected	`N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT` set too high for sparse workloads	Lower it so the runner's process exits between executions instead of idling
Editor won't log in, or the login redirect loops back to the sign-in page	Accessing over plain `http://<pi-ip>:5678` without `N8N_SECURE_COOKIE: "false"`, so the browser drops the `Secure` session cookie	Set `N8N_SECURE_COOKIE: "false"` for LAN-only HTTP access, or put a reverse proxy with HTTPS in front and use `N8N_PROTOCOL: https` instead

Further reading

- https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners
- https://docs.n8n.io/deploy/host-n8n/install-options/install-using-docker-compose
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/deployment

---

Chapter 4: Task Runners and Safe Code Execution

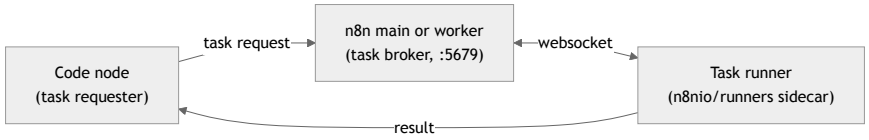
Overview

The Code node runs code that a workflow author typed into a browser. Since n8n 2.0, every Code node execution runs on a task runner. Task runners exist so that this code runs somewhere other than inside the n8n process.

The n8n docs are blunt about the stakes: task runners are the only isolation layer between user-provided code and n8n. Without them, or with a runner in internal mode, anyone who can edit a workflow could potentially read your database, your encryption key, your stored credentials, and your environment variables. That is not a theoretical concern on a shared instance. It is the reason this chapter exists.

In this chapter you will:

1. Learn what a task runner is and how the runner, broker, and requester fit together
2. Compare internal mode and external mode, and see exactly what each one exposes
3. Run the `n8nio/runners` sidecar next to n8n with a complete compose stack
4. Enable native Python in the Code node
5. Wire runners correctly in queue mode, where each worker needs its own sidecar
6. Add npm and pip packages by extending the runners image and allowlisting modules
7. Understand what 2.0 broke inside the Code node and why
8. Set resource limits, health checks, and the Kubernetes equivalent



Diagram

Prerequisites

- A working n8n stack from the reference `compose.yml` in this book, running `n8nio/n8n:2.38.1`
- Docker Engine with Compose v2 (`docker compose`)
- A long random string for `N8N_RUNNERS_AUTH_TOKEN` , generated once and stored in `.env`
- Shell access to the host so you can read container logs

1. What a Task Runner Is and Why It Exists

A task runner is a separate process whose only job is to execute one piece of user code and hand back the result. The Code node does not run JavaScript or Python itself any more. It packages the code and the input items into a task, sends the task away, and waits for an answer.

The point is blast radius. n8n holds your credential store and the `N8N_ENCRYPTION_KEY` that decrypts it. If arbitrary code runs inside that process, a sandbox escape reaches everything the process can reach. If the code runs in a different container, with a different filesystem, no database connection, and no credentials in its environment, an escape reaches a container that has nothing worth stealing.

Since 2.0 you no longer opt in. `N8N_RUNNERS_ENABLED` is deprecated and you can drop it. What you still choose is the *mode*, and that choice is the whole security story.

---

2. The Three Components

The feature has three parts, and it helps to name them because the log lines do.

Task requester. The Code node, running inside n8n. It submits a task and waits for the result.

Task broker. The n8n instance itself — main or worker — acts as the broker. It listens on port `5679` (`N8N_RUNNERS_BROKER_PORT`) and coordinates between requesters and runners. It does not execute anything.

Task runner. The process that actually executes the code. In external mode this lives in the sidecar container, started and supervised by a launcher application.

Runners connect *outward* to the broker over a websocket and authenticate with a shared secret. The broker then hands them tasks over that same connection. The runner sends a heartbeat every `N8N_RUNNERS_HEARTBEAT_INTERVAL` seconds (default 30); if it stops, the broker kills the task and the runner restarts.

The direction of that connection matters for your network design. The runner dials the broker, so the broker's port must be reachable from the runner, and the runner needs no inbound path from n8n at all.

---

3. Internal Mode Versus External Mode

`N8N_RUNNERS_MODE` takes two values, and the default is the wrong one for production.

`internal` (the default). n8n launches the runner as a child process of itself. Same host, same `uid` and `gid`, same filesystem, same environment. The n8n process manages its lifecycle. This is insecure by design and the docs say so. Code that escapes the runner's sandbox lands with exactly the access n8n has: the credentials table, the encryption key, the Postgres connection, the mounted volumes. Internal mode is acceptable on a throwaway instance holding mock data and nothing else.

`external`. A launcher application in a separate container starts runners on demand and supervises them. The runner container has its own filesystem and its own environment. It never receives `N8N_ENCRYPTION_KEY`, never receives `DB_POSTGRESDB_PASSWORD`, and cannot open a socket to Postgres unless you put it on that network yourself. An escape gets an attacker a shell in a container whose contents are a Node.js runtime, a Python runtime, and the code they already wrote.

External mode is what the rest of this chapter configures. One practical note: the runners image does **not** support file-based configuration. Variables with a `_FILE` suffix are ignored there, so pass the auth token as a plain environment variable from your `.env`.

---

4. The `n8nio/runners` Sidecar

The sidecar image contains the launcher, the JavaScript runner, and the Python runner. **Its version must match the n8n image version exactly.** These two images speak a private protocol; a mismatch produces connection errors or tasks that never complete.

`.env`:

```
# Generate with: openssl rand -hex 32
N8N_ENCRYPTION_KEY=replace-with-64-hex-characters
N8N_RUNNERS_AUTH_TOKEN=replace-with-a-long-random-string

POSTGRES_USER=n8n
POSTGRES_PASSWORD=replace-me
POSTGRES_DB=n8n

N8N_HOST=n8n.example.com
GENERIC_TIMEZONE=America/Los_Angeles
```

`compose.yml`:

```
services:
  postgres:
    image: postgres:18
    restart: unless-stopped
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    volumes:
      - postgres_data:/var/lib/postgresql
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER} -d ${POSTGRES_DB}"]
      interval: 10s
      timeout: 5s
      retries: 5

  n8n:
    image: n8nio/n8n:2.38.1
    restart: unless-stopped
    ports:
      - "127.0.0.1:5678:5678"
    environment:
      N8N_ENCRYPTION_KEY: ${N8N_ENCRYPTION_KEY}
      DB_TYPE: postgresdb
      DB_POSTGRESDB_HOST: postgres
      DB_POSTGRESDB_PORT: 5432
      DB_POSTGRESDB_DATABASE: ${POSTGRES_DB}
      DB_POSTGRESDB_USER: ${POSTGRES_USER}
      DB_POSTGRESDB_PASSWORD: ${POSTGRES_PASSWORD}
      N8N_HOST: ${N8N_HOST}
      N8N_PROTOCOL: https
      WEBHOOK_URL: https://${N8N_HOST}/
      N8N_EDITOR_BASE_URL: https://${N8N_HOST}/
      N8N_PROXY_HOPS: 1
      GENERIC_TIMEZONE: ${GENERIC_TIMEZONE}
      TZ: ${GENERIC_TIMEZONE}
      N8N_RUNNERS_MODE: external
      N8N_RUNNERS_BROKER_LISTEN_ADDRESS: 0.0.0.0
      N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
      N8N_NATIVE_PYTHON_RUNNER: "true"
      N8N_DEFAULT_BINARY_DATA_MODE: filesystem
      EXECUTIONS_DATA_PRUNE: "true"
      EXECUTIONS_DATA_MAX_AGE: 336
      N8N_DIAGNOSTICS_ENABLED: "false"
    volumes:
      - n8n_data:/home/node/.n8n
      - n8n_files:/home/node/.n8n-files
    depends_on:
      postgres:
        condition: service_healthy
```

```
  runners:
    image: n8nio/runners:2.38.1
    restart: unless-stopped
    environment:
      N8N_RUNNERS_TASK_BROKER_URI: http://n8n:5679
      N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
      N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT: 15
    depends_on:
      - n8n

volumes:
  postgres_data:
  n8n_data:
  n8n_files:
```

What each runner variable does:

- `N8N_RUNNERS_MODE=external` tells n8n not to fork a child process and to wait for a runner to connect instead.
- `N8N_RUNNERS_BROKER_LISTEN_ADDRESS=0.0.0.0` is required. The broker binds `127.0.0.1` by default, which is unreachable from a second container. Without this, the runner will never connect.
- `N8N_RUNNERS_AUTH_TOKEN` must be byte-identical on both services. It is the only thing standing between your broker port and anything else that can reach it.
- `N8N_RUNNERS_TASK_BROKER_URI=http://n8n:5679` uses the compose service name. If you rename the `n8n` service, change this too.
- `N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT=15` stops an idle runner process after 15 seconds. The launcher restarts it when the next task arrives. Set `0` to keep runners resident, which trades memory for a small latency saving on the first Code node after a quiet period.

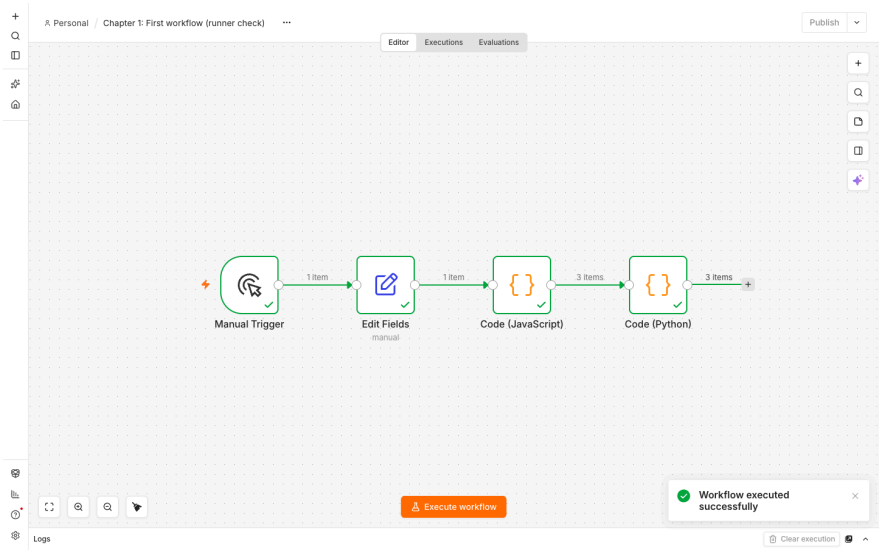
Bring it up and confirm the connection:

```
docker compose up -d
docker compose logs -f runners
```

You are looking for the launcher reporting a successful connection to the broker. Then open any workflow, add a Code node with `return [{json: {ok: true}}]`, and run it.

---

5. Native Python



A workflow with a JavaScript Code node and a native Python Code node, both executed on the external runner. The runner container logs show the `launcher:js` and `launcher:py` processes accepting the tasks.

n8n 2.0 removed the Pyodide-based Python Code node and replaced it with a native Python runner. Set `N8N_NATIVE_PYTHON_RUNNER=true` on the n8n container, as in the stack above. Python in the Code node works **only** with external mode.

Two differences will bite you when migrating old workflows:

- The native runner does not provide the Pyodide built-ins. `_input` is gone, and so is dot-access notation on items. Read items through the documented native API instead.
- In a Python Code *tool* — the Code node attached to an AI agent — the agent's query arrives as `_query`.

Everything else is a normal CPython process, which is why `numpy` and `pandas` are now realistic (see section 7) and why the performance is nothing like Pyodide's.

---

6. Runners in Queue Mode

Queue mode changes the rule, and this is the most common misconfiguration in production stacks.

Every n8n process that acts as a broker needs its own runners sidecar. Each worker is a broker for the Code nodes it executes. One shared runners container pointed at main does not serve the workers.

So:

- Each `n8n-worker` gets its own `runners` service pointing at that worker's port 5679.
- Main also needs a sidecar **unless** `OFFLOAD_MANUAL_EXECUTIONS_TO_WORKERS=true`, which hands manual editor executions to workers so main never runs a Code node. Running with offloading disabled is not recommended for production anyway.

A worker plus its runner, as a fragment of the queue-mode stack:

```yaml
n8n-worker:
  image: n8nio/n8n:2.38.1
  restart: unless-stopped
  command: worker --concurrency=10
  environment:
    N8N_ENCRYPTION_KEY: ${N8N_ENCRYPTION_KEY}
    DB_TYPE: postgresdb
    DB_POSTGRESDB_HOST: postgres
    DB_POSTGRESDB_DATABASE: ${POSTGRES_DB}
    DB_POSTGRESDB_USER: ${POSTGRES_USER}
    DB_POSTGRESDB_PASSWORD: ${POSTGRES_PASSWORD}
    EXECUTIONS_MODE: queue
    QUEUE_BULL_REDIS_HOST: redis
    N8N_DEFAULT_BINARY_DATA_MODE: database
    N8N_RUNNERS_MODE: external
    N8N_RUNNERS_BROKER_LISTEN_ADDRESS: 0.0.0.0
    N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
    N8N_NATIVE_PYTHON_RUNNER: "true"
    GENERIC_TIMEZONE: ${GENERIC_TIMEZONE}
    TZ: ${GENERIC_TIMEZONE}
  depends_on:
    postgres:
      condition: service_healthy

worker-runners:
  image: n8nio/runners:2.38.1
  restart: unless-stopped
  environment:
    N8N_RUNNERS_TASK_BROKER_URI: http://n8n-worker:5679
    N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
    N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT: 15
  depends_on:
    - n8n-worker
```

If you scale workers with `docker compose up -d --scale`, this pairing breaks — the service name resolves to several containers and runners will attach arbitrarily. Define workers as distinct services (`n8n-worker-1`, `n8n-worker-2`) each with its own sidecar, or move to an orchestrator that models sidecars properly.

---

7. Adding npm and pip Dependencies

The runners image ships a deliberately minimal set of modules, and the launcher configuration file is locked down. Adding a package takes two steps: install it in the image, then allowlist it for the Code node. Doing only the first gives you a package the Code node still refuses to import.

Extend the image:

```
FROM n8nio/runners:2.38.1
USER root
RUN cd /opt/runners/task-runner-javascript && pnpm add moment uuid
RUN cd /opt/runners/task-runner-python && uv pip install numpy pandas
COPY n8n-task-runners.json /etc/n8n-task-runners.json
USER runner
```

The launcher reads its configuration from `/etc/n8n-task-runners.json`. You can bake it in with `COPY`, as above, or mount it at that path from the host:

```
    volumes:
      - ./n8n-task-runners.json:/etc/n8n-task-runners.json:ro
```

`n8n-task-runners.json`:

```
{
  "task-runners": [
    {
      "runner-type": "javascript",
      "env-overrides": {
        "NODE_FUNCTION_ALLOW_BUILTIN": "crypto",
        "NODE_FUNCTION_ALLOW_EXTERNAL": "moment,uuid"
      }
    },
    {
      "runner-type": "python",
      "env-overrides": {
        "PYTHONPATH": "/opt/runners/task-runner-python",
        "N8N_RUNNERS_STDLIB_ALLOW": "json",
        "N8N_RUNNERS_EXTERNAL_ALLOW": "numpy,pandas"
      }
    }
  ]
}
```

The four allowlists are:

- `NODE_FUNCTION_ALLOW_BUILTIN` — Node.js built-in modules
- `NODE_FUNCTION_ALLOW_EXTERNAL` — third-party JavaScript packages
- `N8N_RUNNERS_STDLIB_ALLOW` — Python standard library modules
- `N8N_RUNNERS_EXTERNAL_ALLOW` — third-party Python packages

Keep these lists short and specific. Every entry is a capability you are granting to anyone who can edit a workflow. Allowlisting `child_process` or `fs` hands back most of what external mode took away.

Build and pin your own tag, and rebuild it every time you bump n8n:

```
docker build -t registry.example.com/n8n-runners:2.38.1 .
```

---

8. What 2.0 Broke, and the Escape Hatch You Should Not Use

Two changes catch people upgrading.

`$evaluateExpression()` no longer works in the Code node. Code node executions now run on runners in secure mode, and secure mode disables evaluating strings as code — which is precisely the mechanism expressions rely on. Calls return `null` or throw. Expressions in ordinary node fields, such as Edit Fields (Set), are unaffected. Rewrite the logic in plain JavaScript.

`N8N_RUNNERS_INSECURE_MODE=true` disables all security measures in the runner for compatibility with modules that need those insecure features. The docs discourage it for production and so does this book: turning it on undoes the isolation you set up in section 4 while leaving the sidecar in place, which is worse than obvious, because the stack still looks correct.

`N8N_BLOCK_ENV_ACCESS_IN_NODE` now defaults to `true`. Code nodes cannot read `process.env`. If a workflow depended on that, the migration path is to move the value into a credential. Setting `N8N_BLOCK_ENV_ACCESS_IN_NODE=false` restores the old behaviour and re-exposes every environment variable on the process to anyone who can edit a workflow.

---

9. Limits, Health Checks, and Kubernetes

The sidecar runs untrusted code, so cap what it can consume. A runaway loop should hit a limit, not the host's OOM killer.

`N8N_RUNNERS_MAX_CONCURRENCY` and `N8N_RUNNERS_MAX_OLD_SPACE_SIZE` are both **n8n instance** environment variables per the docs, not runner variables — add them to the `n8n` service's environment block:

```
      N8N_RUNNERS_MAX_CONCURRENCY: 5
      N8N_RUNNERS_MAX_OLD_SPACE_SIZE: 1024
```

The runners sidecar itself only needs its connection settings, a resource ceiling, and a health check:

```
  runners:
    image: n8nio/runners:2.38.1
    restart: unless-stopped
    environment:
      N8N_RUNNERS_TASK_BROKER_URI: http://n8n:5679
      N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
      N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT: 15
    deploy:
      resources:
        limits:
          cpus: "2.0"
          memory: 2G
    healthcheck:
      test: ["CMD-SHELL", "wget -q -O- http://127.0.0.1:5680/healthz || exit 1"]
      interval: 30s
      timeout: 5s
      retries: 3
    depends_on:
      - n8n
```

The launcher serves `/healthz` on port `5680` (`N8N_RUNNERS_LAUNCHER_HEALTH_CHECK_PORT`); the broker serves `/healthz` on `5679`. If your image has no `wget`, drop the `healthcheck` block and probe port 5680 from outside instead. To be clear about where each setting lives: `N8N_RUNNERS_MAX_CONCURRENCY` (default 5) and `N8N_RUNNERS_TASK_TIMEOUT` (default 300 seconds) are set on the **n8n** container, not the runner; so is `N8N_RUNNERS_MAX_OLD_SPACE_SIZE`, which caps the Node.js heap (in MB) that the runner is launched with.

On Kubernetes, put the launcher in the **same pod** as the n8n main or worker container. They share the pod network, so `N8N_RUNNERS_TASK_BROKER_URI=http://127.0.0.1:5679` and you can leave the broker on its default listen address. Give the sidecar its own `resources.limits` and a `livenessProbe` on `5680`. If you prefer a separate Deployment, keep `N8N_RUNNERS_BROKER_LISTEN_ADDRESS=0.0.0.0`, expose 5679 through a Service, and point the runners at it — but you lose the one-runner-per-broker guarantee, so the sidecar pattern is the safer default.

---

10. Troubleshooting Common Issues

Issue	Cause	Solution
Runner never connects; broker logs show nothing	`N8N_RUNNERS_AUTH_TOKEN` differs between the two services	Compare with `docker compose exec n8n printenv N8N_RUNNERS_AUTH_TOKEN` and the same on `runners`. A trailing space or unexported `.env` value is the usual cause
Runner logs “connection refused” to port 5679	Broker still bound to `127.0.0.1`	Set `N8N_RUNNERS_BROKER_LISTEN_ADDRESS=0.0.0.0` on the n8n container and recreate it
Runner connects, then drops repeatedly	Version mismatch between `n8nio/n8n` and `n8nio/runners`	Pin both to the same tag (`2.38.1`) and recreate both containers
“Code node timed out”	Task exceeded `N8N_RUNNERS_TASK_TIMEOUT` (300s), or no runner was free within `N8N_RUNNERS_TASK_REQUEST_TIMEOUT` (60s)	Raise the timeout, raise `N8N_RUNNERS_MAX_CONCURRENCY`, or add a second sidecar. Also check `N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT` isn’t fighting a very slow first task
Code node hangs forever in queue mode	A worker has no runners sidecar	Give every worker its own sidecar pointing at that worker’s `:5679`
Manual runs hang, scheduled runs work	Main has no sidecar and manual executions aren’t offloaded	Set `OFFLOAD_MANUAL_EXECUTIONS_TO_WORKERS=true`, or add a sidecar for main
Python: `ModuleNotFoundError` for an installed package	Package installed but not allowlisted	Add it to `N8N_RUNNERS_EXTERNAL_ALLOW` (or `N8N_RUNNERS_STDLIB_ALLOW`) in `/etc/n8n-task-runners.json` and restart the sidecar
JS: “Cannot find module”	Same, on the JavaScript side	Add it to `NODE_FUNCTION_ALLOW_EXTERNAL`, and confirm `pnpm add` ran in `/opt/runners/task-runner-javascript`
Python Code node unavailable in the editor	`N8N_NATIVE_PYTHON_RUNNER` unset, or mode is `internal`	Set `N8N_NATIVE_PYTHON_RUNNER=true` and `N8N_RUNNERS_MODE=external` on n8n
`_input` or dot access throws in Python	Pyodide built-ins removed in 2.0	Rewrite against the native Python API; in Code tools use `_query`
`$evaluateExpression()` returns `null`	Secure mode disables string evaluation	Rewrite as plain JavaScript. Do not enable `N8N_RUNNERS_INSECURE_MODE`
`process.env` is empty in a Code node	`N8N_BLOCK_ENV_ACCESS_IN_NODE` defaults to `true` since 2.0	Move the value into a credential rather than flipping the flag
Token set via `N8N_RUNNERS_AUTH_TOKEN_FILE` is ignored	The runners image has no file-based configuration	Pass the plain variable from `.env`

Further reading

- https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/task-runners
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/security/harden-task-runners
- https://docs.n8n.io/changelog/v20-breaking-changes
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/enable-queue-mode

- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/queue-mode
- https://docs.n8n.io/integrations/builtin/core-nodes/n8n-nodes-base.code
- https://github.com/n8n-io/task-runner-launcher/blob/main/docs/setup.md

---

Chapter 5: Ubuntu VPS with Docker Compose (the Reference Stack)

Overview

This chapter builds the stack every later chapter in this book assumes: n8n, on Docker, on a hardened Ubuntu VPS, backed by PostgreSQL, with task runners split into their own sidecar, and HTTPS terminated by Traefik. Every setting here matches the reference stack described in the appendix, and every environment variable is explained, not just listed. If a later chapter says “start from the reference stack,” this is the chapter it means.

By the end you will have:

1. A provisioned VPS with key-based SSH and a default-deny firewall
2. Docker Engine and the Compose plugin installed (no standalone `docker-compose` binary)
3. A pinned, production-shaped `compose.yml`: Postgres 18, n8n 2.38.1, an external task-runner sidecar, and Traefik for HTTPS
4. The five environment variables that make webhooks and OAuth callbacks resolve correctly behind a proxy
5. A working knowledge of how to launch, back up, update, and troubleshoot the stack

Diagram

Prerequisites

- A fresh Ubuntu 22.04 or 24.04 VPS with root or sudo access
- A domain name with an A record pointed at the VPS’s IP address
- An SSH key pair on your local machine (`ssh-keygen -t ed25519` if you don’t have one)

---

1. Provisioning and Hardening the VPS

Create a non-root user, install your public key, and lock down SSH before you do anything else. Running this as root, over password auth, on a box with an open firewall is how VPSs get compromised before you've finished reading the changelog.

```
adduser deploy
usermod -aG sudo deploy
rsync --archive --chown=deploy:deploy ~/.ssh /home/deploy
```

Edit `/etc/ssh/sshd_config` and set:

```
PasswordAuthentication no
PermitRootLogin no
```

```
sudo systemctl restart ssh
```

Test the key-based login from a **second** terminal before you close the first one — if the config is wrong, you want a session still open to fix it.

Install UFW and allow only what you need. Traefik will need 80 and 443 for HTTP-to-HTTPS redirects and the Let's Encrypt HTTP-01 challenge:

```
sudo apt update
sudo apt install -y ufw
sudo ufw default deny incoming
sudo ufw default allow outgoing
sudo ufw allow OpenSSH
sudo ufw allow 80/tcp
sudo ufw allow 443/tcp
sudo ufw enable
sudo ufw status verbose
```

Do not open 5678 to the internet. n8n stays reachable only through Traefik.

---

2. Installing Docker Engine and the Compose Plugin

Use Docker's own repository, not the convenience script, so you get pinned package versions and automatic security updates through `apt`:

```
sudo apt install -y ca-certificates curl gnupg
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o
        /etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg

echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg]
       https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt update
sudo apt install -y docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

Add your user to the `docker` group and verify:

```
sudo usermod -aG docker $USER
newgrp docker
docker --version
docker compose version
```

Notice the space: `docker compose`, not `docker-compose`. The standalone Python tool is retired; the Compose plugin ships with Docker Engine and is what every command in this book uses.

---

3. The Reference Stack

Create a project directory and two files.

```
mkdir ~/n8n && cd ~/n8n
```

`.env`:

```
# Generate with: openssl rand -hex 32
N8N_ENCRYPTION_KEY=replace-with-64-hex-characters
N8N_RUNNERS_AUTH_TOKEN=replace-with-a-long-random-string

POSTGRES_USER=n8n
POSTGRES_PASSWORD=replace-me
POSTGRES_DB=n8n

N8N_HOST=n8n.example.com
GENERIC_TIMEZONE=America/Los_Angeles

TRAEFIK_ACME_EMAIL=you@example.com
```

`compose.yml`:

```
services:
  postgres:
    image: postgres:18
    restart: unless-stopped
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    volumes:
      - postgres_data:/var/lib/postgresql
    networks:
      - n8n-net
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER} -d ${POSTGRES_DB}"]
      interval: 10s
      timeout: 5s
      retries: 5

  n8n:
    image: n8nio/n8n:2.38.1
    restart: unless-stopped
    ports:
      - "127.0.0.1:5678:5678"
    environment:
      N8N_ENCRYPTION_KEY: ${N8N_ENCRYPTION_KEY}
      DB_TYPE: postgresdb
      DB_POSTGRESDB_HOST: postgres
      DB_POSTGRESDB_PORT: 5432
      DB_POSTGRESDB_DATABASE: ${POSTGRES_DB}
      DB_POSTGRESDB_USER: ${POSTGRES_USER}
      DB_POSTGRESDB_PASSWORD: ${POSTGRES_PASSWORD}
      N8N_HOST: ${N8N_HOST}
      N8N_PROTOCOL: https
      WEBHOOK_URL: https://${N8N_HOST}/
      N8N_EDITOR_BASE_URL: https://${N8N_HOST}/
      N8N_PROXY_HOPS: 1
      GENERIC_TIMEZONE: ${GENERIC_TIMEZONE}
      TZ: ${GENERIC_TIMEZONE}
      N8N_RUNNERS_MODE: external
      N8N_RUNNERS_BROKER_LISTEN_ADDRESS: 0.0.0.0
      N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
      N8N_NATIVE_PYTHON_RUNNER: "true"
      N8N_DEFAULT_BINARY_DATA_MODE: filesystem
      EXECUTIONS_DATA_PRUNE: "true"
      EXECUTIONS_DATA_MAX_AGE: 336
      N8N_DIAGNOSTICS_ENABLED: "false"
    volumes:
      - n8n_data:/home/node/.n8n
      - n8n_files:/home/node/.n8n-files
```

```
    networks:
      - n8n-net
    depends_on:
      postgres:
        condition: service_healthy
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.n8n.rule=Host(`${N8N_HOST}`)"
      - "traefik.http.routers.n8n.entrypoints=websecure"
      - "traefik.http.routers.n8n.tls.certresolver=leresolver"
      - "traefik.http.services.n8n.loadbalancer.server.port=5678"

  runners:
    image: n8nio/runners:2.38.1
    restart: unless-stopped
    environment:
      N8N_RUNNERS_TASK_BROKER_URI: http://n8n:5679
      N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
      N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT: 15
    networks:
      - n8n-net
    depends_on:
      - n8n

networks:
  n8n-net:
    driver: bridge

volumes:
  postgres_data:
  n8n_data:
  n8n_files:
```

A two-line note on SQLite: n8n still supports SQLite (pooled driver, `DB_SQLITE_POOL_SIZE`) for a single-user laptop install, but it has no place on a VPS you intend to keep running — use Postgres as above.

The proxy variables, one line each

- `N8N_HOST` — the public hostname n8n believes it is served from; used to build links and OAuth redirect URIs.
- `N8N_PROTOCOL=https` — tells n8n it is being served over TLS even though the container itself only speaks plain HTTP.
- `WEBHOOK_URL` — the base URL n8n hands to third-party services when they register a webhook; if it's wrong, incoming webhooks 404.
- `N8N_EDITOR_BASE_URL` — the base URL the editor's browser assets and OAuth callback pages resolve against.
- `N8N_PROXY_HOPS=1` — tells n8n to trust one layer of `X-Forwarded-*` headers from the proxy in front of it, so it sees the real client IP and scheme instead of the proxy's.

Get any of these wrong and the symptom is always the same: the UI loads, but webhooks silently fail or OAuth callbacks bounce to the wrong URL.

---

4. Custom Networks and Isolation

The `n8n-net` bridge network above already keeps Postgres and the runners sidecar unreachable from outside the Docker host — nothing publishes their ports. Traefik joins the same network implicitly through Compose's service discovery (see below), so it can reach n8n by service name without any port being exposed to the internet except 80 and 443 on the Traefik container itself.

If you later add services that shouldn't talk to Postgres at all — a monitoring agent, for instance — give them their own network and only attach the services that need cross-talk to more than one.

---

5. Reverse Proxy with Traefik (HTTPS)

Traefik reads container labels and configures itself automatically — no manually written vhost file, and no separate reverse-proxy config to keep in sync when you rename a service. Add it to `compose.yml`:

```yaml
  traefik:
    image: traefik:v3.3.6
    restart: unless-stopped
    command:
      - "--providers.docker=true"
      - "--providers.docker.exposedbydefault=false"
      - "--entrypoints.web.address=:80"
      - "--entrypoints.websecure.address=:443"
      - "--entrypoints.web.http.redirections.entrypoint.to=websecure"
      - "--entrypoints.web.http.redirections.entrypoint.scheme=https"
      - "--certificatesresolvers.leresolver.acme.httpchallenge=true"
      - "--certificatesresolvers.leresolver.acme.httpchallenge.entrypoint=web"
      - "--certificatesresolvers.leresolver.acme.email=${TRAEFIK_ACME_EMAIL}"
      - "--certificatesresolvers.leresolver.acme.storage=/letsencrypt/acme.json"
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro
      - traefik_letsencrypt:/letsencrypt
    networks:
      - n8n-net
```

Add `traefik_letsencrypt` to the `volumes:` block at the bottom of the file alongside the other named volumes.

This is the whole configuration. The `leresolver` certificate resolver requests and renews a Let's Encrypt certificate for whatever hostname shows up in a router rule — which is exactly the `Host( ${N8N_HOST} )` rule already on the `n8n` service's labels above. There is no separate `traefik.toml` and no `acme.json` you create by hand; Traefik writes its own inside the named volume, and Docker manages the file's permissions for you.

You do not need to configure anything for the editor's websocket connection. Traefik proxies the raw TCP stream of an HTTP connection once it's routed, including the `Upgrade` and `Connection` headers a websocket handshake depends on — there's no separate websocket entrypoint or flag.

Bring the resolver up gradually the first time: watch `docker compose logs -f traefik` while you request the certificate, since a typo in `N8N_HOST` or a DNS record that hasn't propagated yet will show up there as an ACME error, not as a Traefik crash.

---

6. Launching and Managing the Stack

```
docker compose up -d
docker compose ps
docker compose logs -f n8n
```

Visit `https://n8n.example.com` (substitute your real hostname) and create the owner account on first load — that account, not any environment variable, is your authentication.

To scale n8n horizontally you need queue mode, not more replicas of this stack — a bare `docker compose up -d --scale n8n=3` will give you three containers writing executions to the same Postgres database with no work distribution between them. See the chapter “Docker Compose in Queue Mode: Main, Workers, Webhooks, and Runners” for the real path to concurrency.

---

7. Auto-Start and Updates

`restart: unless-stopped` already brings every container back after a host reboot or crash; you do not need a systemd unit for that alone. If you still want systemd to own the stack's lifecycle explicitly — for ordering against other units, or so `systemctl status` reflects it — wrap it:

```
# /etc/systemd/system/n8n.service
[Unit]
Description=n8n Docker Compose Stack
Requires=docker.service
After=docker.service

[Service]
Type=oneshot
RemainAfterExit=true
WorkingDirectory=/home/deploy/n8n
ExecStart=/usr/bin/docker compose up -d
ExecStop=/usr/bin/docker compose down
User=deploy

[Install]
WantedBy=multi-user.target
```

```
sudo systemctl enable --now n8n
```

For updates, resist the temptation to run a Watchtower-style auto-updater against this stack. n8n and the runners sidecar must stay on the same version, and an unattended upgrade gives you no chance to read the release notes for a migration step. Instead:

1. Check the n8n changelog for breaking changes since your current version.
2. Bump both pinned tags in `compose.yml` — `n8nio/n8n:X.Y.Z` and `n8nio/runners:X.Y.Z` — to the same new version. They must always match.
3. Pull and recreate:

   ```
   docker compose pull
   docker compose up -d
   ```

4. Watch `docker compose logs -f n8n` for migration output, then check **Settings → Migration Report** in the editor for anything the automated migration couldn't resolve on its own.

---

8. Backups and Restores

Back up three things together, on the same schedule, or a restore will leave you with credentials n8n can't decrypt: the Postgres database, the `n8n_files` volume (binary data and any restricted-file-access uploads), and your `.env` file's `N8N_ENCRYPTION_KEY`.

Compose reads `.env` to fill in `${POSTGRES_USER}` and `${POSTGRES_DB}` inside `compose.yml`, but it never exports those values into your shell — `$POSTGRES_USER` is empty on the host unless you set it there yourself. The reference stack's `.env` sets `POSTGRES_USER=n8n` and `POSTGRES_DB=n8n`, so the commands below use those values literally. If you changed either in `.env`, substitute your own.

```
mkdir -p ~/n8n/backup

docker compose exec -T postgres pg_dump -U n8n -d n8n \
  | gzip > ~/n8n/backup/postgres_$(date +%F).sql.gz

docker run --rm \
  -v n8n_n8n_files:/data \
  -v ~/n8n/backup:/backup \
  alpine sh -c "tar czf /backup/n8n_files_$(date +%F).tar.gz -C /data ."

cp ~/n8n/.env ~/n8n/backup/env_$(date +%F).bak
```

(Volume names are prefixed with the project directory name by default — check `docker volume ls` if `n8n_n8n_files` doesn't match yours.)

To restore, recreate the stack against an empty `postgres_data` volume, then:

```
gunzip < postgres_2026-09-01.sql.gz | docker compose exec -T postgres psql -U n8n -d n8n

docker run --rm \
  -v n8n_n8n_files:/data \
  -v ~/n8n/backup:/backup \
  alpine sh -c "tar xzf /backup/n8n_files_2026-09-01.tar.gz -C /data"
```

Restore the same `N8N_ENCRYPTION_KEY` into `.env` before starting n8n — a different key means every stored credential decrypts to garbage.

---

9. Troubleshooting Common Issues

Issue	Cause	Solution
`docker compose` not found	Compose plugin not installed, or old standalone binary shadowing it	`sudo apt install docker-compose-plugin`; remove any `/usr/local/bin/docker-compose`
Traefik never gets a certificate	Port 80 blocked, DNS not propagated, or wrong `N8N_HOST`	Check `docker compose logs traefik`; confirm the A record with `dig`; confirm UFW allows 80/tcp
Webhooks return 404 or hit the wrong URL	`WEBHOOK_URL` or `N8N_HOST` don't match the real hostname	Fix the `.env` values, `docker compose up -d` to recreate n8n
Editor loads but loses connection / reconnect loop	`N8N_PROXY_HOPS` unset or wrong, breaking the client's view of the connection	Set `N8N_PROXY_HOPS=1` for a single proxy in front of n8n
Restore has unreadable credentials	`N8N_ENCRYPTION_KEY` differs from the key used when the data was created	Restore the original key from your backup of `.env`
Task runner never connects	`N8N_RUNNERS_AUTH_TOKEN` mismatched between `n8n` and `runners` services	Confirm both services read the same `.env` value; recreate both containers
Disk filling up	Executions never pruned	Confirm `EXECUTIONS_DATA_PRUNE=true` and `EXECUTIONS_DATA_MAX_AGE` are set; `docker system df` to find orphaned volumes

Further reading

- https://docs.n8n.io/deploy/host-n8n/install-options/install-using-docker-compose
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/deployment

---

Chapter 6: Reverse Proxy and HTTPS: Caddy, Nginx, and Traefik

Overview

n8n never terminates its own TLS in this book — a reverse proxy does, and n8n sits behind it on plain HTTP over the Docker network. This chapter covers the two proxies you're most likely to hand-configure on a VPS, Caddy and Nginx, in order of how much can go wrong: Caddy first, because it is nearly impossible to misconfigure into a broken websocket connection; then Nginx, because it's the one you'll meet most often in existing infrastructure and it rewards knowing exactly which headers matter. Traefik is covered in full in the chapter "Ubuntu VPS with Docker Compose (the Reference Stack)," since it only makes sense wired directly into that Compose file — this chapter just points to it.

Whichever proxy you choose, the failure mode of getting it wrong is identical: the editor loads fine, and then webhooks silently 404 or OAuth callbacks redirect to the wrong host. That failure isn't a proxy bug — it's five n8n environment variables, and every section below assumes you've set them.

Prerequisites

- n8n running and reachable on `127.0.0.1:5678` (or a Docker service name, if the proxy is a container on the same network)
- A domain name with an A/AAAA record pointed at the server
- sudo privileges

---

1. The Variables Every Proxy Needs

Set these on the n8n container regardless of which proxy sits in front of it:

```
N8N_HOST=n8n.example.com
N8N_PROTOCOL=https
WEBHOOK_URL=https://n8n.example.com/
N8N_EDITOR_BASE_URL=https://n8n.example.com/
N8N_PROXY_HOPS=1
```

`N8N_HOST` and the two URL variables tell n8n what hostname to put in webhook registrations and OAuth redirect URIs; `N8N_PROTOCOL=https` tells it those URLs should be `https://` even though the proxy is talking to it over plain HTTP; `N8N_PROXY_HOPS=1` tells n8n to trust one hop of `X-Forwarded-*` headers so it sees the real client IP and scheme rather than the proxy's. Skip any of these and n8n still boots and the UI still loads — only webhooks and OAuth break, which is exactly the kind of failure that's miserable to debug after the fact.

---

2. Caddy: The Low-Effort Option

If you don't already have Nginx or Traefik opinions, use Caddy. It requests and renews Let's Encrypt certificates automatically with no separate Certbot step, and it forwards websocket upgrade headers without being told to.

Install it from Caddy's own repository:

```
sudo apt install -y debian-keyring debian-archive-keyring apt-transport-https curl
curl -1sLf 'https://dl.cloudsmith.io/public/caddy/stable/gpg.key' | sudo gpg --dearmor -o
        /usr/share/keyrings/caddy-stable-archive-keyring.gpg
curl -1sLf 'https://dl.cloudsmith.io/public/caddy/stable/debian.deb.txt' | sudo tee
        /etc/apt/sources.list.d/caddy-stable.list
sudo apt update && sudo apt install -y caddy
```

The entire configuration lives in `/etc/caddy/Caddyfile`:

```
n8n.example.com {
    reverse_proxy 127.0.0.1:5678
}
```

```
sudo systemctl reload caddy
```

That's it — three lines. Caddy obtains the certificate on first request, renews it automatically, redirects HTTP to HTTPS, and passes `Upgrade` / `Connection` headers through by default, so the editor's websocket connection just works. If you need a longer timeout for long-running workflow executions, add one line:

```
n8n.example.com {
    reverse_proxy 127.0.0.1:5678 {
        transport http {
            read_timeout 3600s
        }
    }
}
```

If you want fine-grained cipher control, rate limiting, or you're integrating with infrastructure that already standardizes on Nginx, keep reading.

---

3. Installing Nginx and Certbot

```
sudo apt update && sudo apt install -y nginx certbot python3-certbot-nginx
sudo systemctl status nginx
```

On a minimal cloud image you may need the Universe repository first:

```
sudo add-apt-repository universe
sudo apt update
```

---

4. Configuring the Firewall (UFW)

```
sudo ufw allow OpenSSH
sudo ufw allow 'Nginx Full'
sudo ufw enable
sudo ufw status
```

`Nginx Full` opens both 80 and 443. If SSH runs on a non-standard port, allow that port explicitly instead of `OpenSSH`.

---

5. Nginx Server Blocks

5.1 HTTP to HTTPS redirect

`/etc/nginx/sites-available/n8n`:

```
server {
    listen 80;
    server_name n8n.example.com;
    return 301 https://$host$request_uri;
}
```

5.2 HTTPS proxy to n8n

Append to the same file. Every line in the `location /` block earns its place — this is the block that has to get the websocket upgrade, the forwarded headers, upload size, and execution timeout right, or the editor either won't load or will silently misbehave:

```
server {
    listen 443 ssl http2;
    server_name n8n.example.com;

    ssl_certificate /etc/letsencrypt/live/n8n.example.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/n8n.example.com/privkey.pem;
    include /etc/letsencrypt/options-ssl-nginx.conf;
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem;

    add_header Strict-Transport-Security "max-age=31536000; includeSubDomains" always;
    add_header X-Frame-Options DENY;
    add_header X-Content-Type-Options nosniff;

    # n8n accepts file uploads and binary data through webhooks and the editor
    client_max_body_size 50M;

    location / {
        proxy_pass http://127.0.0.1:5678;

        # Required for the editor's websocket connection
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";

        # Required so N8N_PROXY_HOPS sees the real client and scheme
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;

        # Long-running workflow executions hold the connection open
        proxy_read_timeout 3600s;
        proxy_connect_timeout 60s;
    }

    error_page 502 503 504 /50x.html;
    location = /50x.html {
        root /usr/share/nginx/html;
    }
}
```

`X-Forwarded-Proto` matters as much as the websocket headers here: it's what `N8N_PROXY_HOPS` uses to tell n8n the original request was HTTPS, even though Nginx is talking to n8n over plain HTTP on the loopback interface. Without it, n8n can generate `http://` links behind a proxy that's actually serving HTTPS.

Enable and test:

```
sudo ln -s /etc/nginx/sites-available/n8n /etc/nginx/sites-enabled/
sudo nginx -t
```

```
sudo systemctl reload nginx
```

To load-balance more than one n8n instance, define an `upstream` block and point `proxy_pass` at it — but a second n8n process needs queue mode, covered in the chapter “Docker Compose in Queue Mode: Main, Workers, Webhooks, and Runners,” not just a second Nginx target.

---

6. Obtaining and Renewing Certificates

```
sudo certbot --nginx -d n8n.example.com
sudo certbot renew --dry-run
```

Certbot edits the server block to point at the issued certificate and installs a systemd timer for renewal. For a wildcard certificate, use the DNS-01 challenge instead:

```
sudo certbot -d example.com -d '*.example.com' --manual --preferred-challenges dns certonly
```

---

7. Advanced TLS Settings

```
ssl_protocols TLSv1.2 TLSv1.3;
ssl_ciphers 'HIGH:!aNULL:!MD5:!3DES';
ssl_prefer_server_ciphers on;
ssl_session_cache shared:SSL:10m;
ssl_session_timeout 10m;
ssl_stapling on;
ssl_stapling_verify on;
resolver 8.8.8.8 8.8.4.4 valid=300s;
resolver_timeout 5s;
```

OCSP stapling lets clients verify certificate revocation status without a separate round trip to the CA, which is worth the two lines it costs.

---

8. Rate Limiting

```
http {
    limit_req_zone $binary_remote_addr zone=n8n:10m rate=5r/s;

    server {
        location / {
            limit_req zone=n8n burst=10 nodelay;
            proxy_pass http://127.0.0.1:5678;
            # ... headers from section 5.2
        }
    }
}
```

Set the rate against expected editor and webhook traffic, not against a guess — five requests per second per client IP is a reasonable starting point for a single-user or small-team instance. If a specific webhook path receives high-volume, legitimate traffic, give it its own `location /webhook/` block with a separate, looser zone.

---

9. Logging and Rotation

```
/var/log/nginx/access.log
/var/log/nginx/error.log
```

`/etc/logrotate.d/nginx` ships with Nginx and rotates both daily by default; confirm it matches your retention needs:

```
/var/log/nginx/*.log {
    daily
    missingok
    rotate 14
    compress
    delaycompress
    notifempty
    create 0640 www-data adm
    sharedscripts
    postrotate
        [ -s /run/nginx.pid ] && kill -USR1 `cat /run/nginx.pid`
    endscript
}
```

---

10. Traefik

If you're running n8n as a Docker Compose stack rather than a bare process, Traefik is usually less work than Nginx because it reads container labels instead of a separate config file, and it renews certificates itself. The full working example — static configuration, the Let's Encrypt resolver, and the router labels on the n8n service — is in the chapter "Ubuntu VPS with Docker Compose (the Reference Stack)." It isn't repeated here because it only makes sense attached to that Compose file.

11. Authentication in Front of the Proxy

None of the proxy configurations above add authentication in front of n8n, and that's intentional. The owner account you create on first login *is* the authentication, backed by n8n's own user management, roles, and optional 2FA. An extra credential bolted onto the proxy is a second, weaker gate that doesn't integrate with n8n's session or webhook paths.

If you want a second layer — for a staging instance, or defense in depth in front of the editor specifically — put it above the proxy, not woven into it: Cloudflare Access (or another zero-trust proxy) in front of the hostname, or an OAuth-proxy sidecar (oauth2-proxy against your identity provider) that gates the editor path before traffic ever reaches Nginx or Caddy. Either approach authenticates the human before n8n's own login screen ever appears, without touching webhook endpoints that external services need to reach unauthenticated.

---

12. Troubleshooting Common Issues

Issue	Cause	Solution
Editor loads but disconnects/reconnects constantly	Missing `Upgrade` / `Connection` headers or `proxy_http_version 1.1`	Add both headers and the HTTP/1.1 directive to the `location /` block (Nginx); Caddy and Traefik do this by default
Webhooks 404 or point at the wrong host	`WEBHOOK_URL` / `N8N_HOST` don't match the real public hostname	Fix the five proxy variables in section 1 and restart n8n
n8n generates `http://` links behind HTTPS	Missing `X-Forwarded-Proto` or `N8N_PROXY_HOPS` unset	Add `proxy_set_header X-Forwarded-Proto $scheme;` and set `N8N_PROXY_HOPS=1`
Large file uploads fail with 413	`client_max_body_size` too low or unset	Raise `client_max_body_size` in the server block
Long executions cut off mid-run	`proxy_read_timeout` too short	Raise it (3600s is a reasonable default) to match your longest expected execution
Nginx fails to reload	Syntax error in the server block	`sudo nginx -t` points at the exact line
Certbot renewal failing	DNS not resolving correctly, or rate-limited	Check `dig` against the A record; `sudo certbot renew --dry-run` for details
Caddy won't get a certificate	Port 80 blocked, or DNS not yet propagated	Confirm UFW allows 80/tcp; wait for DNS propagation and check `journalctl -u caddy`

Further reading

- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/deployment
- https://docs.n8n.io/deploy/host-n8n/install-options/install-using-docker-compose

---

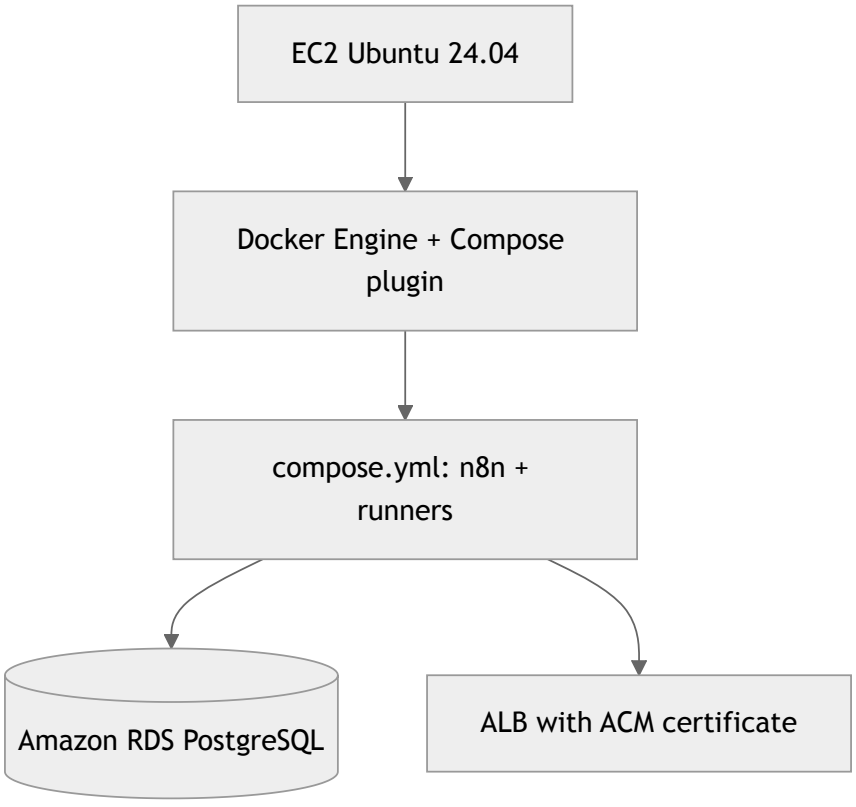
Chapter 7: Deploying on AWS EC2

Overview

An EC2 instance gives you a plain Ubuntu machine with AWS networking, storage, and identity around it. You supply the runtime. From n8n 3.0 (October 2026) the only supported self-hosting runtime is Docker, so this chapter runs n8n as containers from the first step. Nothing here uses a global package install or an external process manager.

In this chapter you will:

1. Provision an EC2 instance and choose an AMI, size, and key pair
2. Configure a security group for SSH, HTTPS, and the n8n port
3. Associate an Elastic IP and point DNS at it
4. Install Docker Engine and the Compose plugin
5. Create the n8n project folder with `.env` and `compose.yml`
6. Point n8n at an Amazon RDS PostgreSQL database
7. Start the stack and make it survive reboots
8. Put an Application Load Balancer in front and terminate TLS on it
9. Ship container logs to CloudWatch and use an IAM role instead of static keys



Diagram

Prerequisites

- An AWS account with permissions for EC2, Elastic IP, security groups, RDS, and IAM
- The AWS CLI configured locally (optional, the console works too)

- An SSH client and a domain name you can add records to
- The background from “n8n on Your Laptop with Docker Compose” if Docker Compose is new to you

---

1. Provisioning the EC2 Instance

Use the **Ubuntu Server 24.04 LTS (x86_64)** AMI. It ships a recent kernel and is the image the Docker apt repository targets.

Size the instance for three containers, not one:

- **t3.small** (2 GiB RAM) is enough to try the stack, not to run it.
- **t3.medium** (2 vCPU, 4 GiB RAM) is the practical minimum for production with an external database.
- **t3.large** or a compute-optimised type if your workflows are CPU heavy.

Give the root volume 30 GB of gp3. Container images, execution data, and binary files all land there.

Create or select a key pair, download the `.pem`, and keep it somewhere you back up. AWS will not show it again.

---

2. Security Group Rules

Protocol	Port	Source	Purpose
TCP	22	Your IP /32	SSH administration
TCP	80, 443	0.0.0.0/0	Public HTTPS, only if TLS terminates on the instance
TCP	5678	ALB security group	n8n, only when an ALB fronts the instance

Port 5678 should never be open to the internet. In the stack below the n8n container publishes to `127.0.0.1` only, so the port is unreachable from outside the instance until you deliberately change that in the load balancer section.

---

3. Elastic IP and DNS

Allocate an Elastic IP and associate it with the instance. Without one, the public address changes whenever the instance stops and starts, and every webhook URL you registered with a third party breaks.

Create an A record for `n8n.example.com` pointing at the Elastic IP. If you add an ALB later, change that record to an alias for the load balancer instead.

---

4. Connecting Over SSH

```
chmod 400 your-key.pem
ssh -i your-key.pem ubuntu@203.0.113.10
```

Confirm you landed where you expect:

```
whoami
lsb_release -ds
```

---

5. Installing Docker Engine and the Compose Plugin

Ubuntu's own `docker.io` package lags and does not include the Compose plugin. Use Docker's apt repository:

```
sudo apt-get update
sudo apt-get install -y ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc
echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
        https://download.docker.com/linux/ubuntu $(. /etc/os-release && echo "$VERSION_CODENAME")
        stable" \
  | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-
        plugin
```

Add your user to the `docker` group so you do not need `sudo` for every command, then start a fresh login shell:

```
sudo usermod -aG docker $USER
newgrp docker
docker compose version
```

Membership in the `docker` group is equivalent to root on the host. Grant it only to accounts you would trust with `sudo`.

---

6. Creating the Project Folder

```
mkdir -p ~/n8n && cd ~/n8n
openssl rand -hex 32   # N8N_ENCRYPTION_KEY
openssl rand -hex 24   # N8N_RUNNERS_AUTH_TOKEN
```

`.env`:

```
# Generate with: openssl rand -hex 32
N8N_ENCRYPTION_KEY=replace-with-64-hex-characters
N8N_RUNNERS_AUTH_TOKEN=replace-with-a-long-random-string

POSTGRES_USER=n8n
POSTGRES_PASSWORD=replace-me
POSTGRES_DB=n8n

# The hostname browsers and webhook callers use.
N8N_HOST=n8n.example.com
GENERIC_TIMEZONE=America/Los_Angeles
```

`N8N_ENCRYPTION_KEY` encrypts every stored credential. Generate it once, back it up with the database, and never rotate it casually: a database restored without its matching key yields unreadable credentials.

`compose.yml`:

```
services:
  postgres:
    image: postgres:18
    restart: unless-stopped
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    volumes:
      - postgres_data:/var/lib/postgresql
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER} -d ${POSTGRES_DB}"]
      interval: 10s
      timeout: 5s
      retries: 5

  n8n:
    image: n8nio/n8n:2.38.1
    restart: unless-stopped
    ports:
      - "127.0.0.1:5678:5678"
    environment:
      N8N_ENCRYPTION_KEY: ${N8N_ENCRYPTION_KEY}
      DB_TYPE: postgresdb
      DB_POSTGRESDB_HOST: postgres
      DB_POSTGRESDB_PORT: 5432
      DB_POSTGRESDB_DATABASE: ${POSTGRES_DB}
      DB_POSTGRESDB_USER: ${POSTGRES_USER}
      DB_POSTGRESDB_PASSWORD: ${POSTGRES_PASSWORD}
      N8N_HOST: ${N8N_HOST}
      N8N_PROTOCOL: https
      WEBHOOK_URL: https://${N8N_HOST}/
      N8N_EDITOR_BASE_URL: https://${N8N_HOST}/
      N8N_PROXY_HOPS: 1
      GENERIC_TIMEZONE: ${GENERIC_TIMEZONE}
      TZ: ${GENERIC_TIMEZONE}
      N8N_RUNNERS_MODE: external
      N8N_RUNNERS_BROKER_LISTEN_ADDRESS: 0.0.0.0
      N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
      N8N_NATIVE_PYTHON_RUNNER: "true"
      N8N_DEFAULT_BINARY_DATA_MODE: filesystem
      EXECUTIONS_DATA_PRUNE: "true"
      EXECUTIONS_DATA_MAX_AGE: 336
      N8N_DIAGNOSTICS_ENABLED: "false"
    volumes:
      - n8n_data:/home/node/.n8n
```

```
      - n8n_files:/home/node/.n8n-files
    depends_on:
      postgres:
        condition: service_healthy

  runners:
    image: n8nio/runners:2.38.1
    restart: unless-stopped
    environment:
      N8N_RUNNERS_TASK_BROKER_URI: http://n8n:5679
      N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
      N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT: 15
    depends_on:
      - n8n

volumes:
  postgres_data:
  n8n_data:
  n8n_files:
```

The `runners` service is not optional decoration. Since n8n 2.0 every Code node executes in a task runner, and the external runner is a sidecar container built from the same version tag as n8n. Keep the two tags in lockstep whenever you upgrade.

The stack has no separate login layer to configure. The first browser visit to the editor creates the owner account, and you add further users, roles, and 2FA from **Settings** → **Users**.

---

7. Using Amazon RDS for PostgreSQL

For production, move the database off the instance. Create an RDS PostgreSQL instance in the same VPC, give it a security group that allows TCP 5432 from the EC2 instance's security group only, and leave public accessibility off.

Download the AWS certificate bundle onto the instance so n8n can verify the server:

```
curl -o ~/n8n/rds-ca.pem https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem
```

Then change the compose file from the version above:

- Delete the whole `postgres` service, the `postgres_data` volume, and the `depends_on` block under `n8n`.
- Add a read-only bind mount to the `n8n` service: `- ./rds-ca.pem:/etc/ssl/rds-ca.pem:ro`.
- Replace the database environment values:

```
    DB_POSTGRESDB_HOST: ${RDS_HOST}
    DB_POSTGRESDB_PORT: 5432
    DB_POSTGRESDB_DATABASE: ${RDS_DATABASE}
    DB_POSTGRESDB_USER: ${RDS_USER}
    DB_POSTGRESDB_PASSWORD: ${RDS_PASSWORD}
    DB_POSTGRESDB_SSL_ENABLED: "true"
    DB_POSTGRESDB_SSL_CA_FILE: /etc/ssl/rds-ca.pem
```

Move `RDS_HOST`, `RDS_DATABASE`, `RDS_USER`, and `RDS_PASSWORD` into `.env` and drop the three `POSTGRES_*` lines. You now get automated backups, point-in-time recovery, and patching from RDS, and the instance becomes disposable.

---

8. Starting the Stack and Surviving Reboots

```
cd ~/n8n
docker compose up -d
docker compose ps
docker compose logs -f n8n
```

Auto-start needs no extra work. Every service carries `restart: unless-stopped`, and Docker's own systemd unit starts at boot, so the stack comes back after a reboot or a crash. Confirm it once:

```
systemctl is-enabled docker
sudo reboot
```

---

9. Fronting the Instance with an Application Load Balancer

An ALB in front of the single instance below is for TLS termination and health checks, not for scaling out. It gives you an ACM certificate and a health check against `/healthz`.

1. Create a target group of type **instance** on port 5678, protocol HTTP, health check path `/healthz`.
2. Create the ALB with an HTTPS:443 listener using an ACM certificate issued in the same region, and an HTTP:80 listener that redirects to HTTPS.
3. Restrict the instance security group so port 5678 accepts traffic from the ALB security group only.

Two changes in `compose.yml`:

- Publish the port on all interfaces so the ALB can reach it: `- "5678:5678"`.
- Set `N8N_HOST` in `.env` to the hostname clients actually use — your DNS alias for the load balancer, not the instance. `WEBHOOK_URL` and `N8N_EDITOR_BASE_URL` follow it automatically.

`N8N_PROXY_HOPS: 1` is already set and is correct behind a single ALB. It tells n8n to trust exactly one layer of `X-Forwarded-For`, so rate limiting and logging see the real client IP. Leave the ALB's default WebSocket support alone — the editor needs it.

Target group stickiness does not make it safe to run more than one n8n instance behind this ALB. Regular-mode n8n instances that share a Postgres database but not a queue will each pick up and execute triggers independently, so more than one instance requires queue mode — see "Docker Compose in Queue Mode: Main, Workers, Webhooks, and Runners" for main, worker, webhook, and runner processes. A multi-main setup (more than one main process) is an Enterprise feature, not something this stack does on its own.

---

10. CloudWatch Logs

Containers write to Docker's JSON log files by default. Send them straight to CloudWatch instead by adding a logging block to the `n8n` and `runners` services:

```
    logging:
      driver: awslogs
      options:
        awslogs-region: us-east-1
        awslogs-group: /n8n/ec2
        awslogs-create-group: "true"
```

Create a metric filter on the string `ERROR` and an alarm that publishes to an SNS topic if you want paging. Prometheus scraping is available too: set `N8N_METRICS: "true"` to expose `/metrics`.

---

11. IAM Roles Instead of Keys

Attach an instance profile rather than putting access keys on the box. Grant it:

- `logs:CreateLogGroup`, `logs:CreateLogStream`, `logs:PutLogEvents` for the log driver above
- `secretsmanager:GetSecretValue` scoped to the secret holding your database password and encryption key

Fetch secrets into `.env` at deploy time with the AWS CLI, keep the file `chmod 600`, and never bake credentials into the compose file or an AMI.

---

12. Updating n8n

Change the two pinned tags in `compose.yml` — `n8nio/n8n` and `n8nio/runners` — to the same new version, then:

```
docker compose pull
docker compose up -d
```

Take an RDS snapshot first. Rolling back means putting the old tags back and repeating the two commands, which only works if the database has not been migrated forward.

---

13. Troubleshooting Common Issues

Issue	Cause	Solution
SSH connection times out	Security group port 22 closed, or your IP changed	Update the SG source to your current /32 and confirm the Elastic IP
ALB target unhealthy	n8n published on `127.0.0.1` only, or wrong health path	Publish `"5678:5678"`, set the health check to `/healthz`
Editor loads but webhooks point at the wrong host	`N8N_HOST` still set to the instance DNS name	Set it to the load balancer hostname, then `docker compose up -d`
`no pg_hba.conf entry` or TLS errors from RDS	SSL not enabled or CA file missing	Set `DB_POSTGRESDB_SSL_ENABLED` and mount the RDS bundle
Credentials unreadable after a restore	Encryption key differs from the one that wrote the data	Restore `N8N_ENCRYPTION_KEY` from backup; there is no recovery without it
Code nodes fail with a broker error	Runner tag drifted from the n8n tag, or tokens differ	Pin both images to the same version and share one `N8N_RUNNERS_AUTH_TOKEN`
Disk fills up	Image layers and execution data accumulate	`docker image prune -a`, and keep `EXECUTIONS_DATA_PRUNE` on

---

Further reading

- https://docs.n8n.io/deploy/host-n8n/install-options/install-using-docker-compose
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/deployment
- https://docs.n8n.io/changelog/v30-breaking-changes
- https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/UsingWithRDS.SSL.html

---

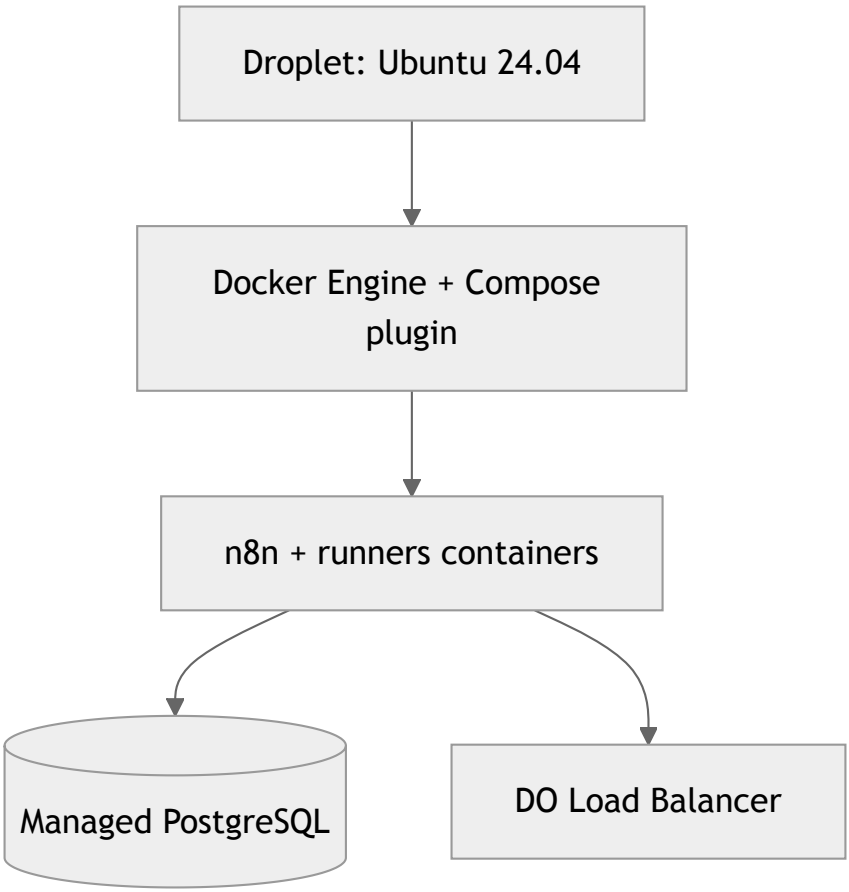
Chapter 8: Deploying on DigitalOcean

Overview

A DigitalOcean Droplet is the cheapest honest way to run a production n8n instance: a fixed monthly price, a reserved IP, an optional managed Postgres, and a load balancer that speaks Let's Encrypt. The runtime is Docker, as it must be for n8n 3.0 and later.

In this chapter you will:

1. Create a Droplet from the console or with `doctl`
2. Layer a Cloud Firewall over UFW
3. Attach a reserved IP and point DNS at it
4. Install Docker Engine and the Compose plugin
5. Adapt the reference stack for DigitalOcean
6. Move the database to DigitalOcean Managed PostgreSQL
7. Add a load balancer, block storage, and snapshots



Diagram

Prerequisites

- A DigitalOcean account, and an API token if you use `doctl`
- An SSH key uploaded to your account

- A domain you can point at a reserved IP
- The compose file from “Deploying on AWS EC2”, which this chapter modifies

---

1. Creating the Droplet

In the console choose **Create** → **Droplets**, image **Ubuntu 24.04 LTS**, and a plan with at least 2 vCPU and 4 GiB RAM. The 1 GiB plans cannot hold n8n, a runner, and Postgres at once. Select your SSH key; never enable password login.

With the CLI:

```
doctl auth init
doctl compute droplet create n8n-prod \
  --size s-2vcpu-4gb \
  --image ubuntu-24-04-x64 \
  --region nyc3 \
  --ssh-keys your-key-fingerprint \
  --enable-monitoring \
  --vpc-uuid your-vpc-uuid
```

Placing the Droplet in a VPC matters later: managed databases are reachable over private networking from inside the same VPC.

---

2. Cloud Firewall and UFW

The Cloud Firewall filters before packets reach the Droplet; UFW is the second layer if a rule is ever mis-scoped.

Cloud Firewall inbound rules:

Port	Source	Purpose
22	Your IP	SSH
80, 443	All IPv4/IPv6	Public HTTPS, when TLS terminates on the Droplet
5678	Load balancer only	n8n, when a load balancer fronts it

```
sudo apt-get update && sudo apt-get install -y ufw
sudo ufw allow OpenSSH
sudo ufw allow 80/tcp
sudo ufw allow 443/tcp
sudo ufw --force enable
```

Do not open 5678 in UFW. The stack binds it to `127.0.0.1`, and the load balancer section changes that deliberately.

---

3. Reserved IP and DNS

Under **Networking** → **Reserved IPs**, assign one to the Droplet and create an A record for `n8n.example.com`. A reserved IP can be remapped to a replacement Droplet during a rebuild without touching DNS, which keeps registered webhook URLs valid.

---

4. Installing Docker Engine and the Compose Plugin

Use Docker's own apt repository, exactly as in "Deploying on AWS EC2":

```
sudo apt-get update
sudo apt-get install -y ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc
echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
        https://download.docker.com/linux/ubuntu $(. /etc/os-release && echo "$VERSION_CODENAME")
        stable" \
  | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-
        plugin
```

If you are logged in as a non-root user, add it to the `docker` group with `sudo usermod -aG docker $USER` and run `newgrp docker`. Working as root is common on Droplets; if you do, create an unprivileged user first.

---

5. The Project Folder

```
mkdir -p /opt/n8n && cd /opt/n8n
openssl rand -hex 32
openssl rand -hex 24
```

Copy `.env` and `compose.yml` from “Deploying on AWS EC2” unchanged, set `N8N_HOST=n8n.example.com`, and paste the two generated secrets into `N8N_ENCRYPTION_KEY` and `N8N_RUNNERS_AUTH_TOKEN`. As written, that stack runs Postgres, n8n, and the `runners` sidecar on the Droplet. The runner sidecar executes every Code node and must always carry the same image tag as n8n.

That is a complete deployment. The rest of this chapter replaces the local database and the local TLS termination with DigitalOcean’s managed equivalents.

---

6. Managed PostgreSQL

Create a database under **Databases** → **PostgreSQL** in the same region and VPC as the Droplet. Under **Settings** → **Trusted Sources**, add the Droplet; the database refuses every other client. Download the CA certificate the connection panel offers:

```
# Save the downloaded ca-certificate.crt next to compose.yml
ls /opt/n8n/ca-certificate.crt
```

Managed PostgreSQL requires TLS and listens on **25060**, not 5432, with a default database named `defaultdb`. Change the compose file from “Deploying on AWS EC2” as follows:

- Delete the `postgres` service, the `postgres_data` volume, and the `depends_on` block under `n8n`.
- Add a read-only mount to `n8n`: `- ./ca-certificate.crt:/etc/ssl/do-ca.crt:ro`.
- Replace the database environment values:

```
      DB_POSTGRESDB_HOST: ${DO_DB_HOST}
      DB_POSTGRESDB_PORT: 25060
      DB_POSTGRESDB_DATABASE: ${DO_DB_NAME}
      DB_POSTGRESDB_USER: ${DO_DB_USER}
      DB_POSTGRESDB_PASSWORD: ${DO_DB_PASSWORD}
      DB_POSTGRESDB_SSL_ENABLED: "true"
      DB_POSTGRESDB_SSL_CA_FILE: /etc/ssl/do-ca.crt
```

Put the four `DO_DB_*` values in `.env` using the **private** host from the connection panel, and delete the `POSTGRES_*` lines. Private networking keeps database traffic off the public internet and out of your bandwidth bill.

---

7. Starting the Stack

```
cd /opt/n8n
docker compose up -d
docker compose ps
docker compose logs -f n8n
```

Open `https://n8n.example.com` and create the owner account on first visit. User management, roles, and 2FA live in **Settings**.

Auto-start is already handled: `restart: unless-stopped` on every service plus Docker's systemd unit brings the stack back after a reboot, an OOM kill, or a crash. Verify with `systemctl is-enabled docker`.

---

8. DigitalOcean Load Balancer

A load balancer in front of the single Droplet below is for TLS termination and health checks, not for scaling out. It gives you a managed Let's Encrypt certificate and a health check against `/healthz`. More than one n8n Droplet behind it needs queue mode — see "Docker Compose in Queue Mode: Main, Workers, Webhooks, and Runners" — and a multi-main setup is an Enterprise feature.

1. Create the load balancer in the same VPC, forwarding HTTPS:443 to HTTP:5678 on the Droplet, with a certificate issued for your domain.
2. Set the health check to HTTP on port 5678, path `/healthz`.
3. Point the DNS record at the load balancer instead of the reserved IP.

Then change two things:

- In `compose.yml`, publish on all interfaces: `- "5678:5678"`, and restrict port 5678 in the Cloud Firewall to the load balancer.
- Keep `N8N_PROXY_HOPS: 1`, which is correct for exactly one proxy in front of n8n, and keep `N8N_HOST` equal to the public hostname the load balancer serves. `WEBHOOK_URL` and `N8N_EDITOR_BASE_URL` derive from it.

The editor uses WebSockets. DigitalOcean's load balancer passes `Upgrade` and `Connection` headers on HTTP forwarding rules; if you build your own proxy instead, see "Reverse Proxy and HTTPS: Caddy, Nginx, and Traefik".

---

9. Block Storage for Docker Data

Workflow binary data and Postgres files grow faster than the Droplet's root disk. Attach a volume and move Docker's data root onto it:

```
sudo mkdir -p /mnt/n8n_data
sudo systemctl stop docker
sudo rsync -aP /var/lib/docker/ /mnt/n8n_data/docker/
echo '{"data-root": "/mnt/n8n_data/docker"}' | sudo tee /etc/docker/daemon.json
sudo systemctl start docker
docker compose -f /opt/n8n/compose.yml up -d
```

Named volumes follow the data root, so `n8n_data` and `n8n_files` land on the volume without any compose change.

---

10. Snapshots and Backups

Droplet snapshots capture the whole machine, including Docker volumes, and are the fastest rollback before an upgrade. They are not a database backup strategy: managed PostgreSQL keeps its own daily backups with point-in-time restore. Back up `N8N_ENCRYPTION_KEY` separately from both — a database restored without it yields credentials n8n cannot decrypt. See “Backup, Restore, and Disaster Recovery” for the full procedure.

---

11. Updating n8n

Edit the two pinned tags in `compose.yml`, keeping `n8nio/n8n` and `n8nio/runners` on the same version, then:

```
docker compose pull
docker compose up -d
```

Take a snapshot first, and read the release notes for the major version you are crossing.

---

12. Troubleshooting Common Issues

Issue	Cause	Solution
SSH refused	Cloud Firewall or UFW blocking 22	Check both layers; the Cloud Firewall is evaluated first
`connection refused` to the database	Droplet not in Trusted Sources, or port 5432 assumed	Add the Droplet as a trusted source and use port 25060
`self signed certificate in certificate chain`	CA file not mounted	Mount `ca-certificate.crt` and set `DB_POSTGRESDB_SSL_CA_FILE`
Load balancer health check fails	Port bound to `127.0.0.1`	Publish `"5678:5678"` and re-run `docker compose up -d`
Editor loads, webhook URLs show the Droplet IP	`N8N_HOST` not set to the public hostname	Fix `.env` and recreate the container
Code nodes hang or error	Runner sidecar missing or token mismatch	Confirm the `runners` service is up and shares the auth token
Root disk full	Docker data root still on the 25 GB volume	Move the data root to block storage, prune unused images

---

Further reading

- https://docs.n8n.io/deploy/host-n8n/install-options/install-using-docker-compose
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/deployment
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners
- https://docs.digitalocean.com/products/databases/postgresql/how-to/connect/

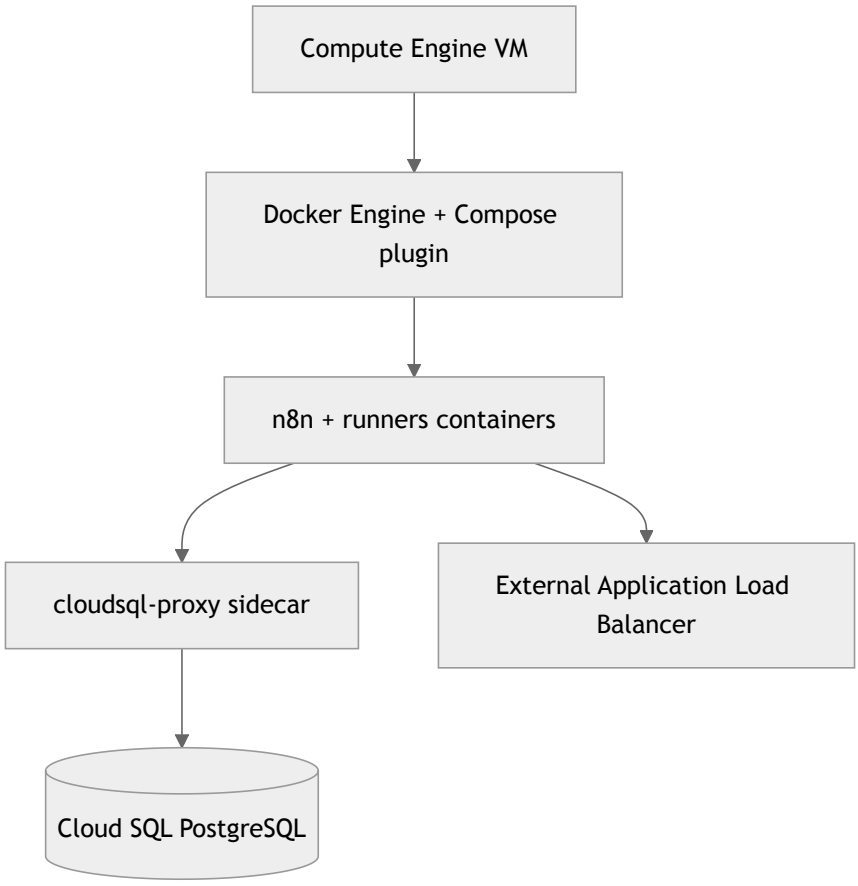
Chapter 9: Deploying on Google Cloud Compute Engine

Overview

Compute Engine gives you a VM plus Google's IAM, logging, and load balancing. The interesting difference from other clouds is Cloud SQL: instead of opening a database port, you run the Cloud SQL Auth Proxy as another container in the same Compose project and connect to it over the Docker network.

In this chapter you will:

1. Provision a VM with `gcloud`
2. Configure VPC firewall rules with network tags
3. Reserve a static external IP
4. Connect with OS Login
5. Install Docker Engine and the Compose plugin
6. Adapt the reference stack for GCP
7. Attach Cloud SQL through the Auth Proxy sidecar
8. Add an external Application Load Balancer, Cloud Logging, and a scoped service account



Diagram

Prerequisites

- A Google Cloud project with the Compute Engine and Cloud SQL Admin APIs enabled

- `gcloud` installed and authenticated
- Permission to create instances, firewall rules, addresses, and service accounts
- The compose file from “Deploying on AWS EC2”, which this chapter modifies

---

1. Provisioning the VM

```
gcloud compute instances create n8n-vm \
  --zone=us-central1-a \
  --machine-type=e2-medium \
  --image-family=ubuntu-2404-lts-amd64 \
  --image-project=ubuntu-os-cloud \
  --boot-disk-size=30GB \
  --boot-disk-type=pd-balanced \
  --tags=n8n \
  --metadata=enable-oslogin=TRUE \
  --service-account=n8n-vm@PROJECT_ID.iam.gserviceaccount.com \
  --scopes=cloud-platform
```

`e2-medium` (2 vCPU, 4 GiB) is the smallest size that comfortably runs n8n, a task runner, and the Cloud SQL proxy. The `n8n` network tag is what the firewall rules below attach to.

---

2. VPC Firewall Rules

Do not create a rule for 5678 open to the world. Allow SSH through Identity-Aware Proxy and let the load balancer reach the application port:

```
gcloud compute firewall-rules create allow-iap-ssh \
  --network=default --direction=INGRESS --action=ALLOW \
  --rules=tcp:22 --source-ranges=35.235.240.0/20 --target-tags=n8n

gcloud compute firewall-rules create allow-lb-to-n8n \
  --network=default --direction=INGRESS --action=ALLOW \
  --rules=tcp:5678 --source-ranges=130.211.0.0/22,35.191.0.0/16 --target-tags=n8n
```

Those two source ranges are Google's health check and load balancer probers. Nothing else can reach port 5678.

---

3. Static External IP

```
gcloud compute addresses create n8n-ip --region=us-central1
gcloud compute instances delete-access-config n8n-vm \
  --access-config-name="external-nat" --zone=us-central1-a
gcloud compute instances add-access-config n8n-vm \
  --access-config-name="external-nat" --zone=us-central1-a \
  --address=$(gcloud compute addresses describe n8n-ip --region=us-central1 --format='get(address)')
```

Point `n8n.example.com` at that address, or at the load balancer's address if you add one in section 8.

---

4. Connecting with OS Login

OS Login maps SSH access to IAM instead of to keys scattered in project metadata. Grant yourself `roles/compute.osLogin` and connect through IAP:

```
gcloud compute ssh n8n-vm --zone=us-central1-a --tunnel-through-iap
```

---

5. Installing Docker Engine and the Compose Plugin

Ubuntu's packaged Docker has no Compose plugin. Use Docker's repository, as in "Deploying on AWS EC2":

```
sudo apt-get update
sudo apt-get install -y ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc
echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
        https://download.docker.com/linux/ubuntu $(. /etc/os-release && echo "$VERSION_CODENAME")
        stable" \
  | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-
        plugin
sudo usermod -aG docker $USER
newgrp docker
```

---

6. The Project Folder

```
sudo mkdir -p /opt/n8n && sudo chown $USER /opt/n8n && cd /opt/n8n
openssl rand -hex 32
openssl rand -hex 24
```

Start from the `.env` and `compose.yml` in “Deploying on AWS EC2”. Set `N8N_HOST` to your public hostname, paste in the two generated secrets, and you have a working stack: Postgres, n8n, and the `runners` sidecar that executes every Code node and must always share n8n’s version tag.

---

7. Cloud SQL Through the Auth Proxy

Create a PostgreSQL instance in Cloud SQL, a database named `n8n`, and a user for it. Grant the VM's service account `roles/cloudsql.client`. Do not add an authorised network: the proxy authenticates with IAM and encrypts the connection itself, so the database keeps no public exposure.

Change the compose file from "Deploying on AWS EC2":

- Delete the `postgres` service and the `postgres_data` volume.
- Add the proxy as a service:

```
cloudsql-proxy:
  image: gcr.io/cloud-sql-connectors/cloud-sql-proxy:2.14.3
  restart: unless-stopped
  command:
    - "--address=0.0.0.0"
    - "--port=5432"
    - "${CLOUDSQL_INSTANCE}"
```

- Point n8n at it and make it wait for the proxy:

```
    DB_POSTGRESDB_HOST: cloudsql-proxy
    DB_POSTGRESDB_PORT: 5432
    DB_POSTGRESDB_DATABASE: ${DB_NAME}
    DB_POSTGRESDB_USER: ${DB_USER}
    DB_POSTGRESDB_PASSWORD: ${DB_PASSWORD}
```

```
  depends_on:
    - cloudsql-proxy
```

In `.env`, replace the `POSTGRES_*` lines with `CLOUDSQL_INSTANCE=project:region:instance`, `DB_NAME`, `DB_USER`, and `DB_PASSWORD`. No `DB_POSTGRESDB_SSL_*` variables are needed here, and no CA file: the hop from n8n to the proxy stays inside the Docker network, and the proxy handles TLS and certificate rotation to Cloud SQL for you.

The tradeoff is one more moving part. If the proxy container is down, n8n cannot reach its database and will restart in a loop until the proxy is healthy, so read `docker compose logs cloudsql-proxy` first when n8n will not start. The alternative — a private IP on the Cloud SQL instance reached directly from the VM's subnet — removes the sidecar but requires VPC peering with the service producer network and gives up IAM-based database authentication.

---

8. Starting the Stack

```
cd /opt/n8n
docker compose up -d
docker compose ps
docker compose logs -f n8n
```

Auto-start requires nothing further. `restart: unless-stopped` on each service plus Docker's systemd unit restores the stack after a reboot or a live-migration restart. Check with `systemctl is-enabled docker`.

---

9. External Application Load Balancer

A global external Application Load Balancer in front of the single VM below is for TLS termination and health checks, not for scaling out — it terminates TLS with a Google-managed certificate and health-checks the VM. More than one n8n VM behind it needs queue mode, covered in “Docker Compose in Queue Mode: Main, Workers, Webhooks, and Runners”; a multi-main setup is an Enterprise feature.

1. Add the VM to an unmanaged instance group with named port `http:5678`.
2. Create a health check on port 5678, path `/healthz`.
3. Create a backend service using that group and health check, with **CDN off**.
4. Create a managed certificate for `n8n.example.com` and an HTTPS target proxy plus forwarding rule; point DNS at the load balancer’s IP.

Then change two things in the project:

- Publish the container port on all interfaces: `- "5678:5678"`.
- Set `N8N_HOST` in `.env` to the hostname served by the load balancer. `WEBHOOK_URL` and `N8N_EDITOR_BASE_URL` follow it.

Leave `N8N_PROXY_HOPS: 1`, which matches exactly one proxy layer and lets n8n read the real client IP from `X-Forwarded-For`. Raise the backend service timeout above the default 30 seconds if long executions stream responses, and remember the editor needs WebSocket support, which the Application Load Balancer provides on HTTP backends.

---

10. Cloud Logging

Send container output to Cloud Logging with Docker's native driver. Add to the `n8n` and `runners` services:

```
    logging:
      driver: gcplogs
      options:
        gcp-log-cmd: "true"
```

The VM's service account needs `roles/logging.logWriter`. Logs then appear under the `gcplogs-docker-driver` resource, searchable by container name, and you can build log-based metrics and alerts on error patterns. For Prometheus instead, set `N8N_METRICS: "true"` and scrape `/metrics`.

---

11. Service Accounts and Secrets

Give the VM a dedicated service account with only `roles/cloudsql.client` and `roles/logging.logWriter`, plus `roles/secretmanager.secretAccessor` if you keep the database password and encryption key in Secret Manager. Fetch them into `.env` at deploy time:

```
gcloud secrets versions access latest --secret=n8n-encryption-key
```

Keep `.env` at mode `600`. The default Compute Engine service account with broad scopes is not appropriate for a machine exposed to the internet.

---

12. Updating n8n

Change the two pinned tags to the same new version and pull:

```
docker compose pull
docker compose up -d
```

Take a Cloud SQL backup before a major version jump, and update the proxy tag on its own schedule.

---

13. Troubleshooting Common Issues

Issue	Cause	Solution
`Permission denied (publickey)`	OS Login role missing	Grant `roles/compute.osLogin` and connect with `gcloud compute ssh`
Load balancer backend unhealthy	Health check blocked or port bound to localhost	Allow `130.211.0.0/22` and `35.191.0.0/16`, publish `"5678:5678"`
Proxy exits with a permissions error	Service account lacks `cloudsql.client`	Grant the role, then `docker compose up -d cloudsql-proxy`
n8n cannot resolve `cloudsql-proxy`	Proxy service removed or renamed	Keep both services in the same Compose project and network
Webhook URLs show the VM IP	`N8N_HOST` not updated after adding the load balancer	Set the public hostname in `.env` and recreate the container
Nothing in Cloud Logging	Missing `logging.logWriter` or driver not applied	Grant the role and recreate the containers
Code node errors after an upgrade	n8n and runners tags diverged	Pin both images to the same version

---

Further reading

- https://docs.n8n.io/deploy/host-n8n/install-options/install-using-docker-compose
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/deployment
- https://cloud.google.com/sql/docs/postgres/connect-auth-proxy

---

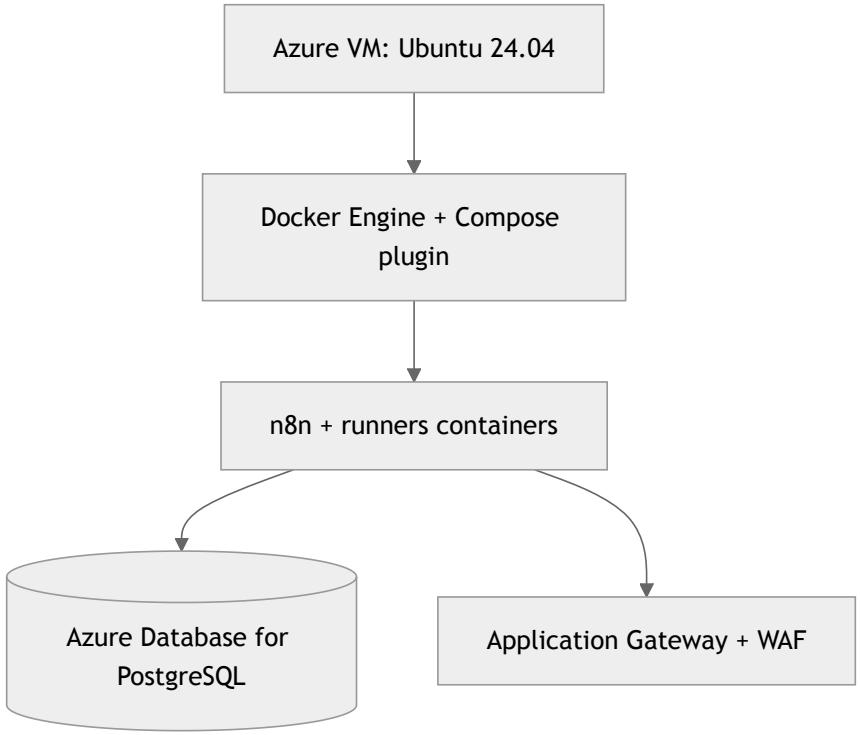
Chapter 10: Deploying on Azure Virtual Machines

Overview

An Azure VM running Ubuntu hosts the same container stack as every other chapter in this part. What Azure adds around it is a network security group, Azure Database for PostgreSQL flexible server on a private subnet, Application Gateway for TLS and WAF, and Azure Monitor for logs.

In this chapter you will:

1. Provision a VM with the `az` CLI
2. Configure the NSG and UFW
3. Assign a static public IP and DNS
4. Install Docker Engine and the Compose plugin
5. Adapt the reference stack for Azure
6. Point n8n at Azure Database for PostgreSQL
7. Add Application Gateway, managed disks, and Azure Monitor



Diagram

Prerequisites

- An Azure subscription with rights to create VMs, networking, and databases
- The `az` CLI installed and logged in
- An SSH key pair
- The compose file from “Deploying on AWS EC2”, which this chapter modifies

1. Provisioning the VM

```
az group create --name rg-n8n-prod --location eastus

az vm create \
  --resource-group rg-n8n-prod \
  --name n8n-vm \
  --image Ubuntu2404 \
  --size Standard_D2as_v5 \
  --admin-username azureuser \
  --ssh-key-values ~/.ssh/id_rsa.pub \
  --public-ip-sku Standard \
  --public-ip-address-allocation static \
  --os-disk-size-gb 30 \
  --nsg-rule NONE
```

`Standard_D2as_v5` (2 vCPU, 8 GiB) leaves headroom for the containers and the OS. `Standard_B2s` works for light workloads; the 1 GiB B-series sizes do not. `--nsg-rule NONE` creates no inbound rules, which is the right default — you add them next.

---

2. Network Security Group and UFW

```
az network nsg rule create --resource-group rg-n8n-prod \
  --nsg-name n8n-vmNSG --name Allow-SSH --priority 1000 \
  --destination-port-ranges 22 --source-address-prefixes 203.0.113.5/32 \
  --protocol Tcp --access Allow

az network nsg rule create --resource-group rg-n8n-prod \
  --nsg-name n8n-vmNSG --name Allow-AppGw --priority 1010 \
  --destination-port-ranges 5678 --source-address-prefixes 10.0.1.0/24 \
  --protocol Tcp --access Allow
```

The second rule allows only the Application Gateway subnet. If you terminate TLS on the VM instead, replace it with rules for 80 and 443 from the internet. Add UFW as a second layer:

```
sudo apt-get update && sudo apt-get install -y ufw
sudo ufw allow ssh
sudo ufw --force enable
```

---

3. Static Public IP and DNS

The `az vm create` above already allocated a Standard SKU static IP, so the address survives deallocation. Create an A record for `n8n.example.com` pointing at it, or at the Application Gateway's frontend IP if you add one.

---

4. Connecting Over SSH

```
ssh azureuser@203.0.113.20
lsb_release -ds
```

The admin user is whatever you passed to `--admin-username`, not `ubuntu`. Confirm with `az vm show --show-details` if you forget.

---

5. Installing Docker Engine and the Compose Plugin

Same repository setup as in “Deploying on AWS EC2”:

```
sudo apt-get update
sudo apt-get install -y ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc
echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
        https://download.docker.com/linux/ubuntu $(. /etc/os-release && echo "$VERSION_CODENAME")
        stable" \
  | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-
        plugin
sudo usermod -aG docker $USER
newgrp docker
```

---

6. The Project Folder

```
sudo mkdir -p /opt/n8n && sudo chown $USER /opt/n8n && cd /opt/n8n
openssl rand -hex 32
openssl rand -hex 24
```

Copy `.env` and `compose.yml` from “Deploying on AWS EC2”, set `N8N_HOST` to the public hostname, and paste the generated values into `N8N_ENCRYPTION_KEY` and `N8N_RUNNERS_AUTH_TOKEN`. The `runners` service in that file is the external task runner that executes every Code node; keep its tag identical to the n8n tag at all times.

---

7. Azure Database for PostgreSQL

Create a flexible server with **private access (VNet integration)** into a delegated subnet of the VM's virtual network:

```
az postgres flexible-server create \
  --resource-group rg-n8n-prod \
  --name n8n-pg \
  --location eastus \
  --tier Burstable --sku-name Standard_B2s \
  --version 17 --database-name n8n \
  --vnet n8n-vmVNET --subnet pg-subnet
```

Flexible server requires TLS. Its certificate chains to the DigiCert Global Root G2, so download that root and mount it rather than disabling verification:

```
curl -o /opt/n8n/azure-pg-ca.pem https://cacerts.digicert.com/DigiCertGlobalRootG2.crt.pem
```

Change the compose file from "Deploying on AWS EC2":

- Delete the `postgres` service, the `postgres_data` volume, and the `depends_on` block under `n8n`.
- Add a read-only mount to `n8n`: `- ./azure-pg-ca.pem:/etc/ssl/azure-pg-ca.pem:ro`.
- Replace the database environment values:

```
    DB_POSTGRESDB_HOST: ${AZ_DB_HOST}
    DB_POSTGRESDB_PORT: 5432
    DB_POSTGRESDB_DATABASE: n8n
    DB_POSTGRESDB_USER: ${AZ_DB_USER}
    DB_POSTGRESDB_PASSWORD: ${AZ_DB_PASSWORD}
    DB_POSTGRESDB_SSL_ENABLED: "true"
    DB_POSTGRESDB_SSL_CA_FILE: /etc/ssl/azure-pg-ca.pem
```

`AZ_DB_HOST` is `n8n-pg.postgres.database.azure.com`, and the user is the plain administrator name — flexible server does not use the `user@server` form that single server required. Replace the `POSTGRES_*` lines in `.env` with these three.

---

8. Starting the Stack

```
cd /opt/n8n
docker compose up -d
docker compose ps
docker compose logs -f n8n
```

Create the owner account on your first visit to the editor; n8n manages users, roles, and 2FA itself.

Auto-start is covered by `restart: unless-stopped` on every service together with Docker's systemd unit, so the stack returns after a reboot or an Azure host maintenance restart. Verify once with `systemctl is-enabled docker`.

---

9. Application Gateway

Application Gateway v2 terminates TLS with a certificate from Key Vault and can run the WAF ruleset in front of n8n.

1. Deploy the gateway into its own subnet in the VM's VNet.
2. Backend pool: the VM's **private** IP, backend setting HTTP on port 5678.
3. Custom health probe: path `/healthz` on port 5678.
4. HTTPS listener on 443 with the Key Vault certificate, plus an HTTP listener that redirects to it.

Two project changes:

- Publish the port on all interfaces so the gateway can reach it: `- "5678:5678"`, with the NSG rule from section 2 limiting the source to the gateway subnet.
- Set `N8N_HOST` in `.env` to the hostname on the gateway's certificate, so `WEBHOOK_URL` and `N8N_EDITOR_BASE_URL` resolve to the public name.

Keep `N8N_PROXY_HOPS: 1` for the single gateway hop. Enable WebSocket support on the backend setting — it is on by default in v2 — and raise the request timeout above 30 seconds if long executions time out at the gateway rather than in n8n.

---

10. Managed Disk for Docker Data

Attach a data disk and move Docker's data root onto it so images and volumes do not fill the OS disk:

```
sudo mkfs.ext4 /dev/sdc
sudo mkdir -p /mnt/n8n_data
echo '/dev/sdc /mnt/n8n_data ext4 defaults,nofail 0 2' | sudo tee -a /etc/fstab
sudo mount -a
sudo systemctl stop docker
sudo rsync -aP /var/lib/docker/ /mnt/n8n_data/docker/
echo '{"data-root": "/mnt/n8n_data/docker"}' | sudo tee /etc/docker/daemon.json
sudo systemctl start docker
```

Named volumes move with the data root, so `compose.yml` needs no change. Prefer the disk's `/dev/disk/by-uuid/...` path in `/etc/fstab`; device letters can shuffle across reboots.

---

11. Azure Monitor and Log Analytics

Install the Azure Monitor Agent on the VM and create a data collection rule with a **custom text log** source pointing at Docker's JSON logs:

```
/var/lib/docker/containers/*/*-json.log
```

Cap their growth in `/etc/docker/daemon.json` alongside the data root:

```
{
  "data-root": "/mnt/n8n_data/docker",
  "log-driver": "json-file",
  "log-opts": { "max-size": "10m", "max-file": "3" }
}
```

Query the resulting table in Log Analytics and alert on error patterns. For metrics, set `N8N_METRICS: "true"` and scrape `/metrics` with Azure Monitor managed Prometheus.

---

12. Updating n8n

Edit the two pinned tags to the same new version, then pull and recreate:

```
docker compose pull
docker compose up -d
```

Trigger an on-demand backup of the flexible server first. Restoring the database without the matching `N8N_ENCRYPTION_KEY` leaves every credential unreadable, so keep the key in Key Vault.

---

13. Troubleshooting Common Issues

Issue	Cause	Solution
`Permission denied` on SSH	Wrong admin username	Check `az vm show --show-details` for `adminUsername`
Application Gateway returns 502	Probe failing or port bound to localhost	Publish `"5678:5678"`, probe `/healthz` on 5678
Database connection times out	VM and flexible server in different VNets	Use private access in the same VNet, or peer the networks
TLS error connecting to PostgreSQL	CA not mounted, or old `user@server` login form	Mount the DigiCert root, use the plain administrator name
Webhooks call the VM's IP	`N8N_HOST` still the instance address	Set the gateway hostname in `.env` and recreate the container
OS disk fills after weeks	Container logs never rotated	Set `max-size` and `max-file`, move the data root to a data disk
Code nodes fail after an upgrade	Runner image tag no longer matches n8n	Pin both to the same version and restart

---

Further reading

- https://docs.n8n.io/deploy/host-n8n/install-options/install-using-docker-compose
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/deployment
- https://learn.microsoft.com/azure/postgresql/flexible-server/concepts-networking-ssl-tls

---

Chapter 11: Exposing a Private Instance: Cloudflare Tunnel, ngrok, and SSH

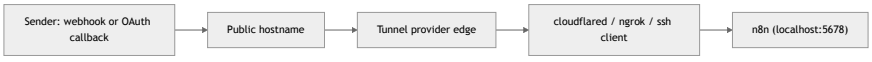
Overview

Webhooks and OAuth callbacks are useless if the sender cannot reach your n8n instance. If n8n is running on a laptop, behind a home router, or on a VPC with no public IP, you need something to carry traffic in from the internet. A tunnel is the fastest way to get that without opening a port on your firewall: a small client process on your machine makes an outbound connection to a provider, and the provider hands you a public hostname that forwards back down that connection.

This chapter covers four ways to do it, in the order to reach for them:

1. **Cloudflare Tunnel** — free, no exposed ports, a stable named hostname, and Cloudflare Access as an authentication layer in front of n8n. The default choice for anything longer-lived than a five-minute test.
2. **ngrok** — fastest to start, good request inspector, paid plans for reserved domains and its own OAuth gate.
3. **localtunnel** — open source, npm-installed, no account, no guarantees.
4. **SSH reverse tunnel** — no third-party service at all, if you already have a reachable box to tunnel through.

n8n's own `n8n --tunnel` flag, which proxied through an n8n.io-run relay, was removed in n8n 2.0. It no longer exists in any 2.x release, so this chapter does not cover it. Use one of the four options above instead — Cloudflare Tunnel costs nothing and takes about the same time to set up.



Diagram

Prerequisites

- A running n8n instance, reachable at `http://localhost:5678` from the machine that will run the tunnel client.
- An owner account already created in n8n — the tunnel does not create one for you, and it is your only login.
- For Cloudflare Tunnel: a domain added to a free Cloudflare account, with its nameservers pointed at Cloudflare.
- For ngrok: an account and an authtoken.
- For the SSH option: a second host you control, reachable over SSH, with a public IP or its own DNS name.

1. The Five Variables Every Tunnel Needs

Whichever tunnel you pick, n8n needs to know its own public address, or webhook URLs and OAuth redirect URLs get generated with `localhost:5678` baked in and nothing that reaches you from the outside will work. Before you run any tunnel client, set these on the n8n container or process, using the tunnel's hostname:

```
N8N_HOST=n8n-dev.example.com
N8N_PROTOCOL=https
WEBHOOK_URL=https://n8n-dev.example.com/
N8N_EDITOR_BASE_URL=https://n8n-dev.example.com/
N8N_PROXY_HOPS=1
```

Every recipe in this chapter assumes these five are set to that session's tunnel hostname before you start the tunnel client. The rest of this chapter says "set the tunnel variables" and refers back here rather than repeating the block. If you tear down a tunnel and get a new random hostname next time (ngrok's free tier, localtunnel), you have to update these and restart n8n — one more reason a named, stable hostname (Cloudflare Tunnel, ngrok's paid reserved domains) is worth having for anything beyond a single afternoon of testing.

Set `N8N_PROXY_HOPS=1` in every case above — the tunnel client sits in front of n8n and terminates the public connection, so from n8n's perspective it is one hop, the same as a reverse proxy would be. If you additionally run a reverse proxy in front of n8n (see "Reverse Proxy and HTTPS: Caddy, Nginx, and Traefik"), increase `N8N_PROXY_HOPS` to the total number of hops between the internet and n8n.

---

2. Cloudflare Tunnel

Cloudflare Tunnel runs `cloudflared`, a small daemon that opens four outbound HTTPS connections to Cloudflare's network. Nothing listens on your firewall; there is no port to forward. A named tunnel gives you a hostname that survives restarts, and you can put Cloudflare Access in front of it for a second authentication layer without touching n8n.

2.1 Install cloudflared

```
# macOS
brew install cloudflared

# Debian/Ubuntu
curl -fsSL https://pkg.cloudflare.com/cloudflare-main.gpg | sudo gpg --dearmor -o
        /usr/share/keyrings/cloudflare-main.gpg
echo "deb [signed-by=/usr/share/keyrings/cloudflare-main.gpg] https://pkg.cloudflare.com/cloudflared
        $(lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/cloudflared.list
sudo apt update && sudo apt install cloudflared
```

2.2 Authenticate and create a named tunnel

```
cloudflared tunnel login
cloudflared tunnel create n8n
```

`tunnel create` writes a credentials file, usually to `~/.cloudflared/<TUNNEL-ID>.json`, and prints the tunnel's UUID. Keep both; you reference them in the config file next.

2.3 Route DNS and write the config

```
cloudflared tunnel route dns n8n n8n-dev.example.com
```

This adds a CNAME for `n8n-dev.example.com` pointing at `<TUNNEL-ID>.cfargotunnel.com` in your Cloudflare DNS zone — no manual DNS editing required.

```
# ~/.cloudflared/config.yml
tunnel: <TUNNEL-ID>
credentials-file: /home/you/.cloudflared/<TUNNEL-ID>.json

ingress:
  - hostname: n8n-dev.example.com
    service: http://localhost:5678
  - service: http_status:404
```

2.4 Set the tunnel variables and run it

Set the five variables from section 1 to `n8n-dev.example.com`, restart n8n, then run:

```
cloudflared tunnel run n8n
```

2.5 Run cloudflared as a service

For anything you want up after a reboot, install it as a system service instead of leaving a terminal open:

```
sudo cloudflared service install
sudo systemctl enable --now cloudflared
sudo systemctl status cloudflared
```

2.6 Run cloudflared as a compose sidecar

If n8n already runs under Compose, add `cloudflared` as another service in the same file rather than running it on the host. The `n8n` service below shows only the lines that change from the reference

stack in “Ubuntu VPS with Docker Compose (the Reference Stack)” — combine it with that chapter’s full compose file for `N8N_ENCRYPTION_KEY`, the `N8N_RUNNERS_*` block, Postgres, and the runners sidecar:

```
services:
  n8n:
    image: n8nio/n8n:2.38.1
    restart: unless-stopped
    environment:
      N8N_HOST: n8n-dev.example.com
      N8N_PROTOCOL: https
      WEBHOOK_URL: https://n8n-dev.example.com/
      N8N_EDITOR_BASE_URL: https://n8n-dev.example.com/
      N8N_PROXY_HOPS: 1
    volumes:
      - n8n_data:/home/node/.n8n

  cloudflared:
    image: cloudflare/cloudflared:2025.9.1
    restart: unless-stopped
    command: tunnel run n8n
    environment:
      TUNNEL_TOKEN: ${CLOUDFLARE_TUNNEL_TOKEN}
    depends_on:
      - n8n

volumes:
  n8n_data:
```

Using `TUNNEL_TOKEN` (from the Cloudflare Zero Trust dashboard’s tunnel page, “Docker” install option) skips the config file entirely — the token carries the tunnel ID, credentials, and ingress rule to a hostname you set up in the dashboard. Inside the Compose network, `cloudflared` reaches n8n at `http://n8n:5678`, the service name, not `localhost`.

2.7 Add Cloudflare Access as the auth layer

n8n’s own login — the owner account, plus any members you invite — is the only authentication n8n does. There is no basic auth to layer on top of it anymore; that mechanism was removed in n8n 1.0. What you can add in front of it is Cloudflare Access: in the Zero Trust dashboard, create an Access application for `n8n-dev.example.com`, and set a policy (email address, email domain, or an identity provider like Google Workspace). Cloudflare then puts its own login page in front of every request before it reaches `cloudflared`, so a stranger who finds the hostname still cannot reach n8n’s login screen at all. This costs nothing on Cloudflare’s free tier and needs no changes to the n8n container.

---

3. ngrok

ngrok is the fastest path to a public URL and has the best request inspector of the four options, at the cost of a random hostname on the free tier.

3.1 Install and authenticate

```
brew install ngrok        # macOS
# or download from https://ngrok.com/download for Linux/Windows

ngrok config add-authtoken <YOUR_AUTHTOKEN>
```

3.2 Start the tunnel

```
ngrok http 5678
```

ngrok prints a forwarding line such as `https://a1b2c3d4.ngrok-free.app -> http://localhost:5678`. Set the tunnel variables from section 1 to that hostname and restart n8n before sending it any webhook traffic — every workflow's webhook URL is generated from `WEBHOOK_URL` at save time, so changing the hostname after you have built workflows means re-saving nodes that show the old URL.

3.3 A reserved domain

Free-tier ngrok hostnames change every time you restart the tunnel, which breaks every webhook URL you registered with an external service. A paid plan gets you a domain that doesn't move:

```
ngrok http 5678 --url=https://n8n-dev.ngrok.app
```

3.4 ngrok's OAuth as the auth layer

Like Cloudflare Access, ngrok can require a login before traffic reaches n8n at all, on paid plans:

```
ngrok http 5678 --url=https://n8n-dev.ngrok.app \
  --oauth=google --oauth-allow-domain=example.com
```

Use this, or Cloudflare Access, as your second authentication layer — n8n's owner login is the first and only layer n8n itself provides.

---

4. localtunnel

localtunnel is a small open-source npm package with no account and no SLA. It is fine for a five-minute test and nothing you should leave running.

```
npm install -g localtunnel
lt --port 5678 --subdomain n8n-dev
```

It prints `your url is: https://n8n-dev.loca.lt`. Set the tunnel variables to that hostname and restart n8n. The `--subdomain` flag is a request, not a reservation — if someone else is already using it, localtunnel silently gives you a different random one, so check the printed URL before assuming it matches what you configured. localtunnel has no authentication layer of its own; if you need one, use Cloudflare Tunnel or ngrok instead.

---

5. SSH Reverse Tunnel

If you already have a VPS or any box with a public IP, you don't need a third-party tunneling service at all — SSH can do it.

5.1 One-off tunnel

```
ssh -R 0.0.0.0:8080:localhost:5678 you@your-vps.example.com
```

This asks the remote host to listen on port 8080 and forward anything it receives back through the SSH connection to `localhost:5678` on your machine. It requires `GatewayPorts yes` in the remote's `/etc/ssh/sshd_config` — without it, the remote binds the forwarded port to its own loopback only, and nothing outside that box can reach it.

Put a reverse proxy (see "Reverse Proxy and HTTPS: Caddy, Nginx, and Traefik") in front of port 8080 on the VPS to get TLS and a real hostname, then set the tunnel variables to that hostname.

5.2 Keep it alive with autossh

A plain `ssh -R` session drops the moment your laptop sleeps or your network hiccups, silently taking the tunnel with it. `autossh` monitors the connection and restarts it:

```
sudo apt install autossh

autossh -M 0 -N \
  -o "ServerAliveInterval 30" -o "ServerAliveCountMax 3" \
  -R 0.0.0.0:8080:localhost:5678 you@your-vps.example.com
```

`-M 0` disables autossh's own legacy monitoring port and relies on the SSH `ServerAlive` options instead, which is the more reliable combination on modern OpenSSH.

---

6. Security Considerations

- **The owner login is the authentication.** There is no basic auth to fall back on; it does not exist in any currently supported n8n release. Anyone who reaches the tunnel hostname sees n8n's own login screen, and that screen is only as strong as the owner account's password and whether you've turned on 2FA for it.
- **Add a second layer for anything that isn't a five-minute test.** Cloudflare Access or ngrok's OAuth gate sits in front of the tunnel and blocks unauthenticated requests before they ever reach n8n's login page. This is the practical replacement for "keep basic auth on" — it isn't optional if the tunnel will run for more than a session.
- **Rotate credentials.** Regenerate ngrok authtokens and Cloudflare tunnel credentials if you suspect they leaked, and delete unused named tunnels from the Cloudflare dashboard.
- **Treat every tunnel as internet-facing.** A webhook path is a URL anyone can guess or find in a leaked log; don't rely on obscurity. Validate webhook signatures inside the workflow when the sending service supports them.
- **Tunnels are not a substitute for a real deployment.** For anything serving production traffic, use a VPS or managed platform with its own TLS certificate and DNS record — see "Ubuntu VPS with Docker Compose (the Reference Stack)" and "Platform-as-a-Service: Render, Railway, and Fly.io".

---

7. Troubleshooting Common Issues

Issue	Cause	Solution
Webhook URLs still show `localhost:5678`	`WEBHOOK_URL` was not set, or n8n wasn't restarted after setting it	Set all five tunnel variables, then restart the n8n container; re-save any workflow whose webhook node cached the old URL
OAuth callback fails with a redirect mismatch	`N8N_EDITOR_BASE_URL` doesn't match the URL registered with the OAuth provider	Make sure the provider's redirect URL and `N8N_EDITOR_BASE_URL` use the exact same scheme and hostname
`cloudflared`: "tunnel not found"	Wrong tunnel name, or `credentials-file` path in `config.yml` doesn't match what `tunnel create` wrote	Run `cloudflared tunnel list` and correct the `tunnel:` and `credentials-file:` lines
Cloudflare Tunnel connects but the hostname 404s	No ingress rule matches the hostname, or the catch-all rule is missing	Confirm the `hostname:` line in `config.yml` matches exactly, and that a `service: http_status:404` catch-all is the last ingress entry
ngrok: `ERR_NGROK_108`	No authtoken configured	Run `ngrok config add-authtoken <token>`
ngrok hostname changes every restart	Free tier does not reserve hostnames	Buy a reserved domain, or accept re-configuring `WEBHOOK_URL` each session
localtunnel: connection refused after a few hours	The free relay dropped the session; there is no uptime guarantee	Restart `lt`, or move to Cloudflare Tunnel for anything long-running
SSH reverse tunnel: remote port not reachable externally	`GatewayPorts` not enabled on the remote sshd	Set `GatewayPorts yes` in `/etc/ssh/sshd_config` on the remote host and restart `sshd`
SSH tunnel dies silently	No keep-alive, connection dropped by an idle NAT timeout	Use `autossh` with `ServerAliveInterval` / `ServerAliveCountMax`, as shown above

---

Further reading

- https://docs.n8n.io/changelog/v20-breaking-changes
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/deployment
- https://developers.cloudflare.com/cloudflare-one/connections/connect-networks/
- https://ngrok.com/docs/http/

Chapter 12: Platform-as-a-Service: Render, Railway, and Fly.io

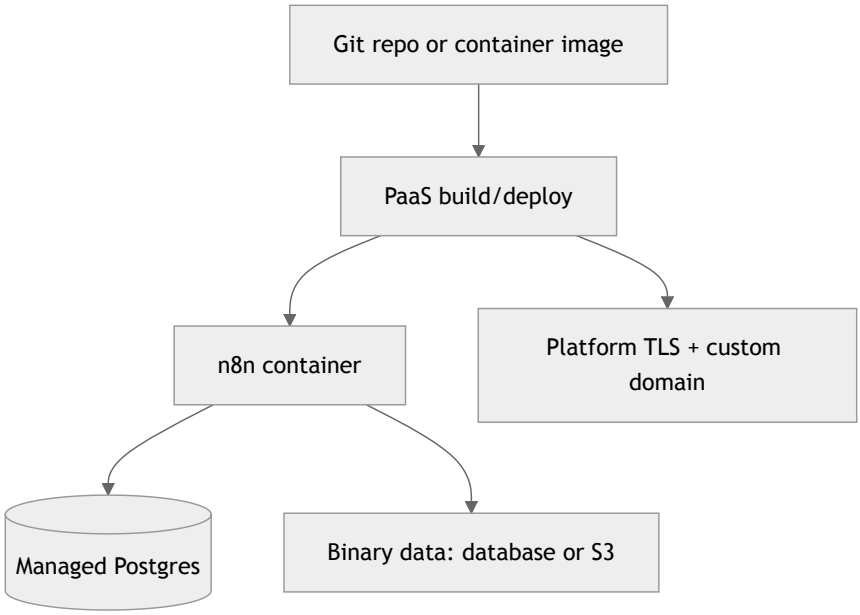
Overview

Platform-as-a-Service providers run your container, terminate TLS, and give you a managed Postgres database, in exchange for less control than a VPS and a bill that grows with usage. For n8n, PaaS trades away one thing that self-hosting takes for granted: a persistent local disk. Every platform in this chapter either wipes the container's filesystem on redeploy, restarts it on a schedule, or won't guarantee your workflow ends up on the same instance twice. That single fact, more than any dashboard difference, is what shapes every recipe below.

This chapter covers three platforms, in the order most people should try them:

1. **Render** — the easiest path: connect a repo or an image, add a managed Postgres instance, done.
2. **Railway** — a one-click community template, or a manual Docker deploy for more control.
3. **Fly.io** — `fly launch` from the n8n image, plus a volume and a Postgres app, for people who want to think about regions and machine sizing.

Heroku is not covered in depth here. Its free tier is gone, and it is no longer where people deploy n8n — see the short note at the end of section 1.



Diagram

Prerequisites

- A pinned n8n image reference, `n8nio/n8n:2.38.1` — every platform in this chapter deploys that image directly; none of them need a Dockerfile you maintain.
- An `N8N_ENCRYPTION_KEY`, generated once with `openssl rand -hex 32` and set as a platform secret, not committed anywhere.
- Accounts on the platform(s) you're deploying to, and their CLI installed where one exists (`railway`, `flyctl`).

- A domain you can add a CNAME or A record to, if you want a custom hostname instead of the platform's default subdomain.

---

1. What Every PaaS Deployment Needs, and Why

Three constraints apply across Render, Railway, and Fly.io, and they come directly from how these platforms run containers:

Disks are ephemeral or single-instance. Render's and Railway's default filesystem does not survive a redeploy; Fly.io volumes do persist, but only attach to a single machine, which blocks horizontal scaling. n8n's default binary data mode, `filesystem`, writes binary data (files passing through workflows) straight to that disk. On any of these platforms, set:

```
N8N_DEFAULT_BINARY_DATA_MODE=database
```

or point it at S3-compatible storage for large binary payloads — filesystem mode isn't an option once the disk isn't guaranteed to persist. See "Docker Compose in Queue Mode: Main, Workers, Webhooks, and Runners" for the full binary data mode discussion.

Use the platform's managed Postgres, not SQLite. SQLite lives on the same ephemeral disk as everything else, so it disappears exactly when you least want it to — on a redeploy. Every platform here offers a managed Postgres instance for a few dollars a month; use it.

Set the URL variables to the platform hostname, and pin the image tag. Same four variables as the tunnel chapter — `N8N_HOST`, `N8N_PROTOCOL=https`, `WEBHOOK_URL`, `N8N_EDITOR_BASE_URL` — pointed at whatever hostname the platform gives you (or your custom domain once you add one). And pin `n8nio/n8n:2.38.1` explicitly; never deploy `latest` on a platform that redeploys automatically on every push, or an upstream release can land on your production instance without you choosing it.

A short note on Heroku: its free dyno tier ended in 2022, and it is rarely where anyone stands up a new n8n instance now — Render, Railway, and Fly.io all offer a comparable or better tier with less buildpack fuss around Docker images. If you already run infrastructure on Heroku, the recipe is the same shape as Render's: a container deploy, Heroku Postgres as the add-on, `heroku config:set` for variables, and a custom domain through Heroku's Automated Certificate Management. This chapter does not repeat that walkthrough in full.

---

2. Render

2.1 Managed Postgres

Create a Postgres instance from the Render dashboard first — New → PostgreSQL. Render gives you a connection string and separate host/port/user/password/database fields; you want the individual fields, not the combined URL, because n8n's `DB_POSTGRESDB_*` variables are discrete.

2.2 The web service

New → Web Service → Deploy an existing image, and give it `n8nio/n8n:2.38.1`. Render expects your service to listen on the port it assigns via the `PORT` environment variable, so tell n8n to use it:

```
N8N_PORT=$PORT
```

Set the rest of the environment under the service's Environment tab:

```
N8N_ENCRYPTION_KEY=<generated once, kept secret>
DB_TYPE=postgresdb
DB_POSTGRESDB_HOST=<from the Render Postgres instance>
DB_POSTGRESDB_PORT=5432
DB_POSTGRESDB_DATABASE=<from the Render Postgres instance>
DB_POSTGRESDB_USER=<from the Render Postgres instance>
DB_POSTGRESDB_PASSWORD=<from the Render Postgres instance>
DB_POSTGRESDB_SSL_ENABLED=true
N8N_HOST=n8n.onrender.com
N8N_PROTOCOL=https
WEBHOOK_URL=https://n8n.onrender.com/
N8N_EDITOR_BASE_URL=https://n8n.onrender.com/
N8N_DEFAULT_BINARY_DATA_MODE=database
N8N_RUNNERS_MODE=external
N8N_RUNNERS_BROKER_LISTEN_ADDRESS=0.0.0.0
N8N_RUNNERS_AUTH_TOKEN=<generated once, kept secret>
```

Use Render's Secret Files or marked-secret environment variables for `N8N_ENCRYPTION_KEY` and `N8N_RUNNERS_AUTH_TOKEN` so they don't show up in build logs.

2.3 Custom domain

Settings → Custom Domains → Add Custom Domain, then create the CNAME Render gives you at your DNS provider. Render provisions a Let's Encrypt certificate automatically once the CNAME resolves. Update the four URL variables to the custom hostname and redeploy.

2.4 Scaling limits

Render's autoscaling adds more copies of the same service, which doesn't work for n8n's main process the way it does for a stateless web app — two `main` instances writing to the same execution data race unless you're running queue mode with workers split out as their own service. Keep the instance count at 1 and scale instance size (CPU/RAM) up before adding workers. If you outgrow one instance, see "Docker Compose in Queue Mode: Main, Workers, Webhooks, and Runners" for the shape a scaled deployment needs, and run the extra worker process on Render as a second Background Worker service pointed at the same database.

---

3. Railway

3.1 One-click template

Railway maintains a community-supported n8n template — search “n8n” in the Railway template gallery, or deploy from `railway.com/template/n8n`. It provisions a Postgres plugin and the n8n service together, wires the `DATABASE_URL` automatically, and generates a `*.up.railway.app` domain with TLS out of the box. Open the deployed service’s Variables tab and confirm `N8N_ENCRYPTION_KEY` was generated (some template versions require you to set it yourself) — if it wasn’t, generate and set one before you create any credentials, since regenerating it later invalidates every stored credential.

3.2 Manual Docker deploy

For control over the image tag and every variable, skip the template and deploy the image directly:

```
npm install -g @railway/cli
railway login
railway init
railway add --database postgres
```

`railway add --database postgres` provisions a Postgres plugin in the project and exposes it as reference variables. Add the n8n service:

```
railway add --service n8n --image n8nio/n8n:2.38.1
```

Then set variables, referencing the Postgres plugin’s own variables so you don’t copy secrets by hand:

```
railway variables --service n8n --set "N8N_ENCRYPTION_KEY=$(openssl rand -hex 32)" \
  --set "DB_TYPE=postgresdb" \
  --set "DB_POSTGRESDB_HOST=\${{Postgres.PGHOST}}" \
  --set "DB_POSTGRESDB_PORT=\${{Postgres.PGPORT}}" \
  --set "DB_POSTGRESDB_DATABASE=\${{Postgres.PGDATABASE}}" \
  --set "DB_POSTGRESDB_USER=\${{Postgres.PGUSER}}" \
  --set "DB_POSTGRESDB_PASSWORD=\${{Postgres.PGPASSWORD}}" \
  --set "N8N_DEFAULT_BINARY_DATA_MODE=database" \
  --set "N8N_PORT=8080"
```

Railway assigns a public domain per service under Settings → Networking → Generate Domain; use that hostname (or a custom domain added the same place) for `N8N_HOST`, `WEBHOOK_URL`, and `N8N_EDITOR_BASE_URL`, all set to `https://`.

3.3 Custom domain and scaling limits

Settings → Networking → Custom Domain, then add the CNAME Railway shows you; certificates are automatic once DNS resolves. Railway bills by usage (CPU- and memory-seconds) rather than a fixed instance size — cheap for light use, easy to underestimate for a busy workload, so watch the usage dashboard early on. Like Render, running more than one replica of the n8n service without queue mode corrupts concurrent executions; keep replicas at 1 unless you’ve split out workers.

---

4. Fly.io

Fly.io runs your container as a “machine” in a region you choose, and is the most VPS-like of the three — you get a real attached volume, but you also do slightly more of the configuration yourself.

4.1 Launch

```
brew install flyctl
fly auth login
fly launch --image n8nio/n8n:2.38.1 --name n8n-prod --region sea --no-deploy
```

`fly launch` writes a `fly.toml`. Edit it before the first deploy to fix the internal port and add a volume mount:

```
# fly.toml
app = "n8n-prod"
primary_region = "sea"

[build]
  image = "n8nio/n8n:2.38.1"

[env]
  N8N_PORT = "5678"
  DB_TYPE = "postgresdb"
  N8N_HOST = "n8n-prod.fly.dev"
  N8N_PROTOCOL = "https"
  WEBHOOK_URL = "https://n8n-prod.fly.dev/"
  N8N_EDITOR_BASE_URL = "https://n8n-prod.fly.dev/"
  N8N_DEFAULT_BINARY_DATA_MODE = "database"
  GENERIC_TIMEZONE = "America/Los_Angeles"

[[mounts]]
  source = "n8n_data"
  destination = "/home/node/.n8n"

[http_service]
  internal_port = 5678
  force_https = true
  min_machines_running = 1
```

Create the volume before deploying (it isn’t created automatically from `[[mounts]]`):

```
fly volumes create n8n_data --region sea --size 3
```

A Fly volume attaches to exactly one machine in one region and cannot be shared across machines. That’s fine for `/home/node/.n8n` (n8n’s config; you’re using Postgres below, not the SQLite file) but it’s one more reason binary data belongs in the database or S3, not the filesystem, once you might ever run more than one machine.

4.2 Postgres app

Fly’s managed Postgres is itself a small Fly app:

```
fly postgres create --name n8n-db --region sea --initial-cluster-size 1
fly postgres attach n8n-db --app n8n-prod
```

`postgres attach` creates a database and user for `n8n-prod` and sets a `DATABASE_URL` secret on it automatically. n8n doesn’t read `DATABASE_URL` directly, so set the discrete variables instead, using the values `postgres attach` printed:

```
fly secrets set --app n8n-prod \
  DB_POSTGRESDB_HOST=n8n-db.flycast \
  DB_POSTGRESDB_PORT=5432 \
  DB_POSTGRESDB_DATABASE=n8n_prod \
```

```
  DB_POSTGRESDB_USER=n8n_prod \
  DB_POSTGRESDB_PASSWORD='<from postgres attach output>' \
  N8N_ENCRYPTION_KEY="$(openssl rand -hex 32)" \
  N8N_RUNNERS_AUTH_TOKEN="$(openssl rand -hex 24)"
```

Then deploy:

```
fly deploy --app n8n-prod
```

4.3 Custom domain and scaling limits

```
fly certs create n8n.example.com --app n8n-prod
```

Fly prints the DNS records to add; once they resolve, `fly certs check n8n.example.com` confirms the certificate issued. Update the four URL variables to the custom hostname and redeploy.

Fly machines can scale to zero when idle, which breaks webhooks — a sleeping machine can't receive the request that would wake it. `min_machines_running = 1` (set above) keeps one instance up. As with Render and Railway, `fly scale count` above 1 for the main machine is not safe without queue mode: the volume attaches to one machine only, so a second machine has no access to `/home/node/.n8n`.

---

5. External Task Runners on a PaaS

Task runners have run every Code node since n8n 2.0. Internal mode runs the runner as a child process inside the same container, which works but is not the supported production mode. External mode is a second service, on all three platforms — here's the shape for Fly.io, since it's the platform where you write the most configuration by hand.

Add a second Fly app for the runner, or a second process in the same app's `fly.toml` under `[processes]`:

```
[processes]
  app = ""
  runner = ""

[[services]]
  processes = ["app"]
  internal_port = 5678

[env]
  N8N_RUNNERS_MODE = "external"
  N8N_RUNNERS_BROKER_LISTEN_ADDRESS = "0.0.0.0"
  N8N_RUNNERS_AUTH_TOKEN = "shared-secret-here"
```

Then a second Fly app running the runner image, pointed back at the n8n app's internal Fly DNS name:

```
fly launch --image n8nio/runners:2.38.1 --name n8n-prod-runner --no-deploy
fly secrets set --app n8n-prod-runner \
  N8N_RUNNERS_TASK_BROKER_URI=http://n8n-prod.internal:5679 \
  N8N_RUNNERS_AUTH_TOKEN=shared-secret-here \
  N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT=15
fly deploy --app n8n-prod-runner
```

`n8n-prod.internal` is Fly's private networking DNS for apps in the same organization. Render and Railway support the same pattern: a second service from the `n8nio/runners:2.38.1` image, `N8N_RUNNERS_TASK_BROKER_URI` pointed at the main service's internal hostname on port 5679, and a shared `N8N_RUNNERS_AUTH_TOKEN`.

---

6. Troubleshooting Common Issues

Issue	Cause	Solution
Uploaded/binary files vanish after a redeploy	`N8N_DEFAULT_BINARY_DATA_MODE=filesystem` on an ephemeral disk	Set `N8N_DEFAULT_BINARY_DATA_MODE=database`, or point it at S3-compatible storage
Two executions of the same workflow interleave or corrupt data	More than one instance of the main n8n service running without queue mode	Set replica/instance count back to 1, or move to queue mode with dedicated worker services
Database connection refused	Wrong host/port, or the managed Postgres requires SSL and `DB_POSTGRESDB_SSL_ENABLED` isn't set	Re-check the platform's connection details; add `DB_POSTGRESDB_SSL_ENABLED=true` where the provider requires TLS
Webhook never arrives, works from the platform's own test button	`WEBHOOK_URL` / `N8N_EDITOR_BASE_URL` still point at a previous hostname	Update all four URL variables to the current hostname and redeploy
Custom domain shows a certificate error	DNS hasn't propagated, or the CNAME/A record is wrong	Recheck the exact record the platform asked for; wait for propagation before assuming the cert is broken
Fly machine never responds to the first webhook after idling	`min_machines_running` unset, machine scaled to zero	Set `min_machines_running = 1` in `fly.toml`
Task runner never connects	Auth token mismatch, or wrong internal hostname/port	Confirm both services share the exact same `N8N_RUNNERS_AUTH_TOKEN`, and that the broker URI uses port 5679
Deploy pulls in an unplanned n8n version	Image tag left as `stable` or unpinned	Pin the exact tag, `n8nio/n8n:2.38.1`, on every service

---

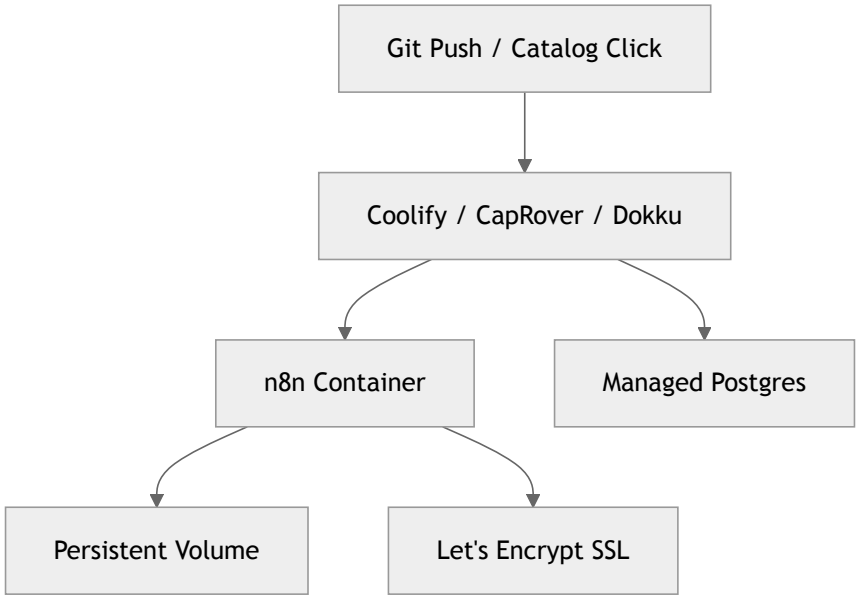
Further reading

- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/handle-binary-data
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/deployment
- https://render.com/docs/deploy-an-image
- https://docs.railway.com/guides/postgresql
- https://fly.io/docs/postgres/

Chapter 13: Self-Hosted PaaS: Coolify, CapRover, and Dokku

Overview

Coolify, CapRover, and Dokku all sit on top of Docker and give you Heroku-style deploys — git push or a UI click — on infrastructure you control. Coolify is the newest of the three and today the most widely used self-hosted PaaS for running n8n: it ships an n8n entry in its service catalog, and its Docker Compose resource type accepts the reference stack from earlier chapters almost unchanged. CapRover and Dokku remain solid choices, particularly if you already run other apps on one of them. This chapter covers all three: installation, deploying n8n, environment variables, domains and SSL, scaling, CI/CD, monitoring, and backups — and, for each platform, whether adding the task runners sidecar is practical.



Diagram

Prerequisites

- A VM or VPS (Ubuntu 22.04+ or Debian 12+) with root or sudo access and at least 2 GB RAM
- A domain name with DNS pointing at the server’s IP
- SSH access with a key pair
- Familiarity with Docker and basic Linux CLI

1. Choosing a Platform

All three give you a domain, TLS, and a place to set environment variables without hand-writing a `compose.yml` on the server. Coolify's UI is the most opinionated about persistent volumes and managed databases, which is why it's the default recommendation now. CapRover has the longest track record and the simplest One-Click App model. Dokku is the lightest — closer to bare `git push` than a full UI — and best if you're comfortable on the command line and want the smallest footprint.

2. Deploying n8n with Coolify

2.1 Installing Coolify

Coolify installs with a single script on a fresh server:

```
curl -fsSL https://cdn.coollabs.io/coolify/install.sh | bash
```

The installer sets up Coolify itself as a Docker stack and prints the URL for its own dashboard, typically `http://<server-ip>:8000`. Finish setup there: create the admin account and add the server itself as a deployment target (Coolify can also manage remote servers over SSH, but for a single-box setup, add `localhost`).

2.2 One-Click n8n from the Catalog

Coolify maintains a services catalog with n8n as a predefined stack (n8n plus its own Postgres). From **Projects** → **New Resource** → **Services**, search for `n8n` and deploy it. This is the fastest path and a reasonable one for evaluation, but it pins whatever version the catalog entry ships — check the resource's compose definition after deploying and pin `n8nio/n8n:2.38.1` explicitly if it doesn't already, so an unattended catalog update doesn't move you to an untested version.

2.3 Docker Compose Resource (Reference Stack)

For control over versions and the task runners sidecar, skip the catalog and create a **Docker Compose** resource instead: **Projects** → **New Resource** → **Docker Compose**. Paste the reference stack from the earlier reference chapter — the same `compose.yml` with `postgres`, `n8n`, and `runners` services — into Coolify's compose editor. Coolify parses it, lets you set the `.env` values (`N8N_ENCRYPTION_KEY`, `N8N_RUNNERS_AUTH_TOKEN`, `POSTGRES_PASSWORD`, `GENERIC_TIMEZONE`) as encrypted environment variables in its own UI rather than a plaintext file, and manages the named volumes it declares.

Remove the `ports:` mapping from the `n8n` service before deploying — Coolify's own reverse proxy handles the public-facing port, and a container also publishing to the host directly can conflict with it. Coolify injects its proxy as a sidecar on the same Docker network, so `n8n` stays reachable internally without a host port.

2.4 Setting the Domain

Under the resource's **Domains** setting, enter your hostname, for example `n8n.example.com`. Coolify requests a Let's Encrypt certificate automatically and — this is the part worth calling out — sets `N8N_HOST`, `N8N_PROTOCOL`, `WEBHOOK_URL`, and `N8N_EDITOR_BASE_URL` for you based on that domain, so you don't hand-edit them into the compose environment. If you also set them via the compose file's environment block, Coolify's generated values and yours can disagree; let the Domains field be the source of truth for these four variables and remove them from your `compose.yml` and `.env`.

2.5 Managed Postgres Resource

Rather than running `postgres` as a service inside the same compose resource, Coolify can provision Postgres as its own **Database** resource with its own backups, restarts, and update path independent of the n8n container. Create it from **New Resource** → **Databases** → **PostgreSQL**, then point `DB_POSTGRESDB_HOST` at the internal hostname Coolify assigns it and `DB_POSTGRESDB_PASSWORD` at the generated credential. This is the better long-term setup if you'll run more than one app against the same Postgres instance, or want database backups scheduled independently of the n8n deploy lifecycle.

2.6 Task Runners in Coolify

Because a Coolify Docker Compose resource deploys the whole file as one unit on the same internal network, the `runners` service from the reference stack works unmodified — no special Coolify configuration needed beyond what's already in the compose file. Set `N8N_RUNNERS_TASK_BROKER_URI=http://n8n:5679` (Coolify resolves service names the same way plain Compose does) and give `N8N_RUNNERS_AUTH_TOKEN` the same value on both `n8n` and `runners`. If you used the One-Click catalog entry instead of a custom compose resource, check whether its definition includes a `runners` service; if not, add one by editing the resource's compose file directly in Coolify before redeploying.

2.7 Backups

For the managed Postgres resource, Coolify has a built-in **Backups** tab: schedule periodic dumps to local disk or S3-compatible storage from there. For the `n8n_data` volume — workflows, credentials, settings — Coolify doesn't back this up automatically; use the same `docker run` plus `tar` pattern from "Docker Deep Dive: docker run, Volumes, Secrets, and Air-Gapped Installs" against the volume it created, or schedule it as a cron job on the host.

3. Installing CapRover

```
curl -sL https://captain.install.caprover.com | bash
```

Follow the interactive prompts to choose the HTTP/HTTPS ports and set a root domain and admin password. Then install the CLI locally and log in:

```
npm install -g caprover
caprover login
```

4. Installing Dokku

```
wget https://raw.githubusercontent.com/dokku/dokku/v0.35.18/bootstrap.sh
sudo DOKKU_TAG=v0.35.18 bash bootstrap.sh
```

The installer prompts for your root domain and an SSH public key for the `dokku` user. Everything after this runs through `dokku` commands over SSH.

5. Provisioning Postgres

CapRover: Apps → One-Click Apps/Databases → PostgreSQL → Deploy. CapRover generates credentials and a reachable host name (`srv-captain--<app-name>`) that you'll reference from the n8n app's environment variables.

Dokku:

```
sudo dokku plugin:install https://github.com/dokku/dokku-postgres.git
dokku postgres:create n8n-db
dokku postgres:link n8n-db n8n-app
dokku postgres:info n8n-db
```

`postgres:link` sets `DATABASE_URL` and a set of `DOKKU_POSTGRES_*` variables on the linked app; you still need to map those into the `DB_POSTGRESDB_*` names n8n expects (step 6).

6. Configuring Environment Variables

6.1 CapRover

Apps → n8n → App Configs → Environment Variables:

Key	Value
`N8N_ENCRYPTION_KEY`	output of `openssl rand -hex 32`
`DB_TYPE`	`postgresdb`
`DB_POSTGRESDB_HOST`	`srv-captain--n8n-db`
`DB_POSTGRESDB_DATABASE`	database name from the One-Click deploy
`DB_POSTGRESDB_USER`	user from the One-Click deploy
`DB_POSTGRESDB_PASSWORD`	password from the One-Click deploy
`N8N_HOST`	`n8n.example.com`
`N8N_PROTOCOL`	`https`
`WEBHOOK_URL`	`https://n8n.example.com/`
`N8N_EDITOR_BASE_URL`	`https://n8n.example.com/`
`N8N_PROXY_HOPS`	`1`
`GENERIC_TIMEZONE`	your IANA timezone

Under **Persistent Directories**, mount `/home/node/.n8n` so workflows and credentials survive redeploys.

6.2 Dokku

```
dokku config:set n8n-app \
  N8N_ENCRYPTION_KEY=$(openssl rand -hex 32) \
  N8N_HOST=n8n.example.com \
  N8N_PROTOCOL=https \
  WEBHOOK_URL=https://n8n.example.com/ \
  N8N_EDITOR_BASE_URL=https://n8n.example.com/ \
  N8N_PROXY_HOPS=1 \
  GENERIC_TIMEZONE=America/Los_Angeles \
  DB_TYPE=postgresdb

dokku storage:mount n8n-app /var/lib/dokku/data/storage/n8n-data:/home/node/.n8n
```

`postgres:link` already set the `DB_POSTGRESDB_*` equivalents through `DATABASE_URL`; confirm with `dokku config:show n8n-app` and set `DB_POSTGRESDB_HOST`, `DB_POSTGRESDB_USER`, `DB_POSTGRESDB_PASSWORD`, and `DB_POSTGRESDB_DATABASE` explicitly if n8n doesn't parse the connection string directly.

7. Deploying n8n

Both platforms need to run the prebuilt `n8nio/n8n:2.38.1` image rather than build from source, since n8n isn't a buildpack-detectable Node app in the conventional sense — it needs the image's baked-in runners protocol and file layout.

CapRover: Apps → n8n → Deployment → Method: **Deploy an Image**, and specify `n8nio/n8n:2.38.1` directly. This skips the Dockerfile/GitHub build path entirely and pulls the pinned tag.

Dokku: point the app at the image instead of building it:

```
dokku git:from-image n8n-app n8nio/n8n:2.38.1
```

8. Task Runners on CapRover and Dokku

CapRover supports this reasonably well: create a second app, `n8n-runners`, deploy the `n8nio/runners:2.38.1` image to it the same way (Deploy an Image), and set `N8N_RUNNERS_TASK_BROKER_URI=http://srv-captain--n8n-app:5679` and a shared `N8N_RUNNERS_AUTH_TOKEN` on both apps. CapRover's internal `srv-captain--*` DNS names resolve between apps on the same server, so this works without exposing the runner publicly. On the n8n app, also set `N8N_RUNNERS_MODE=external` and `N8N_RUNNERS_BROKER_LISTEN_ADDRESS=0.0.0.0`.

Dokku is not practical for this today. Each Dokku app is its own isolated container without a straightforward built-in way to address another app's container by internal DNS name the way CapRover's `srv-captain--*` convention does; you'd need the network plugin and manual container networking to replicate it, which works but is fragile enough that it's easier to run Dokku only for n8n itself and put Postgres and the runners sidecar on a plain Compose host instead, using Dokku for the app layer alone. If you need queue mode with workers and runners on Dokku specifically, that's a sign to move to the plain Compose or Kubernetes chapters instead.

9. Custom Domains and SSL

CapRover: App → Settings → Custom Domain, then **Enable HTTPS**; CapRover requests the Let's Encrypt certificate itself.

Dokku:

```
dokku domains:add n8n-app n8n.example.com
sudo dokku plugin:install https://github.com/dokku/dokku-letsencrypt.git
dokku config:set --no-restart n8n-app DOKKU_LETSENCRYPT_EMAIL=you@example.com
dokku letsencrypt:enable n8n-app
dokku letsencrypt:cron-job --add
```

The cron job handles renewal; without it, certificates expire silently.

10. Scaling and Zero-Downtime Deploys

CapRover: App → Scaling → increase Instance Count; CapRover waits for new instances to pass their health check before removing old ones.

Dokku:

```
dokku ps:scale n8n-app web=2
```

Running more than one n8n instance behind either platform's load balancer without queue mode risks webhook and execution-state inconsistencies — n8n's regular mode assumes one process owns the database connection for scheduled triggers. Scale the web process for HTTP throughput only if you've moved to queue mode first (see "Docker Compose in Queue Mode: Main, Workers, Webhooks, and Runners").

11. CI/CD Integration

```yaml
# .github/workflows/caprover.yml
name: CapRover Deploy
on:
  push:
    branches: [main]
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: caprover/deploy-from-github@main
        with:
          server-url: https://captain.yourdomain.com
          app-name: n8n
          branch: main
          token: ${{ secrets.CAPROVER_TOKEN }}
```

```yaml
# .github/workflows/dokku.yml
name: Dokku Deploy
on:
  push:
    branches: [main]
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: dokku/github-action@v1
        with:
          git_remote_url: ssh://dokku@your-server.com:22/n8n-app
          ssh_private_key: ${{ secrets.DOKKU_SSH_KEY }}
```

Since n8n deploys from a pinned image rather than a build, these pipelines are mainly useful for bumping the pinned tag through a reviewed pull request rather than for building application code.

12. Monitoring and Logs

CapRover: App → Logs streams stdout/stderr live; the dashboard shows basic CPU and memory charts per app.

Dokku:

```
dokku logs n8n-app --tail
dokku drains:add n8n-app syslog+tls://logs.papertrailapp.com:12345
```

13. Backups and Migrations

Coolify: see section 2.7.

CapRover: the Postgres One-Click App has its own Backup tab for scheduled dumps to S3-compatible storage. Back up `/home/node/.n8n` (the Persistent Directory) separately with `scp` or a cron job on the host.

Dokku:

```
dokku postgres:export n8n-db > n8n-backup.sql
# restore:
dokku postgres:import n8n-db < n8n-backup.sql
tar czf n8n_data_$(date +%F).tar.gz -C /var/lib/dokku/data/storage/n8n-data .
```

14. Troubleshooting Common Issues

Issue	Cause	Solution
Coolify proxy returns 502 on the n8n domain	`ports:` still published on the `n8n` service, conflicting with Coolify's proxy	Remove the host `ports:` mapping; let Coolify's proxy reach the container over the internal network
Coolify-set URL variables don't match what n8n reports	Domain also set manually in the compose environment block	Remove `N8N_HOST` / `WEBHOOK_URL` / `N8N_EDITOR_BASE_URL` from the compose file; let the Domains field own them
CapRover captain UI unreachable	Firewall blocking port 3000	Open port 3000, or whichever port was chosen at install, in the server's firewall
Dokku deploy fails with "no such image"	`git:from-image` given an unpinned or misspelled tag	Confirm the exact tag with `docker images` on the server after `docker pull n8nio/n8n:2.38.1`
SSL renewal fails on Dokku	DNS not pointing at the server, or renewal cron never installed	Verify the A record, then `dokku letsencrypt:cron-job --add`
Database connection refused	Postgres plugin not linked, or `DB_POSTGRESDB_*` vars unset after linking	Re-run `postgres:link`, confirm with `config:show`/App Configs
Workflows silently double-execute after scaling instances	Multiple n8n instances in regular mode, no queue mode	Scale only after switching to queue mode; see "Docker Compose in Queue Mode: Main, Workers, Webhooks, and Runners"

Further reading

- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/enable-queue-mode
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/deployment

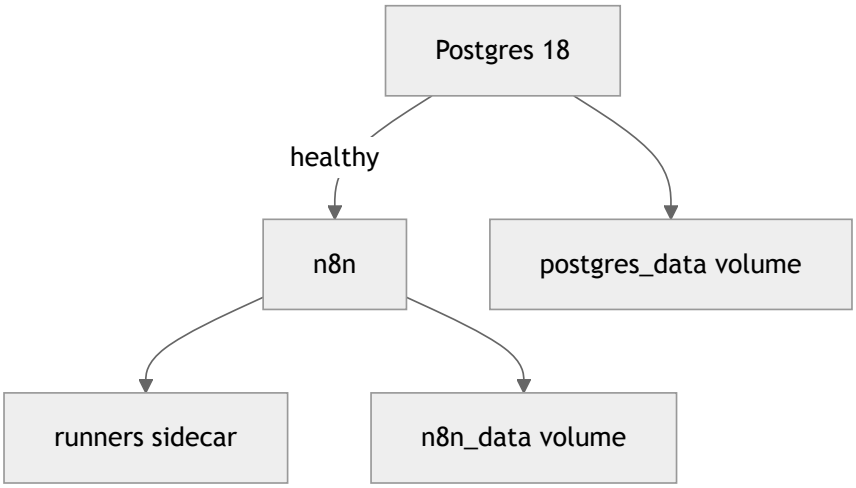
---

Chapter 14: Docker Compose with PostgreSQL

Overview

This chapter builds the reference stack into a working Postgres-backed deployment: `postgres:18` with a healthcheck, pinned `n8nio/n8n:2.38.1` and `n8nio/runners:2.38.1` images, the encryption key, execution pruning, and the URL variables that make webhooks resolve correctly behind a reverse proxy. It also covers backups with `pg_dump`, rolling upgrades, and a few advanced scenarios: custom Postgres images, TLS between n8n and Postgres, and adding pgAdmin.

Postgres over SQLite buys you concurrent writes, better behavior under load, and a database you can back up and restore with standard tools instead of copying a file mid-write.



Diagram

Prerequisites

- Docker Engine with the Compose v2 plugin (`docker compose version` should print a `v2.x` line)
- A host with at least 2 GB RAM; Postgres itself is not heavy, but give it headroom
- `openssl` for generating the encryption key and runner token

1. The Compose Stack

```
mkdir n8n-postgres && cd n8n-postgres
```

`.env`:

```
# Generate with: openssl rand -hex 32
N8N_ENCRYPTION_KEY=replace-with-64-hex-characters
N8N_RUNNERS_AUTH_TOKEN=replace-with-a-long-random-string

POSTGRES_USER=n8n
POSTGRES_PASSWORD=replace-me
POSTGRES_DB=n8n

N8N_HOST=n8n.example.com
GENERIC_TIMEZONE=America/Los_Angeles
```

Generate the two secrets instead of typing placeholders:

```
openssl rand -hex 32   # -> N8N_ENCRYPTION_KEY
openssl rand -hex 32   # -> N8N_RUNNERS_AUTH_TOKEN
```

`compose.yml`:

```
services:
  postgres:
    image: postgres:18
    restart: unless-stopped
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    volumes:
      - postgres_data:/var/lib/postgresql
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER} -d ${POSTGRES_DB}"]
      interval: 10s
      timeout: 5s
      retries: 5

  n8n:
    image: n8nio/n8n:2.38.1
    restart: unless-stopped
    ports:
      - "127.0.0.1:5678:5678"
    environment:
      N8N_ENCRYPTION_KEY: ${N8N_ENCRYPTION_KEY}
      DB_TYPE: postgresdb
      DB_POSTGRESDB_HOST: postgres
      DB_POSTGRESDB_PORT: 5432
      DB_POSTGRESDB_DATABASE: ${POSTGRES_DB}
      DB_POSTGRESDB_USER: ${POSTGRES_USER}
      DB_POSTGRESDB_PASSWORD: ${POSTGRES_PASSWORD}
      N8N_HOST: ${N8N_HOST}
      N8N_PROTOCOL: https
      WEBHOOK_URL: https://${N8N_HOST}/
      N8N_EDITOR_BASE_URL: https://${N8N_HOST}/
      N8N_PROXY_HOPS: 1
      GENERIC_TIMEZONE: ${GENERIC_TIMEZONE}
      TZ: ${GENERIC_TIMEZONE}
      N8N_RUNNERS_MODE: external
      N8N_RUNNERS_BROKER_LISTEN_ADDRESS: 0.0.0.0
      N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
      N8N_NATIVE_PYTHON_RUNNER: "true"
      N8N_DEFAULT_BINARY_DATA_MODE: filesystem
      EXECUTIONS_DATA_PRUNE: "true"
      EXECUTIONS_DATA_MAX_AGE: 336
      N8N_DIAGNOSTICS_ENABLED: "false"
    volumes:
      - n8n_data:/home/node/.n8n
      - n8n_files:/home/node/.n8n-files
```

```
    depends_on:
      postgres:
        condition: service_healthy

  runners:
    image: n8nio/runners:2.38.1
    restart: unless-stopped
    environment:
      N8N_RUNNERS_TASK_BROKER_URI: http://n8n:5679
      N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
      N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT: 15
    depends_on:
      - n8n

volumes:
  postgres_data:
  n8n_data:
  n8n_files:
```

The `postgres_data` volume mounts at `/var/lib/postgresql`, not `/var/lib/postgresql/data`. The official `postgres:18` image moved its data directory; mounting the old `.../data` path crash-loops the container with an "upgrading the Docker image without upgrading the underlying database" error, because Postgres finds an empty directory where it expects its existing files. This is a new stack — if you're instead carrying a volume forward from an older `postgres:1x` deployment that used the `.../data` path, that's a major-version Postgres upgrade (dump on the old version, restore on the new), not a path change on the same data.

`EXECUTIONS_DATA_PRUNE` with `EXECUTIONS_DATA_MAX_AGE: 336` (14 days, in hours) keeps the execution history table from growing indefinitely — with SQLite this mattered less because you'd notice the file size; with Postgres it's easy to forget until a `pg_dump` starts taking minutes.

If you're serving n8n over plain HTTP on a non-localhost address instead of through a reverse proxy with TLS, you additionally need `N8N_SECURE_COOKIE=false` — not recommended outside of testing, since it sends the session cookie without the `Secure` flag.

2. Launching and Verifying the Stack

```
docker compose up -d
docker compose ps
```

Confirm `postgres` reports `healthy` before worrying about `n8n`'s state — `depends_on: condition: service_healthy` makes `n8n` wait for it, but the first boot still takes a few seconds.

```
docker compose logs -f postgres
docker compose logs -f n8n
```

Watch for a migrations-complete line in the n8n logs — this is n8n creating its schema in Postgres on first start.

3. Accessing n8n

Visit `https://n8n.example.com` (or `http://localhost:5678` if you're testing without a reverse proxy in front). The first visit prompts you to create the owner account. Create a workflow, run it, then confirm the execution shows up in Postgres:

```
docker compose exec postgres psql -U n8n -d n8n -c "SELECT count(*) FROM execution_entity;"
```

This uses the literal `n8n` user and database name from the `.env` file above. `${POSTGRES_USER}` and `${POSTGRES_DB}` only exist inside `compose.yml`, where Compose reads and substitutes them from `.env` itself — your shell never sees them, so typing `${POSTGRES_USER}` at the command line expands to nothing. If you changed either value in `.env`, use your actual values here instead of `n8n`.

4. Backups and Restores

4.1 Backup with pg_dump

Run `pg_dump` inside the running Postgres container rather than copying the raw data directory — it produces a consistent, portable SQL dump without needing to stop the database:

```
docker compose exec -T postgres pg_dump -U n8n -d n8n \
  > backups/n8n_$(date +%F).sql
```

Automate it with cron, keeping at least a week of dated dumps rather than overwriting one file.

Also back up the `n8n_data` volume (workflow files, settings) and, separately, the `N8N_ENCRYPTION_KEY` value itself — a Postgres dump alone is unreadable credentials without the matching key.

4.2 Restore

```
docker compose stop n8n
docker compose exec -T postgres psql -U n8n -d n8n \
  < backups/n8n_2026-09-05.sql
docker compose start n8n
```

As above, `n8n` / `n8n` are the literal `.env` values, not shell variables — swap them for your own if you changed `POSTGRES_USER` or `POSTGRES_DB`.

Restoring into a database that already has n8n's tables will conflict; for a clean restore, drop and recreate the database first, or restore into a fresh `postgres_data` volume.

5. Rolling Upgrades

Upgrading means bumping two tags together — `n8nio/n8n` and `n8nio/runners` should always match:

```
    image: n8nio/n8n:2.39.0
```

```
    image: n8nio/runners:2.39.0
```

```
docker compose pull
docker compose up -d
```

n8n runs its database migrations automatically on startup, before it starts accepting traffic. These migrations are **not reversible** — there is no built-in command to undo a schema change shipped this way, and rolling the image tag back down does not undo a migration that already ran. Take the `pg_dump` backup from section 4.1 immediately before every upgrade, not after; if the new version misbehaves, your rollback path is restoring that backup into a fresh Postgres, not just reverting the image tag.

Check the changelog for the target version before upgrading across a minor or major boundary for anything that also needs an environment variable change.

6. Advanced Scenarios

6.1 Custom Postgres Image

For extensions like `uuid-ossp` or `pgvector`:

```
# Dockerfile.postgres
FROM postgres:18
RUN apt-get update && apt-get install -y postgresql-18-pgvector \
  && rm -rf /var/lib/apt/lists/*
```

```
  postgres:
    build:
      context: .
      dockerfile: Dockerfile.postgres
```

6.2 TLS Between n8n and Postgres

```
  postgres:
    volumes:
      - ./certs/server.crt:/var/lib/postgresql/server.crt:ro
      - ./certs/server.key:/var/lib/postgresql/server.key:ro
      - ./postgresql.conf:/etc/postgresql/postgresql.conf:ro
    command: ["postgres", "-c", "config_file=/etc/postgresql/postgresql.conf"]

  n8n:
    environment:
      DB_POSTGRESDB_SSL_ENABLED: "true"
      DB_POSTGRESDB_SSL_CA_FILE: /certs/ca.pem
    volumes:
      - ./certs/ca.pem:/certs/ca.pem:ro
```

Set `ssl = on`, `ssl_cert_file = 'server.crt'`, and `ssl_key_file = 'server.key'` in `postgresql.conf`. On the `n8n` side, the variable is `DB_POSTGRESDB_SSL_ENABLED` — there is no `DB_POSTGRESDB_SSL` or `DB_POSTGRESDB_SSL_MODE`, and setting either does nothing. If you're using a private CA rather than a certificate a system trust store already recognizes, mount the CA certificate read-only and point `DB_POSTGRESDB_SSL_CA_FILE` at the mounted path, as above.

6.3 pgAdmin for Manual Database Access

```
services:
  pgadmin:
    image: dpage/pgadmin4:8.12
    restart: unless-stopped
    environment:
      PGADMIN_DEFAULT_EMAIL: you@example.com
      PGADMIN_DEFAULT_PASSWORD: ${PGADMIN_PASSWORD}
    ports:
      - "127.0.0.1:5050:80"
    volumes:
      - pgadmin_data:/var/lib/pgadmin
    depends_on:
      - postgres

volumes:
  pgadmin_data:
```

Merge the `pgadmin` service into the `services:` block of the main `compose.yml` and the `pgadmin_data` line into its `volumes:` block — it's shown as its own fragment here only for clarity. Bind pgAdmin to `127.0.0.1` unless you're putting it behind its own authenticated reverse proxy — it's a full database admin console.

7. Troubleshooting Common Issues

Issue	Cause	Solution
`postgres` never reports healthy	Wrong `POSTGRES_USER` / `POSTGRES_DB` in the healthcheck vs. the environment	Confirm the `pg_isready` command's `-U` / `-d` match `POSTGRES_USER` / `POSTGRES_DB` exactly
n8n can't connect to Postgres	`DB_POSTGRESDB_HOST` isn't the Compose service name	Use `postgres` (the service name), not `localhost` or an IP
`pg_dump` fails with authentication error	Wrong password, or run from outside the container without matching credentials	Run it with `docker compose exec`, using the same `.env` values, not from the host
Migrations fail partway through an upgrade	Upgrading across too many versions at once, or a corrupted prior state	Restore the pre-upgrade `pg_dump`, upgrade one minor version at a time, checking the changelog for each
Dump file has grown very large	No execution pruning configured	Set `EXECUTIONS_DATA_PRUNE=true` and a reasonable `EXECUTIONS_DATA_MAX_AGE`
SSL handshake errors after enabling TLS	Certificate CN doesn't match the host n8n connects to, or the CA isn't trusted	Regenerate the cert for the `postgres` hostname; confirm `DB_POSTGRESDB_SSL_CA_FILE` points at the mounted CA on the n8n side

Further reading

- https://docs.n8n.io/deploy/host-n8n/install-options/install-using-docker-compose
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/deployment
- https://docs.n8n.io/changelog/v20-breaking-changes

Chapter 15: Docker Compose in Queue Mode: Main, Workers, Webhooks, and Runners

---

Overview

In the stacks you have built so far, one n8n container does everything: it serves the editor, listens for webhooks, fires timers, and runs every execution. That is the right shape until a slow workflow blocks the editor, or a burst of webhooks queues up behind a long-running job.

Queue mode splits those jobs across processes. A **main** instance keeps the editor, the API, and the triggers. One or more **worker** instances do nothing but execute workflows. **Redis** sits between them as the message broker. Postgres stays where it was, holding the workflows, credentials, and execution data that both sides read and write.

You get four things from this:

- **Horizontal scaling.** Add workers to add throughput; remove them when the load drops.
- **A responsive editor.** Heavy executions no longer compete with the UI for main's event loop.
- **Fault isolation.** A worker that dies takes its in-flight executions with it, not your webhook endpoint.
- **Separate sizing.** Give workers CPU and memory. Main needs far less of both.

You will build a five-service stack — `postgres`, `redis`, `n8n` (main), `n8n-worker`, and a task runner sidecar for each n8n process — then test it, scale it, add an optional webhook processor, and update it without losing executions.

Prerequisites

- The stack from the VPS Docker Compose chapter, working, on Postgres. Queue mode over SQLite is not supported.
- Docker Engine with Compose v2. Every command is `docker compose`, with a space.
- At least 4 GB RAM. Redis is small; two n8n processes plus two runner sidecars are not.
- A pinned n8n version — `2.38.1` throughout this chapter. Every n8n process must run the same one.

---

1. How an Execution Travels

Queue mode changes where work happens, not what happens. The main instance still decides an execution should occur. It just refuses to run it.

Diagram

Step by step:

1. Main handles a timer, a poller, or an incoming webhook and creates an execution record.
2. It pushes the execution ID onto a Bull queue in Redis — the ID, not the workflow.
3. The next available worker picks the ID off the queue.
4. The worker reads the workflow and its credentials from Postgres, decrypts the credentials with the shared encryption key, and runs the workflow.
5. The worker writes results back to Postgres and posts a completion message to Redis.
6. Redis notifies main, which updates the UI and, for a webhook, returns the response to the caller still holding the connection open.

Two consequences drive most of the configuration below. Redis carries **references**, not data, except for webhook responses on their way back. And **every n8n process needs the same database and the same encryption key**, because the worker, not main, decrypts your credentials.

---

2. Three Rules You Cannot Break

Get these wrong and the stack starts, looks healthy, and fails at the first execution.

The encryption key must be identical on every n8n container. `N8N_ENCRYPTION_KEY` encrypts credentials at rest in Postgres. A worker with a different key reads the row, fails to decrypt it, and the execution errors with an unhelpful message about credentials. Generate the key once with `openssl rand -hex 32`, put it in `.env`, and reference it from main, every worker, and any webhook processor. If you have been letting n8n generate the key, read it out of `/home/node/.n8n/config` on your existing instance first, and back it up with your database dumps.

Filesystem binary data mode is not supported in queue mode.
`N8N_DEFAULT_BINARY_DATA_MODE=filesystem` writes binary payloads to a local disk the other containers cannot see. Use `database`, which stores payloads in Postgres and is the queue-mode default, or `s3` if your workflows move files big enough that you do not want them in your primary database. The old in-memory `default` mode was removed in n8n 2.0.

Each n8n process needs its own task runner sidecar. Since 2.0, every Code node runs in an external task runner. The n8n process hosts a broker on port 5679 and the `n8nio/runners` container connects to it. A runner talks to one broker only, so a worker cannot borrow main's runner. One sidecar per n8n process, each pointed at its own service name, all sharing `N8N_RUNNERS_AUTH_TOKEN`, all on the same version tag as n8n.

---

3. The `.env` File

```
# Generate with: openssl rand -hex 32
N8N_ENCRYPTION_KEY=replace-with-64-hex-characters
N8N_RUNNERS_AUTH_TOKEN=replace-with-a-long-random-string

POSTGRES_USER=n8n
POSTGRES_PASSWORD=replace-me
POSTGRES_DB=n8n

N8N_HOST=n8n.example.com
GENERIC_TIMEZONE=America/Los_Angeles
```

Nothing here is new except that these values now reach four containers instead of two. Keep the file at mode `0600` and out of version control.

---

4. The Compose File

```
services:
  postgres:
    image: postgres:18
    restart: unless-stopped
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    volumes:
      - postgres_data:/var/lib/postgresql
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER} -d ${POSTGRES_DB}"]
      interval: 10s
      timeout: 5s
      retries: 5

  redis:
    image: redis:7-alpine
    restart: unless-stopped
    command: ["redis-server", "--appendonly", "yes"]
    volumes:
      - redis_data:/data
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 10s
      timeout: 5s
      retries: 5

  n8n:
    image: n8nio/n8n:2.38.1
    restart: unless-stopped
    ports:
      - "127.0.0.1:5678:5678"
    environment:
      N8N_ENCRYPTION_KEY: ${N8N_ENCRYPTION_KEY}
      DB_TYPE: postgresdb
      DB_POSTGRESDB_HOST: postgres
      DB_POSTGRESDB_PORT: 5432
      DB_POSTGRESDB_DATABASE: ${POSTGRES_DB}
      DB_POSTGRESDB_USER: ${POSTGRES_USER}
      DB_POSTGRESDB_PASSWORD: ${POSTGRES_PASSWORD}
      EXECUTIONS_MODE: queue
      QUEUE_BULL_REDIS_HOST: redis
      QUEUE_BULL_REDIS_PORT: 6379
      OFFLOAD_MANUAL_EXECUTIONS_TO_WORKERS: "true"
      N8N_DEFAULT_BINARY_DATA_MODE: database
      N8N_HOST: ${N8N_HOST}
      N8N_PROTOCOL: https
      WEBHOOK_URL: https://${N8N_HOST}/
      N8N_EDITOR_BASE_URL: https://${N8N_HOST}/
      N8N_PROXY_HOPS: 1
      GENERIC_TIMEZONE: ${GENERIC_TIMEZONE}
      TZ: ${GENERIC_TIMEZONE}
      N8N_RUNNERS_MODE: external
      N8N_RUNNERS_BROKER_LISTEN_ADDRESS: 0.0.0.0
      N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
      N8N_NATIVE_PYTHON_RUNNER: "true"
      EXECUTIONS_DATA_PRUNE: "true"
      EXECUTIONS_DATA_MAX_AGE: 336
      N8N_GRACEFUL_SHUTDOWN_TIMEOUT: 60
      N8N_DIAGNOSTICS_ENABLED: "false"
    volumes:
      - n8n_data:/home/node/.n8n
      - n8n_files:/home/node/.n8n-files
    healthcheck:
      test: ["CMD-SHELL", "node -e
        \"require('http').get('http://127.0.0.1:5678/healthz',r=>process.exit(r.statusCode===200?
        0:1)).on('error',()=>process.exit(1))\""]
      interval: 30s
      timeout: 10s
      retries: 5
      start_period: 30s
    depends_on:
```

```
      postgres:
        condition: service_healthy
      redis:
        condition: service_healthy

  runners:
    image: n8nio/runners:2.38.1
    restart: unless-stopped
    environment:
      N8N_RUNNERS_TASK_BROKER_URI: http://n8n:5679
      N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
      N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT: 15
    depends_on:
      - n8n

  n8n-worker:
    image: n8nio/n8n:2.38.1
    restart: unless-stopped
    command: worker --concurrency=10
    environment:
      N8N_ENCRYPTION_KEY: ${N8N_ENCRYPTION_KEY}
      DB_TYPE: postgresdb
      DB_POSTGRESDB_HOST: postgres
      DB_POSTGRESDB_PORT: 5432
      DB_POSTGRESDB_DATABASE: ${POSTGRES_DB}
      DB_POSTGRESDB_USER: ${POSTGRES_USER}
      DB_POSTGRESDB_PASSWORD: ${POSTGRES_PASSWORD}
      EXECUTIONS_MODE: queue
      QUEUE_BULL_REDIS_HOST: redis
      QUEUE_BULL_REDIS_PORT: 6379
      QUEUE_HEALTH_CHECK_ACTIVE: "true"
      N8N_DEFAULT_BINARY_DATA_MODE: database
      N8N_WEBHOOK_RESPONSE_RELAY_OFFLOAD_ENABLED: "true"
      WEBHOOK_URL: https://${N8N_HOST}/
      GENERIC_TIMEZONE: ${GENERIC_TIMEZONE}
      TZ: ${GENERIC_TIMEZONE}
      N8N_RUNNERS_MODE: external
      N8N_RUNNERS_BROKER_LISTEN_ADDRESS: 0.0.0.0
      N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
      N8N_NATIVE_PYTHON_RUNNER: "true"
      N8N_GRACEFUL_SHUTDOWN_TIMEOUT: 60
      N8N_DIAGNOSTICS_ENABLED: "false"
    volumes:
      - n8n_files:/home/node/.n8n-files
    healthcheck:
      test: ["CMD-SHELL", "node -e
        \"require('http').get('http://127.0.0.1:5678/healthz/readiness',r=>process.exit(r.statusCode==
        =200?0:1)).on('error',()=>process.exit(1))\""]
      interval: 30s
      timeout: 10s
      retries: 5
      start_period: 30s
    depends_on:
      postgres:
        condition: service_healthy
      redis:
        condition: service_healthy

  worker-runners:
    image: n8nio/runners:2.38.1
    restart: unless-stopped
    environment:
      N8N_RUNNERS_TASK_BROKER_URI: http://n8n-worker:5679
      N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
      N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT: 15
    depends_on:
      - n8n-worker

volumes:
  postgres_data:
  redis_data:
  n8n_data:
  n8n_files:
```

What changed, and why

- `EXECUTIONS_MODE: queue` on main *and* every worker. A worker started without it silently runs as a regular instance and never touches the queue.

- **`QUEUE_BULL_REDIS_HOST` / `QUEUE_BULL_REDIS_PORT`** are the only Redis variables n8n reads. Anything else Redis-shaped you have picked up from an older guide is doing nothing; delete it. Add `QUEUE_BULL_REDIS_PASSWORD` if your Redis requires one.
- **`OFFLOAD_MANUAL_EXECUTIONS_TO_WORKERS: "true"`** sends the executions you start by clicking *Test workflow* to a worker too. Without it, main still runs those itself — exactly the load you moved off main.
- **`N8N_DEFAULT_BINARY_DATA_MODE: database`** on every process, for the reason in section 2.
- **No published ports on the worker.** Workers take work from Redis. Nothing needs to reach them from outside; the healthcheck runs inside the container.
- **`QUEUE_HEALTH_CHECK_ACTIVE: "true"`** turns on the worker's own HTTP server at `/healthz` and `/healthz/readiness`. Readiness is the useful one: it reports whether the worker's Postgres *and* Redis connections are live. It listens on 5678; change `QUEUE_HEALTH_CHECK_PORT` if that collides.
- **The healthcheck uses `node`, not `curl`.** The n8n image does not ship `curl`, so a `curl`-based healthcheck fails permanently and produces a restart loop that looks like an n8n problem.
- **The worker has no `/home/node/.n8n` volume.** It does not need one: the encryption key comes from the environment. Sharing one `n8n_data` volume across processes invites settings-file permission errors and two processes writing one event log.
- **`N8N_RESTRICT_FILE_ACCESS_TO` applies on every n8n container**, so main and every worker mount the same `n8n_files` volume at `/home/node/.n8n-files`, matching the reference stack.
- **No `version:` key, no `container_name`.** `version:` is obsolete in Compose v2, and `container_name` makes the scaling in section 7 impossible.

---

5. Launch and Verify

```
docker compose up -d
docker compose ps
```

Wait for Postgres and Redis to report `healthy`, then n8n and the worker. Confirm the worker joined the pool — you want a line saying it has started and is waiting for jobs, with its concurrency:

```
docker compose logs n8n-worker | grep -i "waiting for"
```

Then confirm both runner sidecars registered:

```
docker compose logs runners worker-runners | grep -i "registered\|broker"
```

If a sidecar reports a connection refused on port 5679, the n8n process it targets is missing `N8N_RUNNERS_BROKER_LISTEN_ADDRESS: 0.0.0.0`. The broker binds to localhost otherwise and nothing outside the container can reach it.

---

6. Testing the Queue

The point of this test is to prove executions land on the worker, not on main.

1. Build a workflow with a Manual Trigger, a Code node that prints something identifiable, and a Wait node set to 30 seconds.
2. Publish it with a Webhook trigger, or just run it manually — with `OFFLOAD_MANUAL_EXECUTIONS_TO_WORKERS` on, a manual run goes to the queue too.
3. Watch both sides at once:

```
docker compose logs -f n8n
```

```
docker compose logs -f n8n-worker
```

Main should log that it enqueued the job; the worker should log starting and finishing it. If main logs the execution *running*, `EXECUTIONS_MODE` is missing on main, or the offload variable is.

Then push it. Fire the webhook fifteen times and watch the worker interleave them up to its concurrency limit:

```
for i in $(seq 1 15); do
  curl -s -o /dev/null -X POST "https://${N8N_HOST}/webhook/your-path" &
done
wait
```

Finally, confirm the queue drains. With everything idle:

```
docker compose exec redis redis-cli --scan --pattern 'bull:*' | head
```

A backlog that never shrinks means you need more workers, or your workers are blocked on something — usually the Postgres connection pool.

---

7. Scaling Workers

The obvious command works:

```
docker compose up -d --scale n8n-worker=3
```

Three worker containers now pull from the same queue. But look at what you did *not* scale: `worker-runners` is still one container pointed at `http://n8n-worker:5679`. That name now round-robins across three containers, and the sidecar holds one connection to whichever it resolved first. Two of your three workers have no task runner, and every Code node they pick up fails.

Scaling the sidecar does not fix it. Compose has no concept of pairing sidecar *n* with worker *n*; three runners pointed at one service name may all land on the same worker. That pairing is a scheduler's job, which is why the Kubernetes queue-mode chapter can express it and Compose cannot.

On Compose, scale by writing the workers out explicitly. A YAML anchor keeps the duplication to the two lines that actually differ, and the shared worker environment moves into a `worker.env` file:

```
x-worker: &worker
  image: n8nio/n8n:2.38.1
  restart: unless-stopped
  command: worker --concurrency=10
  env_file: worker.env
  depends_on:
    postgres:
      condition: service_healthy
    redis:
      condition: service_healthy

x-worker-runner: &worker-runner
  image: n8nio/runners:2.38.1
  restart: unless-stopped
  env_file: runner.env

services:
  n8n-worker-1:
    <<: *worker

  worker-1-runners:
    <<: *worker-runner
    environment:
      N8N_RUNNERS_TASK_BROKER_URI: http://n8n-worker-1:5679
    depends_on:
      - n8n-worker-1

  n8n-worker-2:
    <<: *worker

  worker-2-runners:
    <<: *worker-runner
    environment:
      N8N_RUNNERS_TASK_BROKER_URI: http://n8n-worker-2:5679
    depends_on:
      - n8n-worker-2
```

`worker.env` and `runner.env` carry everything the workers and sidecars share, so the only per-service line left is the broker URI. It is more YAML than `--scale`, but each worker gets exactly one runner and you can restart one worker without disturbing the others.

`--scale` is still fine if no workflow uses a Code node, or if you accept `N8N_RUNNERS_MODE=internal` on the workers. Internal mode runs Code in a child process of the worker itself; n8n does not recommend it for production, because it drops the process isolation that makes an untrusted Code node safe to run.

On concurrency. `--concurrency=10` is the default and a reasonable start; n8n recommends 5 or higher. Do not chase throughput with many workers at low concurrency: each worker holds its own Postgres connections, and twenty workers at concurrency 2 exhaust the pool long before ten at concurrency 10 do.

---

8. Adding a Webhook Processor

Add `n8n-webhook` when webhook *ingestion* is the bottleneck — thousands of inbound requests per minute, where accepting and enqueuing the request is itself the cost. If your load is heavy workflows arriving at a modest rate, more workers help and a webhook processor does not.

```
n8n-webhook:
  image: n8nio/n8n:2.38.1
  restart: unless-stopped
  command: webhook
  env_file: worker.env
  environment:
    N8N_HOST: ${N8N_HOST}
    N8N_PROTOCOL: https
    WEBHOOK_URL: https://${N8N_HOST}/
    N8N_PROXY_HOPS: 1
  depends_on:
    postgres:
      condition: service_healthy
    redis:
      condition: service_healthy
```

It needs the same encryption key, database, Redis, and `EXECUTIONS_MODE=queue`, and it listens on 5678 like main. Your reverse proxy — see the Nginx reverse proxy chapter — routes by path: `/webhook/*` and `/webhook-waiting/*` to the processors, everything else to main. Keep `/webhook-test/*` on main; that is where manual test URLs resolve.

Do not put main in the webhook pool. It will pick up production traffic and slow the editor down, which is the problem you were solving. Once the processors are in place, set `N8N_DISABLE_PRODUCTION_MAIN_PROCESS=true` on main so it stops registering production webhooks at all.

A webhook processor does not execute workflows, so it needs no runner sidecar.

---

9. Large Webhook Responses

In queue mode a **Respond to Webhook** node runs on a worker, but the HTTP client is still connected to main. The response travels back through Redis inside a queue message, and Redis holds several copies of that message in flight.

`N8N_WEBHOOK_RESPONSE_RELAY_SIZE_MAX` caps the message at 64 MiB by default. Budget roughly 1.5 times the cap in Redis memory per in-flight response. Above the cap, the node fails.

Since n8n 2.34 you can offload instead of failing. The worker writes the body to binary data storage, the queue message carries a reference, and main streams the body to the client, then deletes it:

```
      N8N_WEBHOOK_RESPONSE_RELAY_OFFLOAD_ENABLED: "true"
      N8N_WEBHOOK_RESPONSE_RELAY_SIZE_MAX: 64
```

Set the offload variable **on workers only**, and only after every main and webhook process is on 2.34 or later — an older main returns the storage reference to the client instead of the body.

Offloading needs storage every process can read. `database` mode works and is what this stack uses, but main loads the whole body into memory before sending and the bytes pass through your primary database. If you routinely return tens of megabytes, switch `N8N_DEFAULT_BINARY_DATA_MODE` to `s3`, which streams a chunk at a time. Filesystem mode is out here for the same reason it is out everywhere in queue mode.

---

10. Graceful Shutdown and Rolling Updates

When a worker gets `SIGTERM` it stops accepting new jobs and finishes what it has. `N8N_GRACEFUL_SHUTDOWN_TIMEOUT` sets how many seconds it waits before killing the process; the default is 30. Set it above your longest normal execution — 60 in the compose file above — and give Docker room to honour it:

```
docker compose stop --timeout 90 n8n-worker
```

If Docker's stop timeout is shorter than n8n's, Docker sends `SIGKILL` first, the graceful path never runs, and half-finished executions are marked as crashed.

`QUEUE_WORKER_TIMEOUT` is the deprecated old name for this setting.
`QUEUE_WORKER_MAX_STALLED_COUNT` was removed in 2.0 and does nothing.

To update, change the pinned tag — never a floating tag here, since a partial rollout across versions is exactly what you are avoiding:

```
sed -i 's/2\.38\.1/2\.39\.0/g' compose.yml
docker compose pull
```

Then roll, workers first:

```
docker compose up -d --no-deps n8n-worker-1 worker-1-runners
docker compose logs -f --tail 20 n8n-worker-1
docker compose up -d --no-deps n8n-worker-2 worker-2-runners
docker compose up -d --no-deps n8n n8n-webhook runners
```

The queue absorbs the gap: while a worker restarts, jobs sit in Redis and the remaining workers pick them up. Main is the only service with visible downtime, and only for the few seconds it takes to restart. Read the release notes before a major version bump — some releases want main started first so it can run migrations.

---

11. Persistence and Backups

Postgres is the thing that matters, and it is covered in the VPS Docker Compose chapter. Back up the encryption key with it — a dump you cannot decrypt is not a backup.

Redis holds only in-flight queue state. With `--appendonly yes` a restart replays jobs that were enqueued but not finished, which is worth having; a backup routine is not. Lose Redis entirely and you lose the mid-flight executions and nothing else. Rebuild it empty and let the triggers fire again.

---

12. Redis High Availability

One Redis container is a single point of failure for execution dispatch. Two ways out, both a step up in cost.

Sentinel watches a primary and promotes a replica when it fails:

```
  redis-sentinel:
    image: redis:7-alpine
    restart: unless-stopped
    command: ["redis-sentinel", "/etc/redis/sentinel.conf"]
    volumes:
      - ./sentinel.conf:/etc/redis/sentinel.conf:ro
```

Run three sentinels across three hosts for a quorum that means anything. Three on one host is a rehearsal, not high availability.

Redis Cluster shards keys across nodes. Point n8n at them directly:

```
      QUEUE_BULL_REDIS_CLUSTER_NODES: redis-1:6379,redis-2:6379,redis-3:6379
```

With this set, n8n creates a cluster client and ignores `QUEUE_BULL_REDIS_HOST` and `QUEUE_BULL_REDIS_PORT`. Add `QUEUE_BULL_REDIS_PASSWORD` and `QUEUE_BULL_REDIS_TLS: "true"` if Redis is off-host. At that point a managed Redis is usually cheaper.

---

13. Troubleshooting Common Issues

Issue	Cause	Solution
Executions stay “waiting” forever	Nothing is consuming the queue	Check `docker compose logs n8n-worker` for a startup line; confirm `EXECUTIONS_MODE: queue` on the worker, not only on main
Worker starts but runs nothing	It started as a regular instance	Add `EXECUTIONS_MODE: queue` to the worker and recreate it
Credential errors on the worker only	`N8N_ENCRYPTION_KEY` differs between processes	Set the same key from `.env` on every n8n container, then recreate all of them
Code nodes fail on some workers	One sidecar shared across scaled workers	Use the per-worker layout in section 7: one sidecar each, own broker URI
Runner logs “connection refused” on 5679	Broker bound to localhost	Set `N8N_RUNNERS_BROKER_LISTEN_ADDRESS: 0.0.0.0` on the process the sidecar targets
Binary data missing or unreadable	`N8N_DEFAULT_BINARY_DATA_MODE: filesystem`	Switch every process to `database` or `s3`
“Response is too large to be sent back from the worker”	Over `N8N_WEBHOOK_RESPONSE_RELAY_SIZE_MAX`, offload off	Set `N8N_WEBHOOK_RESPONSE_RELAY_OFFLOAD_ENABLED: "true"` on the workers, or raise the limit
Restart loop, healthcheck never passes	Healthcheck uses `curl`, absent from the image	Use the `node` one-liner from section 4, or `wget --spider`
Executions marked crashed after a deploy	Docker’s stop timeout is shorter than n8n’s	Raise `--timeout` on `docker compose stop`, or lower `N8N_GRACEFUL_SHUTDOWN_TIMEOUT`
Slower as worker count grows	Postgres connection pool exhausted	Fewer workers at higher `--concurrency`; raise `max_connections`
Backlog grows and never drains	Not enough worker capacity	Add workers or raise concurrency, then re-measure

Further reading

- Enable queue mode: https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/enable-queue-mode
- Queue mode environment variables: https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/queue-mode
- Set up task runners: https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners
- Handle binary data: https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/handle-binary-data
- Install with Docker Compose: https://docs.n8n.io/deploy/host-n8n/install-options/install-using-docker-compose
- n8n 2.0 breaking changes: https://docs.n8n.io/changelog/v20-breaking-changes

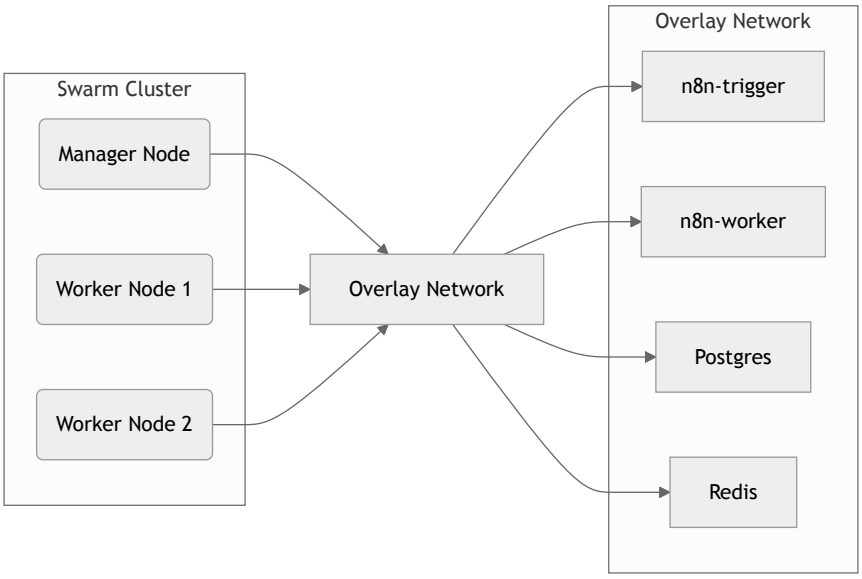
---

Chapter 16: Docker Swarm Cluster Deployment

Overview

As you move toward production-grade, highly available deployments, Docker Swarm offers built-in clustering, service discovery, and rolling updates across multiple hosts. In this chapter, you will learn to:

1. **Initialize** a Docker Swarm cluster with manager and worker nodes
2. **Create overlay networks** for inter-service communication across hosts
3. **Define a** `docker-stack.yml` manifest with deploy options (replicas, placement constraints, update configs)
4. **Use Docker secrets** for the encryption key and the database password — and understand why the runners auth token cannot be one
5. **Deploy the stack** via `docker stack deploy` and monitor services
6. **Scale services** up and down dynamically, and perform zero-downtime rolling updates
7. **Implement persistent volumes** using Docker volume plugins or NFS for shared storage
8. **Configure service constraints** (node labels, resource reservations) for workload segregation
9. **Run n8n in queue mode** with a task runners sidecar service per n8n service
10. **Back up and restore** both application state and secrets
11. **Advanced scenarios**:
    - **Multiple manager nodes** for quorum
    - Integrating **Traefik** as a Swarm ingress load balancer
    - Using **global services** for one-per-node tasks



Diagram

Prerequisites

- **At least three Linux hosts** (or VMs) with Docker Engine v20.10+ installed
- **Passwordless SSH** between manager and worker nodes for automation (optional)
- **Sufficient resources**: each node with at least 2 GB RAM, adequate CPU and disk
- **Basic understanding** of Docker networking and service architecture

1. Initializing the Swarm Cluster

1.1 On the First Manager Node

```
docker swarm init --advertise-addr MANAGER_IP
```

- **Output** will include a `docker swarm join` command for workers.

1.2 Joining Worker Nodes

On each worker:

```
docker swarm join --token SWARM_WORKER_TOKEN MANAGER_IP:2377
```

- Verify membership on the manager:

  ```
  docker node ls
  ```

- **Scenario:** For high availability, promote a second manager:

  ```
  docker node promote WORKER_ID
  ```

---

2. Creating an Overlay Network

On any manager:

```
docker network create -d overlay n8n-net
```

- **Use case:** Allows containers on different nodes to communicate via DNS (`postgres` , `redis` , `n8n-trigger`).

---

3. Managing Secrets

`N8N_ENCRYPTION_KEY` encrypts every stored credential; the database password protects your Postgres instance; the runners auth token authenticates the task runner sidecars to n8n's broker. n8n has no basic auth to manage — the owner account is created on first login. Store the encryption key and the database password as Swarm secrets:

```
openssl rand -hex 32 | docker secret create n8n_encryption_key -
echo -n "SuperSecretPg" | docker secret create postgres_password -
```

- List secrets:

  ```
  docker secret ls
  ```

- **Back up the encryption key alongside the database.** If you lose it, a restored database's credentials are unreadable and every workflow using them stops working.
- **Scenario:** Use an external secret management tool (Vault) and integrate via `docker config`.

The runners auth token cannot be a Swarm secret. Swarm delivers a secret as a file under `/run/secrets/`, and the only way a container turns that file into a variable is the `_FILE` convention — which the `n8nio/runners` image does not implement. Swarm has no mechanism for injecting a secret directly into a service's environment, so the token has to travel as a plain environment value. Generate it once and export it in the shell you deploy from; `docker stack deploy` substitutes `${N8N_RUNNERS_AUTH_TOKEN}` at deploy time, so the value stays out of the stack file and out of version control:

```
export N8N_RUNNERS_AUTH_TOKEN="$(openssl rand -hex 32)"
```

What you give up: the value is readable in `docker service inspect` and in `/proc` on the node running the task, for anyone who can reach the Docker socket. Two things limit the damage. Put the runners and their brokers on their own overlay network so the broker port is not reachable from elsewhere in the cluster, and use a token that authorises nothing but broker connections — never reuse the encryption key, the database password, or an API key here.

---

4. Defining `docker-stack.yml`

Docker secrets mount at `/run/secrets/<name>`. The `n8nio/n8n` image reads the `_FILE`-suffixed variant of a variable (`N8N_ENCRYPTION_KEY_FILE`, `DB_POSTGRESDB_PASSWORD_FILE`) and loads the secret from disk at startup, so those values never show up in `docker service inspect`. The `n8nio/runners` image does not: it ignores `_FILE` variables entirely, so `N8N_RUNNERS_AUTH_TOKEN` is passed as a plain value on both the n8n services and the runner services, as explained in section 3.

Create `docker-stack.yml`:

```yaml
services:
  postgres:
    image: postgres:18
    environment:
      POSTGRES_USER: n8n
      POSTGRES_DB: n8n
      POSTGRES_PASSWORD_FILE: /run/secrets/postgres_password
    secrets:
      - postgres_password
    volumes:
      - postgres_data:/var/lib/postgresql
    networks:
      - n8n-net
    deploy:
      replicas: 1
      placement:
        constraints:
          - node.role == manager
      update_config:
        parallelism: 1
        delay: 10s
      restart_policy:
        condition: on-failure
      resources:
        limits:
          memory: 512M
        reservations:
          memory: 256M
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U n8n -d n8n"]
      interval: 30s
      timeout: 10s
      retries: 5

  redis:
    image: redis:7-alpine
    command: ["redis-server", "--appendonly", "yes"]
    volumes:
      - redis_data:/data
    networks:
      - n8n-net
    deploy:
      replicas: 1
      placement:
        constraints:
          - node.role == manager
      restart_policy:
        condition: on-failure
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 10s
      timeout: 5s
      retries: 5

  n8n-trigger:
    image: n8nio/n8n:2.38.1
    ports:
      - target: 5678
        published: 80
        protocol: tcp
        mode: host
    environment:
```

```
      DB_TYPE: postgresdb
      DB_POSTGRESDB_HOST: postgres
      DB_POSTGRESDB_PORT: 5432
      DB_POSTGRESDB_DATABASE: n8n
      DB_POSTGRESDB_USER: n8n
      DB_POSTGRESDB_PASSWORD_FILE: /run/secrets/postgres_password
      N8N_ENCRYPTION_KEY_FILE: /run/secrets/n8n_encryption_key
      EXECUTIONS_MODE: queue
      QUEUE_BULL_REDIS_HOST: redis
      QUEUE_BULL_REDIS_PORT: 6379
      OFFLOAD_MANUAL_EXECUTIONS_TO_WORKERS: "true"
      N8N_DEFAULT_BINARY_DATA_MODE: database
      N8N_HOST: n8n.example.com
      N8N_PROTOCOL: https
      WEBHOOK_URL: https://n8n.example.com/
      N8N_EDITOR_BASE_URL: https://n8n.example.com/
      N8N_PROXY_HOPS: 1
      N8N_RUNNERS_MODE: external
      N8N_RUNNERS_BROKER_LISTEN_ADDRESS: 0.0.0.0
      N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
      N8N_NATIVE_PYTHON_RUNNER: "true"
      N8N_GRACEFUL_SHUTDOWN_TIMEOUT: 60
    secrets:
      - postgres_password
      - n8n_encryption_key
    volumes:
      - n8n_data:/home/node/.n8n
    networks:
      - n8n-net
    stop_grace_period: 90s
    deploy:
      mode: replicated
      replicas: 2
      placement:
        max_replicas_per_node: 1
      update_config:
        parallelism: 1
        delay: 15s
      restart_policy:
        condition: on-failure
      resources:
        limits:
          cpus: "0.50"
          memory: 512M
        reservations:
          cpus: "0.25"
          memory: 256M
    healthcheck:
      test: ["CMD-SHELL", "node -e
        \"require('http').get('http://localhost:5678/healthz',r=>process.exit(r.statusCode===200?
        0:1))\""]
      interval: 30s
      timeout: 10s
      retries: 5

  n8n-trigger-runners:
    image: n8nio/runners:2.38.1
    environment:
      N8N_RUNNERS_TASK_BROKER_URI: http://n8n-trigger:5679
      N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
      N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT: 15
    networks:
      - n8n-net
    deploy:
      mode: replicated
      replicas: 2
      endpoint_mode: dnsrr
      restart_policy:
        condition: on-failure

  n8n-worker:
    image: n8nio/n8n:2.38.1
    command: ["n8n", "worker", "--concurrency=10"]
    environment:
      DB_TYPE: postgresdb
      DB_POSTGRESDB_HOST: postgres
      DB_POSTGRESDB_PORT: 5432
      DB_POSTGRESDB_DATABASE: n8n
      DB_POSTGRESDB_USER: n8n
      DB_POSTGRESDB_PASSWORD_FILE: /run/secrets/postgres_password
```

```
      N8N_ENCRYPTION_KEY_FILE: /run/secrets/n8n_encryption_key
      EXECUTIONS_MODE: queue
      QUEUE_BULL_REDIS_HOST: redis
      QUEUE_BULL_REDIS_PORT: 6379
      N8N_DEFAULT_BINARY_DATA_MODE: database
      N8N_HOST: n8n.example.com
      N8N_PROTOCOL: https
      WEBHOOK_URL: https://n8n.example.com/
      N8N_EDITOR_BASE_URL: https://n8n.example.com/
      N8N_PROXY_HOPS: 1
      N8N_RUNNERS_MODE: external
      N8N_RUNNERS_BROKER_LISTEN_ADDRESS: 0.0.0.0
      N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
      N8N_NATIVE_PYTHON_RUNNER: "true"
      N8N_GRACEFUL_SHUTDOWN_TIMEOUT: 60
    secrets:
      - postgres_password
      - n8n_encryption_key
    volumes:
      - n8n_data:/home/node/.n8n
    networks:
      - n8n-net
    stop_grace_period: 90s
    deploy:
      mode: replicated
      replicas: 3
      update_config:
        parallelism: 1
        delay: 10s
      restart_policy:
        condition: on-failure
      resources:
        limits:
          cpus: "0.50"
          memory: 512M
        reservations:
          cpus: "0.25"
          memory: 256M

  n8n-worker-runners:
    image: n8nio/runners:2.38.1
    environment:
      N8N_RUNNERS_TASK_BROKER_URI: http://n8n-worker:5679
      N8N_RUNNERS_AUTH_TOKEN: ${N8N_RUNNERS_AUTH_TOKEN}
      N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT: 15
    networks:
      - n8n-net
    deploy:
      mode: replicated
      replicas: 3
      endpoint_mode: dnsrr
      restart_policy:
        condition: on-failure

volumes:
  postgres_data:
    driver: local
  redis_data:
    driver: local
  n8n_data:
    driver: local

networks:
  n8n-net:
    external: true

secrets:
  postgres_password:
    external: true
  n8n_encryption_key:
    external: true
```

Why the healthcheck runs `node`. The `n8nio/n8n` image ships no `curl`. A `curl -f http://localhost:5678/healthz` test exits 127 on every run, Swarm marks the task unhealthy, and the service restarts forever without ever telling you why. Node is in the image by definition, so the probe above uses it directly. `wget -qO- http://localhost:5678/healthz` works too.

Why one runners service per n8n service. Every process that runs Code nodes needs a task runner attached to its own broker: the trigger service runs manual executions, the workers run everything else. That is why there are two runner services rather than one shared pool.

What Swarm cannot do here. Kubernetes puts a runners sidecar in the same pod as n8n, reachable at `localhost:5679`, so a runner is bound to exactly one broker for its lifetime. Swarm has no pod equivalent and no way to pin task *n* of one service to task *n* of another. A runner dialling `http://n8n-worker:5679` resolves the service name and connects to whichever task it lands on. Two mitigations keep this workable:

- `endpoint_mode: dnsrr` on both runner services turns off the Swarm virtual IP so each runner resolves a real task address and connects directly, rather than through a load balancer that may move the connection.
- Set `replicas` on a runners service equal to `replicas` on the n8n service it serves, and change them together. With equal counts and round-robin DNS, each broker ends up with roughly one runner. Roughly is the operative word: nothing enforces it, and a restart can leave one broker with two runners and another with none until the next reconnect.

If you need strict one-runner-per-broker pairing, use Kubernetes instead, where the sidecar model gives it to you by construction. That is the subject of **Kubernetes: Scaled Queue Mode**.

Why `max_replicas_per_node: 1` on `n8n-trigger`. The service publishes port 80 in `mode: host`, which binds the port on the node itself instead of routing through the Swarm ingress mesh. Host mode preserves the client's source address, which matters when you are counting on `N8N_PROXY_HOPS`, but two replicas scheduled onto the same node both try to bind port 80 and the second one fails to start. The placement constraint forces Swarm to spread the replicas, so you need at least as many eligible nodes as replicas. If you would rather not manage that, drop `mode: host` and let the port publish through the ingress mesh; you then lose the real client IP unless the proxy in front of the cluster supplies it.

---

5. Deploying the Stack

On a manager node, with `N8N_RUNNERS_AUTH_TOKEN` exported in the shell so the stack file's `${N8N_RUNNERS_AUTH_TOKEN}` resolves:

```
export N8N_RUNNERS_AUTH_TOKEN="<the token you generated in section 3>"
docker stack deploy --compose-file docker-stack.yml n8n-stack
```

- **`n8n-stack`** is the name of your stack; commands like `docker stack ps n8n-stack` operate on it.
- If the variable is not set, Swarm substitutes an empty string and every runner fails broker authentication. Export it again in any shell you redeploy from.

5.1 Verifying Deployment

```
docker stack ls
docker stack services n8n-stack
docker service ls
```

- Check service replicas, update status, and whether they are running.

---

6. Scaling and Rolling Updates

6.1 Scaling

To adjust the number of trigger or worker replicas, scale the runner service to match:

```
docker service scale n8n-stack_n8n-worker=5 n8n-stack_n8n-worker-runners=5
docker service scale n8n-stack_n8n-trigger=3 n8n-stack_n8n-trigger-runners=3
```

6.2 Rolling Update

To move to a newer pinned patch release:

1. **Modify** `docker-stack.yml`: change the image tag on `n8n-trigger`, `n8n-worker`, `n8n-trigger-runners`, and `n8n-worker-runners` together — the runner and n8n images must stay on the same version.
2. **Deploy** again:

   ```
   docker stack deploy --compose-file docker-stack.yml n8n-stack
   ```

- Swarm performs a rolling update according to `update_config`. Never deploy with an unpinned or `latest` tag — a manager restart or node failure could reschedule a task and silently pull a different version.

---

7. Persistent Volumes Across Nodes

Using the default **local** volume driver attaches data to the node where the container runs. For data persistence across nodes:

7.1 NFS Volume Plugin

1. **Install** the NFS plugin on each node:

```
docker plugin install vieux/sshfs
# or use an NFS-specific plugin: docker plugin install oliver006/volume-nfs
```

2. **Define** the volume in `docker-stack.yml`:

```
volumes:
  n8n_data:
    driver: oliver006/volume-nfs
    driver_opts:
      share: "nfs-server:/path/to/n8n-data"
      mount_opts: "vers=4,rw"
```

- **Benefit:** All nodes see the same data directory.
- Mount it read-write. n8n writes its settings file and its own working files into `/home/node/.n8n` on every start; a read-only mount stops the container before it finishes booting.

---

8. Health Monitoring & Maintenance

- **Service Logs**:

```
docker service logs n8n-stack_n8n-trigger --follow
```

- **Node Health**:

```
docker node ls
docker node inspect <NODE_ID> --format '{{.Status}}'
```

- **Remove Failed Nodes**:

```
docker node demote <NODE_ID>
docker node rm <NODE_ID>
```

---

9. Backups & Restores

9.1 Secrets

Secrets are stored in Swarm's encrypted Raft log, but Raft is not a backup — re-create the same secret values on a fresh cluster from your own password manager or vault, using the same names:

```
docker secret inspect n8n_encryption_key
docker secret inspect postgres_password
```

`docker secret inspect` never returns the secret value itself, so keep the encryption key and database password recorded outside the cluster before you need them. Record the runners auth token in the same place — it is not a Swarm secret at all, so nothing in the cluster holds a copy you can recover.

9.2 Volume Backups

Follow the backup commands in **Docker Compose in Queue Mode: Main, Workers, Webhooks, and Runners** on the host that holds the volume data.

---

10. Advanced Scenarios

10.1 Traefik Integration as Ingress

Deploy Traefik as a global service:

```
  traefik:
    image: traefik:v3.3.6
    command:
      - "--providers.docker=true"
      - "--entrypoints.web.address=:80"
      - "--entrypoints.websecure.address=:443"
      - "--api.insecure=true"
    ports:
      - target: 80
        published: 80
        mode: host
      - target: 443
        published: 443
        mode: host
      - target: 8080
        published: 8080
        mode: host
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro
    networks:
      - n8n-net
    deploy:
      mode: global
      placement:
        constraints:
          - node.role == manager
```

Add labels to `n8n-trigger`, and drop its `ports:` block in favor of routing through Traefik:

```
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.n8n.rule=Host(`n8n.example.com`)"
      - "traefik.http.services.n8n.loadbalancer.server.port=5678"
```

10.2 Manager Quorum

Always have an **odd number** of manager nodes (3 or 5) to maintain quorum:

```
docker node promote worker-node3
```

---

11. Troubleshooting Common Issues

Issue	Cause	Solution
`docker stack deploy` fails on secret error	Secret not created or `external` flag mismatched	Re-create the secret without `external: true`, or create it before deploying
Service stuck in “Pending”	Node constraint mismatch or insufficient resources	Check `docker service inspect` for placement issues, update constraints, or add more nodes
Overlay network unreachable	Firewall blocking VXLAN ports (2377, 7946, 4789)	Open UDP 4789 (VXLAN), TCP/UDP 7946 (control), TCP 2377 (manager)
Data volume not accessible on other nodes	Using the local driver rather than a shared volume	Switch to NFS or another distributed volume plugin
Rolling update fails	Update parallelism too high or healthchecks failing	Lower parallelism, increase delay, fix underlying service errors
Runner never picks up Code node executions	Auth token mismatch between n8n and its runner service	Confirm `N8N_RUNNERS_AUTH_TOKEN` holds the same value on both services and that both run the same image tag
Runner container exits with an auth error after you set `N8N_RUNNERS_AUTH_TOKEN_FILE`	The `n8nio/runners` image ignores `_FILE` variables	Pass `N8N_RUNNERS_AUTH_TOKEN` as a plain value; only the `n8nio/n8n` image reads `_FILE`
`n8n-trigger` replica stuck starting, port already in use	Two `mode: host` replicas landed on the same node	Keep `max_replicas_per_node: 1`, or publish the port through the ingress mesh
n8n container unhealthy immediately, exit code 127 in the healthcheck	The healthcheck calls `curl`, which the n8n image does not ship	Use the `node -e` probe shown in section 4
Container starts then stops with a settings-file write error	Shared volume mounted read-only	Set `mount_opts: "vers=4,rw"` on the NFS volume

---

Further reading

- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/enable-queue-mode
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/queue-mode
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/handle-binary-data

---

Chapter 17: Kubernetes: Single Pod

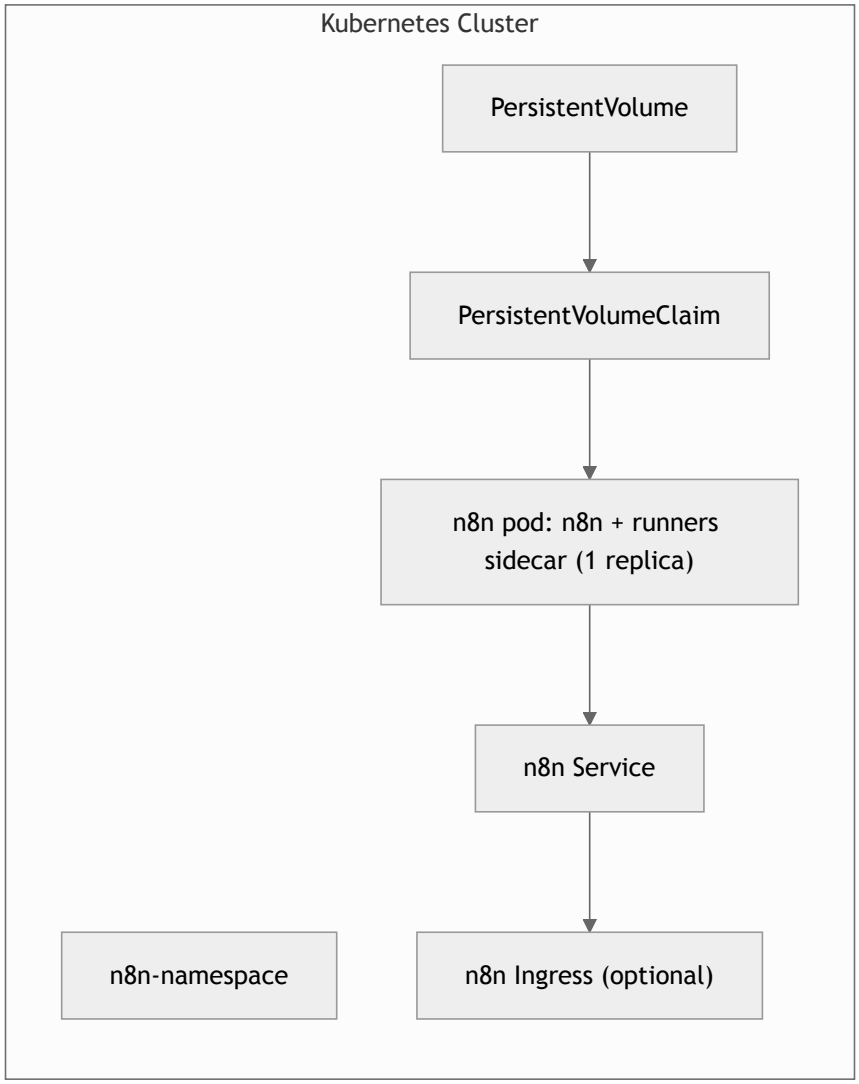
Overview

Deploying n8n on Kubernetes in a single-pod configuration provides you with:

- **Basic orchestration**: simplified rollout, self-healing, and standardized deployment
- **Declarative management**: versioned YAML manifests for reproducible environments
- **Built-in health checks**: readiness and liveness probes to ensure pod reliability
- **Persistent storage**: workflow data on a PersistentVolumeClaim that survives pod restarts and rescheduling

In this chapter you will learn to:

1. **Set up a Kubernetes namespace** for logical isolation
2. **Create Kubernetes Secrets** for credentials (the encryption key, the runners auth token, and DB passwords)
3. **Define a PersistentVolume (PV) and PersistentVolumeClaim (PVC)** for n8n data
4. **Write a Deployment** manifest for a single n8n pod with a task runners sidecar, resource limits, probes, and environment variables
5. **Expose the pod** via a Service (ClusterIP or LoadBalancer)
6. **(Optional) Configure an Ingress** with TLS termination
7. **Apply** the manifests and **verify** pod health and Service endpoints
8. **Monitor logs** and **describe** resources
9. **Perform updates** by applying new Deployment versions
10. **Troubleshoot** common pod failures (CrashLoopBackOff, PV binding issues)



Diagram

Prerequisites

- A **Kubernetes cluster** (minikube, kind, EKS, GKE, AKS, or self-managed)
- **kubectl** configured to access the cluster
- **StorageClass** available for dynamic provisioning (e.g., standard, gp2, rook-ceph)
- Basic familiarity with **Kubernetes resources** (Namespaces, Pods, Deployments, Services, Ingress)

---

1. Create a Namespace

Namespaces isolate resources:

```
kubectl create namespace n8n
```

Verify:

```
kubectl get namespaces
```

---

2. Create Secrets

n8n has no basic auth — the first visit to the editor creates the owner account. What you do need to protect are the encryption key (which decrypts every stored credential), the runners auth token (which authenticates the task runner sidecar to n8n's broker), and, if you use Postgres, the database password:

```
kubectl -n n8n create secret generic n8n-auth \
  --from-literal=N8N_ENCRYPTION_KEY="$(openssl rand -hex 32)" \
  --from-literal=N8N_RUNNERS_AUTH_TOKEN="$(openssl rand -hex 32)"

# If using Postgres:
kubectl -n n8n create secret generic n8n-db-credentials \
  --from-literal=DB_USER=n8n \
  --from-literal=DB_PASSWORD=SuperSecretPg
```

List secrets:

```
kubectl -n n8n get secrets
```

- **Back up `N8N_ENCRYPTION_KEY`.** Read it out with `kubectl -n n8n get secret n8n-auth -o jsonpath='{.data.N8N_ENCRYPTION_KEY}' | base64 -d` and store it with your database backups — losing it makes every stored credential unreadable.

---

3. Define Persistent Volume & Claim

3.1 PersistentVolumeClaim (Dynamic Provisioning)

```yaml
# n8n-pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: n8n-pvc
  namespace: n8n
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  storageClassName: standard   # adjust to your cluster's StorageClass
```

Apply:

```
kubectl apply -f n8n-pvc.yaml
```

Verify:

```
kubectl -n n8n get pvc
```

> **Scenario:** For shared access (ReadWriteMany), use an NFS or CSI-based StorageClass.

---

4. Create the Deployment

This chapter uses the default SQLite database, so the Deployment stays at a single replica — SQLite assumes one writer, and a second pod sharing the PVC would corrupt the database file. Changing the database does not change that count. On Community Edition a deployment runs exactly one main pod no matter which database is behind it; the way to add capacity is queue mode with worker pods, which is the subject of **Kubernetes: Scaled Queue Mode**.

The `strategy: Recreate` setting matters for the same reason. The default rolling update starts the new pod before stopping the old one, and two pods would then contend for the same `ReadWriteOnce` volume and the same SQLite file. `Recreate` stops the old pod first, at the cost of a short gap during every update.

The pod runs two containers: `n8n` and a `runners` sidecar that executes Code node tasks out of process. They share the pod's network namespace, so the sidecar reaches n8n's broker at `localhost:5679` with no separate Service needed.

Define `n8n-deployment.yaml`:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: n8n
  namespace: n8n
spec:
  replicas: 1   # exactly one main instance — see the note above
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: n8n
  template:
    metadata:
      labels:
        app: n8n
    spec:
      containers:
        - name: n8n
          image: n8nio/n8n:2.38.1
          ports:
            - containerPort: 5678
          env:
            - name: DB_TYPE
              value: sqlite   # or 'postgresdb'
            # For Postgres use secrets:
            # - name: DB_TYPE
            #   value: postgresdb
            # - name: DB_POSTGRESDB_HOST
            #   value: your-postgres-host
            # - name: DB_POSTGRESDB_USER
            #   valueFrom:
            #     secretKeyRef:
            #       name: n8n-db-credentials
            #       key: DB_USER
            # - name: DB_POSTGRESDB_PASSWORD
            #   valueFrom:
            #     secretKeyRef:
            #       name: n8n-db-credentials
            #       key: DB_PASSWORD
            - name: N8N_ENCRYPTION_KEY
              valueFrom:
                secretKeyRef:
                  name: n8n-auth
                  key: N8N_ENCRYPTION_KEY
            - name: N8N_HOST
              value: n8n.example.com
            - name: N8N_PROTOCOL
              value: https
            - name: WEBHOOK_URL
```

```
            value: https://n8n.example.com/
          - name: N8N_EDITOR_BASE_URL
            value: https://n8n.example.com/
          - name: N8N_PROXY_HOPS
            value: "1"
          - name: N8N_RUNNERS_MODE
            value: external
          - name: N8N_RUNNERS_BROKER_LISTEN_ADDRESS
            value: 0.0.0.0
          - name: N8N_RUNNERS_AUTH_TOKEN
            valueFrom:
              secretKeyRef:
                name: n8n-auth
                key: N8N_RUNNERS_AUTH_TOKEN
          - name: N8N_NATIVE_PYTHON_RUNNER
            value: "true"
          - name: N8N_DEFAULT_BINARY_DATA_MODE
            value: filesystem
        volumeMounts:
          - name: n8n-data
            mountPath: /home/node/.n8n
        resources:
          requests:
            memory: "512Mi"
            cpu: "250m"
          limits:
            memory: "1Gi"
            cpu: "500m"
        livenessProbe:
          httpGet:
            path: /healthz
            port: 5678
          initialDelaySeconds: 60
          periodSeconds: 30
          failureThreshold: 3
        readinessProbe:
          httpGet:
            path: /healthz
            port: 5678
          initialDelaySeconds: 30
          periodSeconds: 15
      - name: runners
        image: n8nio/runners:2.38.1
        env:
          - name: N8N_RUNNERS_TASK_BROKER_URI
            value: http://localhost:5679
          - name: N8N_RUNNERS_AUTH_TOKEN
            valueFrom:
              secretKeyRef:
                name: n8n-auth
                key: N8N_RUNNERS_AUTH_TOKEN
          - name: N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT
            value: "15"
        resources:
          requests:
            memory: "256Mi"
            cpu: "100m"
          limits:
            memory: "512Mi"
            cpu: "250m"
    volumes:
      - name: n8n-data
        persistentVolumeClaim:
          claimName: n8n-pvc
```

Apply:

```
kubectl apply -f n8n-deployment.yaml
```

Verify pods:

```
kubectl -n n8n get pods -l app=n8n
```

Both containers must show `2/2` under `READY` before the pod is fully up.

---

5. Expose via Service

Create `n8n-service.yaml`:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: n8n-service
  namespace: n8n
spec:
  type: LoadBalancer   # or ClusterIP for internal
  selector:
    app: n8n
  ports:
    - port: 80
      targetPort: 5678
      protocol: TCP
```

Apply:

```
kubectl apply -f n8n-service.yaml
```

Check service:

```
kubectl -n n8n get svc
```

- For **type: LoadBalancer**, note the external IP assigned.

> **Scenario:** On bare-metal clusters without LoadBalancer support, use **NodePort** or an Ingress controller.

---

6. (Optional) Configure Ingress with TLS

Assuming an Ingress controller (e.g., NGINX Ingress) is installed, add the annotations n8n's websocket-based editor needs — a long-lived connection that a default proxy timeout would otherwise cut:

```yaml
# n8n-ingress.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: n8n-ingress
  namespace: n8n
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-prod   # if using cert-manager
    nginx.ingress.kubernetes.io/proxy-read-timeout: "3600"
    nginx.ingress.kubernetes.io/proxy-send-timeout: "3600"
    nginx.ingress.kubernetes.io/proxy-body-size: "50m"
spec:
  rules:
    - host: n8n.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: n8n-service
                port:
                  number: 80
  tls:
    - hosts:
        - n8n.example.com
      secretName: n8n-tls
```

Apply:

```
kubectl apply -f n8n-ingress.yaml
```

- **Cert-Manager** will provision a certificate via Let's Encrypt into `n8n-tls`. See Kubernetes: Ingress and Let's Encrypt for the full Cert-Manager setup.

---

7. Monitoring and Logs

7.1 View Pod Logs

```
kubectl -n n8n logs deployment/n8n -c n8n
kubectl -n n8n logs deployment/n8n -c runners
```

7.2 Describe Resources

```
kubectl -n n8n describe pod <pod-name>
kubectl -n n8n describe svc n8n-service
kubectl -n n8n describe pvc n8n-pvc
```

7.3 Metrics and Dashboards

Integrate with **Prometheus** and **Grafana** using community Helm charts for metrics on container CPU/memory and custom n8n endpoints (`N8N_METRICS=true` exposes `/metrics`).

---

8. Performing Updates

8.1 Change Image Version

Edit `n8n-deployment.yaml`, bumping both the `n8n` and `runners` images together — they must stay on matching versions:

```
    image: n8nio/n8n:2.39.0      # next pinned release
    # ...
    image: n8nio/runners:2.39.0  # keep in lockstep with n8n
```

Reapply:

```
kubectl apply -f n8n-deployment.yaml
```

- With `strategy: Recreate` the old pod stops before the new one starts, so expect a short outage during the swap. Never set the tag to `latest` — an unpinned image can change under you on any pod reschedule.

8.2 Scaling

Leave `replicas` at 1. A second main pod is not a scaling option here, for two reasons that both hold regardless of the database:

- Each main instance runs the scheduler, poll triggers, and webhook registrations on its own. Two mains on the same database fire the same cron and poll triggers twice. The supported way to run more than one main is queue mode with workers, and coordinating multiple mains needs `N8N_MULTI_MAIN_SETUP_ENABLED`, which is an Enterprise feature. Community Edition therefore runs exactly one main pod — `replicas: 1`, `strategy: Recreate` — plus as many worker pods as you need.
- On SQLite specifically, a second pod would also open the same database file through the same `ReadWriteOnce` volume and corrupt it.

To add execution capacity, convert to queue mode and add worker pods. **Kubernetes: Scaled Queue Mode** covers the Redis StatefulSet, the worker Deployment, and the runners sidecar each worker needs.

---

9. Troubleshooting Common Issues

Issue	Cause	Solution
Pod in CrashLoopBackOff	Missing environment variable or volume mount issue	Inspect pod logs with `kubectl logs -c n8n`, verify env and PVC binding
PVC stays “Pending”	No matching PersistentVolume or StorageClass	Ensure a dynamic StorageClass exists or create a static PV
Service external IP not assigned	Cloud provider LoadBalancer not supported	Use a **NodePort** Service or configure MetalLB for bare-metal
Readiness probe never passes	`/healthz` endpoint not available	Check that the n8n health endpoint is correct, increase `initialDelaySeconds`
Ingress returns 404	Service name or port mismatch	Verify the Ingress backend service name/port and that the Service is `ClusterIP`
TLS certificate not provisioned	Cert-Manager misconfigured or annotation missing	Check Cert-Manager logs and Ingress annotations, ensure a ClusterIssuer exists
Code nodes hang or fail	Runner container not ready, or auth token mismatch	Check `kubectl logs -c runners`; confirm both containers read the same `N8N_RUNNERS_AUTH_TOKEN`
SQLite database is locked or corrupt	Replicas raised above 1	Scale back to 1 replica and restore from backup; add capacity with queue mode workers, not more mains
Duplicate scheduled or polling executions	More than one main pod running	Return to `replicas: 1` and move execution load to worker pods in queue mode
Pod stuck `Pending` on update, volume in use	Rolling update with a `ReadWriteOnce` PVC	Set `strategy: type: Recreate` so the old pod releases the volume first

---

Further reading

- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/enable-queue-mode
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/handle-binary-data
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/deployment

---

Chapter 18: Kubernetes: External Database

Overview

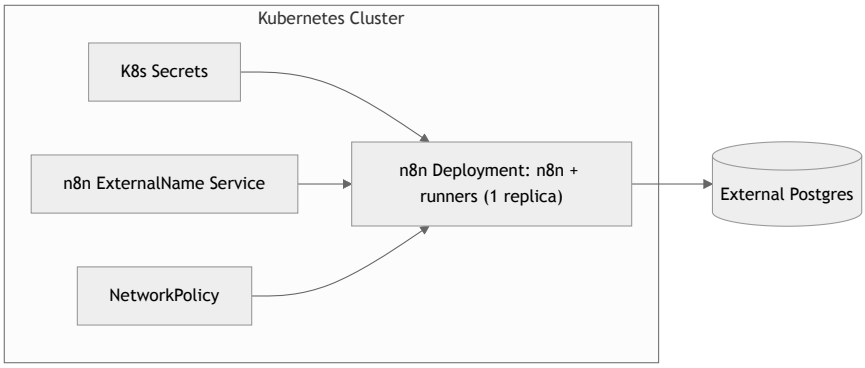
Decoupling your database from your Kubernetes application pods is a best practice in cloud-native deployments. Using an **external PostgreSQL** instance (managed service or standalone server) provides:

- **Separation of concerns**: database lifecycle and storage managed independently of the pods
- **Availability**: the provider's failover, point-in-time recovery, and managed backups instead of your own
- **Sizing**: grow database CPU, memory, and disk without touching the cluster
- **Security**: network isolation, VPC peering, IAM authentication

An external database does not raise the replica count. That is worth saying up front, because it is the usual expectation. The main pod stays at one; capacity comes from queue mode workers, covered in **Kubernetes: Scaled Queue Mode**.

In this chapter you will learn to:

1. **Provision or identify** an external PostgreSQL instance (Cloud SQL, RDS, Azure Database, self-hosted VM)
2. **Configure Kubernetes Secrets** for database credentials and SSL certificates
3. **Define a Kubernetes Service of type ExternalName** to reference the external host
4. **Update your Deployment manifest** to consume the external database over TLS, keeping the single main pod and its task runners sidecar
5. **Set up NetworkPolicies** (optional) to restrict pod egress to only the DB host
6. **Perform rolling updates** of your n8n pods with no downtime
7. **Monitor connectivity** and troubleshoot common connection issues
8. **Advanced scenarios**: IAM-based authentication (Cloud SQL Proxy), TLS enforcement, read replicas



Diagram

Prerequisites

- A **running Kubernetes cluster** with `kubectl` access
- The `n8n-auth` Secret and `n8n-pvc` PersistentVolumeClaim from Kubernetes: Single Pod
- **External PostgreSQL** endpoint and credentials (host, port, database, user, password)
- Optional: TLS certificates (CA, client cert/key) for mTLS or SSL
- Familiarity with **Secrets**, **Services**, and **NetworkPolicy** in Kubernetes

1. Provision or Identify Your External PostgreSQL

Examples:

- **Google Cloud SQL** PostgreSQL
- **AWS RDS** PostgreSQL
- **Azure Database for PostgreSQL**
- **Self-hosted VM** running Postgres in the same VPC

Ensure the database:

- Is reachable from your cluster nodes (via VPC, peered network, VPN)
- Has a database created (e.g., `n8n`) and a user with appropriate privileges

---

2. Create Kubernetes Secrets for DB Credentials and TLS

Store connection details securely:

```
kubectl -n n8n create secret generic n8n-db-credentials \
  --from-literal=DB_HOST=external-db.example.com \
  --from-literal=DB_PORT=5432 \
  --from-literal=DB_NAME=n8n \
  --from-literal=DB_USER=n8nuser \
  --from-literal=DB_PASSWORD=SuperSecretExternal
```

If the database requires TLS, put the CA certificate in a generic Secret. Use `--from-file=ca.crt=ca.crt` rather than `kubectl create secret tls`: a TLS Secret always stores its contents under the keys `tls.crt` and `tls.key`, and the file n8n is told to read is `/etc/secrets/ca.crt`. A generic Secret lets you choose the key, and the key becomes the file name when the Secret is mounted as a volume.

```
kubectl -n n8n create secret generic n8n-db-ca \
  --from-file=ca.crt=ca.crt
```

- List secrets:

  ```
  kubectl -n n8n get secrets
  ```

This chapter reuses the `n8n-auth` Secret (`N8N_ENCRYPTION_KEY`, `N8N_RUNNERS_AUTH_TOKEN`) created in Kubernetes: Single Pod. n8n still has no basic auth to configure.

---

3. Define an ExternalName Service

Enable your pods to address the external DB via an internal DNS name:

```yaml
# n8n-external-db-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: n8n-external-db
  namespace: n8n
spec:
  type: ExternalName
  externalName: external-db.example.com   # your DB host
  ports:
    - port: 5432
```

Apply:

```
kubectl apply -f n8n-external-db-service.yaml
```

- Verify:

  ```
  kubectl -n n8n get svc n8n-external-db
  ```

---

4. Update n8n Deployment to Use External DB

Modify your `n8n-deployment.yaml` to consume the secrets and the ExternalName service, add the URL variables so n8n generates correct webhook and editor links behind your load balancer, and keep the runners sidecar alongside the n8n container.

The replica count stays at 1 and the strategy stays `Recreate`. Moving from SQLite to Postgres removes the single-writer file, but it does not remove the reason for one main instance: every main runs the scheduler, the poll triggers, and the webhook registrations for itself, so two mains on one database fire the same cron and poll triggers twice. Running more than one main is a queue-mode arrangement that also needs `N8N_MULTI_MAIN_SETUP_ENABLED`, an Enterprise feature. On Community Edition you run one main pod and add worker pods.

There is a storage reason as well. The `n8n-pvc` claim from **Kubernetes: Single Pod** is `ReadWriteOnce`, which one node can mount at a time. Two pods scheduled onto different nodes leave the second one stuck with a multi-attach error, and `Recreate` avoids the same collision during an update.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: n8n
  namespace: n8n
spec:
  replicas: 1
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: n8n
  template:
    metadata:
      labels:
        app: n8n
    spec:
      containers:
        - name: n8n
          image: n8nio/n8n:2.38.1
          ports:
            - containerPort: 5678
          env:
            - name: DB_TYPE
              value: postgresdb
            - name: DB_POSTGRESDB_HOST
              value: n8n-external-db
            - name: DB_POSTGRESDB_PORT
              valueFrom:
                secretKeyRef:
                  name: n8n-db-credentials
                  key: DB_PORT
            - name: DB_POSTGRESDB_DATABASE
              valueFrom:
                secretKeyRef:
                  name: n8n-db-credentials
                  key: DB_NAME
            - name: DB_POSTGRESDB_USER
              valueFrom:
                secretKeyRef:
                  name: n8n-db-credentials
                  key: DB_USER
            - name: DB_POSTGRESDB_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: n8n-db-credentials
                  key: DB_PASSWORD
            - name: N8N_ENCRYPTION_KEY
              valueFrom:
                secretKeyRef:
                  name: n8n-auth
```

```
                key: N8N_ENCRYPTION_KEY
          - name: N8N_HOST
            value: n8n.example.com
          - name: N8N_PROTOCOL
            value: https
          - name: WEBHOOK_URL
            value: https://n8n.example.com/
          - name: N8N_EDITOR_BASE_URL
            value: https://n8n.example.com/
          - name: N8N_PROXY_HOPS
            value: "1"
          - name: N8N_RUNNERS_MODE
            value: external
          - name: N8N_RUNNERS_BROKER_LISTEN_ADDRESS
            value: 0.0.0.0
          - name: N8N_RUNNERS_AUTH_TOKEN
            valueFrom:
              secretKeyRef:
                name: n8n-auth
                key: N8N_RUNNERS_AUTH_TOKEN
          - name: N8N_NATIVE_PYTHON_RUNNER
            value: "true"
          - name: N8N_DEFAULT_BINARY_DATA_MODE
            value: filesystem
          # TLS to the database:
          - name: DB_POSTGRESDB_SSL_ENABLED
            value: "true"
          - name: DB_POSTGRESDB_SSL_CA_FILE
            value: /etc/secrets/ca.crt
          - name: DB_POSTGRESDB_SSL_REJECT_UNAUTHORIZED
            value: "true"
        volumeMounts:
          - name: n8n-data
            mountPath: /home/node/.n8n
          - name: db-ca
            mountPath: /etc/secrets
            readOnly: true
        readinessProbe:
          httpGet:
            path: /healthz
            port: 5678
          initialDelaySeconds: 30
          periodSeconds: 15
        livenessProbe:
          httpGet:
            path: /healthz
            port: 5678
          initialDelaySeconds: 60
          periodSeconds: 30
      - name: runners
        image: n8nio/runners:2.38.1
        env:
          - name: N8N_RUNNERS_TASK_BROKER_URI
            value: http://localhost:5679
          - name: N8N_RUNNERS_AUTH_TOKEN
            valueFrom:
              secretKeyRef:
                name: n8n-auth
                key: N8N_RUNNERS_AUTH_TOKEN
          - name: N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT
            value: "15"
        resources:
          requests:
            memory: "256Mi"
            cpu: "100m"
          limits:
            memory: "512Mi"
            cpu: "250m"
    volumes:
      - name: n8n-data
        persistentVolumeClaim:
          claimName: n8n-pvc
      - name: db-ca
        secret:
          secretName: n8n-db-ca
```

Those three TLS variables are the only ones n8n reads. `DB_POSTGRESDB_SSL_ENABLED` turns TLS on; `DB_POSTGRESDB_SSL_CA_FILE` points at the mounted CA file (use `DB_POSTGRESDB_SSL_CA` instead if

you want to pass the certificate contents directly); `DB_POSTGRESDB_SSL_REJECT_UNAUTHORIZED` defaults to true and is spelled out here so nobody quietly turns it off. There is no `DB_POSTGRESDB_SSL`, no `DB_POSTGRESDB_SSL_MODE`, and no `DB_POSTGRESDB_CA_CERT_FILE` — n8n ignores names it does not recognise, so a typo here does not error, it just connects without verifying anything.

Apply the updated deployment:

```
kubectl apply -f n8n-deployment.yaml
```

- Check rollout status:

  ```
  kubectl -n n8n rollout status deployment/n8n
  ```

> The pod still mounts `n8n-pvc`, which now holds only local settings, community nodes, filesystem binary data, and log files — workflow data lives in Postgres. The claim is `ReadWriteOnce`, so it can be attached on one node at a time; that is one more reason a second pod is not an option here. If you later need several pods to share a volume, switch to an NFS or CSI-based StorageClass that supports `ReadWriteMany`, as noted in **Kubernetes: Single Pod**.

---

5. (Optional) Restrict Egress with NetworkPolicy

Lock down pod egress so only the external DB hostname is reachable:

```yaml
# n8n-egress-policy.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: n8n-egress
  namespace: n8n
spec:
  podSelector:
    matchLabels:
      app: n8n
  policyTypes:
    - Egress
  egress:
    - to:
        - namespaceSelector: {}     # allow DNS lookup within cluster
          podSelector: {}
      ports:
        - protocol: UDP
          port: 53                  # DNS queries
    - to:
        - ipBlock:
            cidr: 203.0.113.0/24    # external DB network or CIDR
      ports:
        - protocol: TCP
          port: 5432
```

Apply:

```
kubectl apply -f n8n-egress-policy.yaml
```

- Verify:

  ```
  kubectl -n n8n get networkpolicy
  ```

---

6. Monitoring and Troubleshooting

6.1 Pod Logs

```
kubectl -n n8n logs -l app=n8n -c n8n
kubectl -n n8n logs -l app=n8n -c runners
```

6.2 Describe Deployment and Service

```
kubectl -n n8n describe deployment n8n
kubectl -n n8n describe service n8n-external-db
```

6.3 Verify DNS Resolution

Exec into a pod and test:

```
kubectl -n n8n exec -it $(kubectl -n n8n get pod -l app=n8n -o jsonpath='{.items[0].metadata.name}') -
        c n8n -- sh
# inside container:
nslookup n8n-external-db
pg_isready -h n8n-external-db -U n8nuser -p 5432
```

---

7. Performing Updates

After changing any credential or image version (bump `n8n` and `runners` together):

```
kubectl -n n8n rollout restart deployment n8n
```

- Monitor rollout:

  ```
  kubectl -n n8n rollout status deployment/n8n
  ```

---

8. Advanced Scenarios

8.1 Cloud SQL Proxy for IAM-Based Auth

When using Google Cloud SQL with IAM, run the **Cloud SQL Proxy** as a third container in the pod. `containers:` and `volumes:` are siblings under `spec.template.spec`, so they take the same indentation — add the entries below to the lists you already have rather than starting new ones:

```
    spec:
      containers:
        # ... the n8n and runners containers from section 4 ...
        - name: cloud-sql-proxy
          image: gcr.io/cloudsql-docker/gce-proxy:1.33.1
          command:
            - "/cloud_sql_proxy"
            - "-instances=my-project:us-central1:my-db-instance=tcp:5432"
            - "-credential_file=/secrets/gcp-key.json"
          volumeMounts:
            - name: gcp-credentials
              mountPath: /secrets
              readOnly: true
      volumes:
        # ... the n8n-data and db-ca volumes from section 4 ...
        - name: gcp-credentials
          secret:
            secretName: gcp-service-account-key
```

With the proxy in the pod, set `DB_POSTGRESDB_HOST` to `localhost` on the n8n container and drop the TLS variables — the proxy terminates TLS to Cloud SQL for you.

8.2 Read Replicas

For read-intensive workflows, configure n8n to use a **read replica**:

- Provide both **primary** and **replica** hostnames in env vars
- Customize connection logic in a **ConfigMap** or **init container**

---

9. Troubleshooting Common Issues

Issue	Cause	Solution
Cannot resolve ExternalName service	DNS policy or CoreDNS config	Ensure `externalName` is allowed, check `kubectl logs -n kube-system -l k8s-app=kube-dns`
Connection timed out	NetworkPolicy blocking egress or firewall rules	Adjust NetworkPolicy or cloud VPC firewall to allow egress on port 5432
SSL handshake failed	Missing or incorrect CA cert	Check that the Secret's key is `ca.crt`, that it mounts at `/etc/secrets`, and that `DB_POSTGRESDB_SSL_ENABLED` and `DB_POSTGRESDB_SSL_CA_FILE` are set
Connection succeeds but is not verified	An invented variable such as `DB_POSTGRESDB_SSL` or `DB_POSTGRESDB_SSL_MODE`	n8n ignores unknown names silently; use `DB_POSTGRESDB_SSL_ENABLED`, `DB_POSTGRESDB_SSL_CA_FILE`, `DB_POSTGRESDB_SSL_REJECT_UNAUTHORIZED`
`/etc/secrets/ca.crt` not found in the pod	Secret created with `kubectl create secret tls`, which stores `tls.crt` / `tls.key`	Re-create it with `kubectl create secret generic n8n-db-ca --from-file=ca.crt=ca.crt`
Rolling restart hanging	Readiness probe failing due to DB latency	Increase `initialDelaySeconds` on readinessProbe
Credentials not updated	Secret was changed but not re-mounted in pods	Delete pods to force a re-mount: `kubectl -n n8n delete pod -l app=n8n`
`Multi-Attach error for volume` on a new pod	More than one replica, or a rolling update, against a `ReadWriteOnce` PVC	Keep `replicas: 1` and `strategy: type: Recreate`; use a `ReadWriteMany` StorageClass if several pods must share the volume
Cron and poll triggers fire twice	A second main pod was started	Return to one main pod; add capacity with queue mode workers instead
Code nodes hang or fail	Runner container not ready, or token mismatch	Check `kubectl logs -c runners`; confirm both containers read `N8N_RUNNERS_AUTH_TOKEN` from the same `n8n-auth` Secret

---

Further reading

- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/deployment
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/enable-queue-mode
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/handle-binary-data

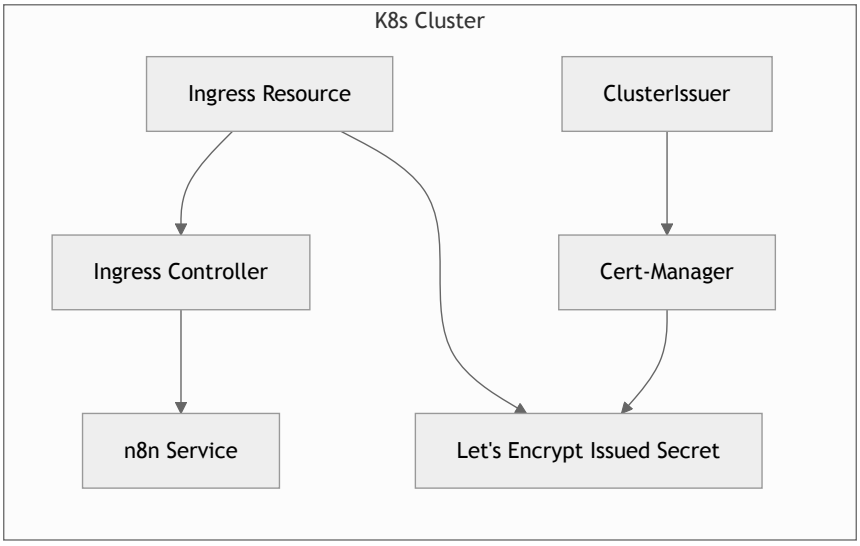
---

Chapter 19: Kubernetes: Ingress and Let’s Encrypt

Overview

Securing your n8n application with HTTPS and routing external traffic is essential for production readiness. Using a Kubernetes **Ingress** controller together with **Let’s Encrypt** (via Cert-Manager) automates TLS certificate provisioning and renewal. In this chapter, you will learn to:

1. **Install Cert-Manager** in your cluster
2. **Deploy an Ingress Controller** (NGINX, Traefik, or Contour)
3. **Configure a ClusterIssuer** for Let’s Encrypt (staging and production)
4. **Write an Ingress** manifest with TLS and websocket-friendly annotations
5. **Set up DNS records** to point your domain to the Ingress LoadBalancer
6. **Observe certificate issuance** and verify Ingress status
7. **Enable HTTP-to-HTTPS redirection** and custom error pages
8. **Troubleshoot** common certificate or routing failures
9. **Advanced scenarios**: wildcard certificates, private CA, mutual TLS



Diagram

Prerequisites

- A running **Kubernetes cluster** with `kubectl` access
- The **n8n Deployment** and **Service** already applied in the `n8n` namespace (see Kubernetes: Single Pod or Kubernetes: External Database)
- A valid **domain name** (e.g. `n8n.example.com`) and DNS control
- `helm` installed locally (optional but recommended for Cert-Manager)

1. Install Cert-Manager

Cert-Manager automates certificate issuance using ACME (Let's Encrypt).

1.1 Using Helm

```
# Add Jetstack repo
helm repo add jetstack https://charts.jetstack.io
helm repo update

# Install Cert-Manager CRDs
kubectl apply --validate=false -f https://github.com/jetstack/cert-
        manager/releases/download/v1.13.0/cert-manager.crds.yaml

# Install Cert-Manager
helm install cert-manager jetstack/cert-manager \
  --namespace cert-manager --create-namespace \
  --version v1.13.0
```

1.2 Verify Installation

```
kubectl get pods -n cert-manager
# Expect pods: cert-manager, cert-manager-cainjector, cert-manager-webhook
```

---

2. Deploy an Ingress Controller

Choose an Ingress implementation; here we use **NGINX Ingress** via Helm.

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
helm repo update

helm install ingress-nginx ingress-nginx/ingress-nginx \
  --namespace ingress-nginx --create-namespace \
  --set controller.publishService.enabled=true
```

- `publishService.enabled` exposes the LoadBalancer IP.

Verify:

```
kubectl get svc -n ingress-nginx
# Note the EXTERNAL-IP for later DNS configuration
```

---

3. Configure Let’s Encrypt ClusterIssuers

Create staging and production ClusterIssuers.

3.1 Staging Issuer (Testing)

```
# staging-issuer.yaml
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-staging
spec:
  acme:
    server: https://acme-staging-v02.api.letsencrypt.org/directory
    email: you@example.com
    privateKeySecretRef:
      name: letsencrypt-staging-key
    solvers:
      - http01:
          ingress:
            class: nginx
```

Apply:

```
kubectl apply -f staging-issuer.yaml
```

3.2 Production Issuer

```
# production-issuer.yaml
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
spec:
  acme:
    server: https://acme-v02.api.letsencrypt.org/directory
    email: you@example.com
    privateKeySecretRef:
      name: letsencrypt-prod-key
    solvers:
      - http01:
          ingress:
            class: nginx
```

Apply:

```
kubectl apply -f production-issuer.yaml
```

Verify:

```
kubectl get clusterissuer
```

---

4. Configure DNS

Point your domain (e.g., `n8n.example.com`) to the Ingress controller's LoadBalancer IP gathered earlier:

- Create an **A record**: `n8n.example.com` → `<EXTERNAL-IP>`

Wait for DNS propagation (`dig n8n.example.com`).

---

5. Create Ingress Resource with TLS

n8n's editor is a single-page app that keeps a websocket open for live execution updates. A default NGINX Ingress proxy timeout (60 seconds) will silently drop that connection during long-running workflows, so raise the read/send timeouts alongside the usual TLS and redirect annotations. Define `n8n-ingress.yaml`:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: n8n-ingress
  namespace: n8n
  annotations:
    kubernetes.io/ingress.class: "nginx"
    cert-manager.io/cluster-issuer: "letsencrypt-prod"
    nginx.ingress.kubernetes.io/force-ssl-redirect: "true"
    nginx.ingress.kubernetes.io/proxy-body-size: "50m"
    nginx.ingress.kubernetes.io/proxy-read-timeout: "3600"
    nginx.ingress.kubernetes.io/proxy-send-timeout: "3600"
spec:
  tls:
    - hosts:
        - n8n.example.com
      secretName: n8n-tls
  rules:
    - host: n8n.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: n8n-service
                port:
                  number: 80
```

Apply:

```
kubectl apply -f n8n-ingress.yaml
```

Make sure the n8n Deployment's `N8N_HOST`, `WEBHOOK_URL`, and `N8N_EDITOR_BASE_URL` (set in Kubernetes: Single Pod or Kubernetes: External Database) match this Ingress host exactly — a mismatch produces broken webhook URLs even once TLS is working.

---

6. Observe Certificate Issuance

Monitor Cert-Manager and Ingress:

```
kubectl -n n8n describe certificate n8n-tls
kubectl get certificate -n n8n
kubectl get orders -n n8n
kubectl logs -n cert-manager -l app=cert-manager
```

- The `Certificate` resource should transition to **Ready** within a few minutes.

---

7. Verify HTTPS Access

Open:

```
https://n8n.example.com
```

- The browser shows a valid Let’s Encrypt certificate (expires in 90 days; Cert-Manager renews it automatically).
- HTTP requests are automatically redirected to HTTPS.

---

8. Custom Error Pages and Advanced Annotations

8.1 Custom 502/503 Pages

Deploy a ConfigMap with custom pages and reference it via annotations:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: n8n-error-pages
  namespace: n8n
data:
  50x.html: |
    <html><body><h1>Service Unavailable</h1><p>Please try again later.</p></body></html>
```

Ingress annotation:

```
nginx.ingress.kubernetes.io/custom-http-errors: "502,503,504"
nginx.ingress.kubernetes.io/server-snippet: |
  error_page 502 503 504 /50x.html;
  location = /50x.html {
    root /etc/nginx/html;
  }
```

Mount the ConfigMap via the NGINX Ingress controller's ConfigMap or volume.

8.2 Wildcard Certificates

Use a DNS-01 solver with Cert-Manager and your DNS provider:

```
# wildcard-issuer.yaml
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-dns
spec:
  acme:
    server: https://acme-v02.api.letsencrypt.org/directory
    email: you@example.com
    privateKeySecretRef:
      name: letsencrypt-dns-key
    solvers:
      - dns01:
          cloudflare:
            email: you@domain.com
            apiTokenSecretRef:
              name: cloudflare-api-token
              key: api-token
```

---

9. Troubleshooting Common Issues

Issue	Cause	Solution
Certificate stuck in “Pending”	DNS record not pointing correctly, or ACME challenge failing	Verify the DNS A record, check Cert-Manager logs, ensure the Ingress class matches the solver
Ingress not picking up rules	Wrong `ingress.class` annotation	Ensure `kubernetes.io/ingress.class=nginx` matches your controller
HTTP not redirecting to HTTPS	Missing `force-ssl-redirect` annotation or ConfigMap override	Add `nginx.ingress.kubernetes.io/force-ssl-redirect: "true"`
Editor disconnects during long workflows	Proxy timeout shorter than the websocket connection	Raise `proxy-read-timeout` / `proxy-send-timeout`, confirm `N8N_PROXY_HOPS` is set correctly
Rate limit from Let’s Encrypt	Frequent test renewals or ACME limit exceeded	Switch to the staging issuer while testing, wait an hour before retrying production
Custom error pages not served	NGINX Ingress controller not mounting the ConfigMap volume	Configure the Ingress controller’s ConfigMap or volume mounts correctly

---

Further reading

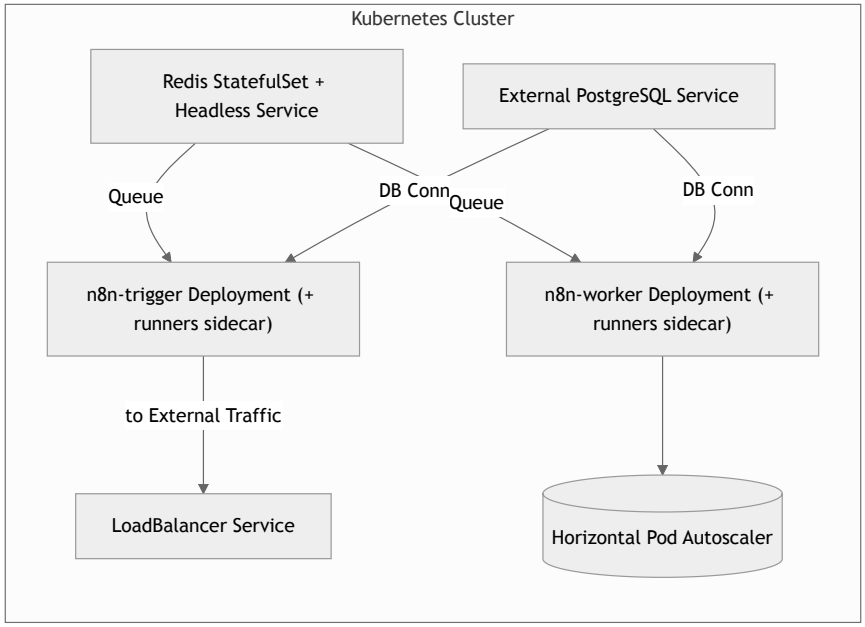
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/deployment
- https://cert-manager.io/docs/configuration/acme/
- https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuration/annotations/

Chapter 20: Kubernetes: Scaled Queue Mode

Overview

In a production Kubernetes environment handling high volumes of workflows, you separate the n8n trigger instance (receiving webhooks and scheduling executions) from worker instances (running the workflows). With Deployments, StatefulSets, and Horizontal Pod Autoscalers (HPAs), you can:

- **Scale workers independently** of the trigger for burst workloads
- **Ensure high availability** with multiple replicas and self-healing
- **Persist queue state** reliably via a Redis StatefulSet with durable storage
- **Automate scaling** based on CPU, memory, or custom metrics
- **Give every trigger and worker pod its own task runners sidecar**, since each broker connection is local to one pod



Diagram

Prerequisites

- A Kubernetes cluster (v1.24+) with **Metrics Server** or custom metrics enabled
- `kubectl` configured and authenticated
- A **PersistentVolumeClaim** for `n8n-data` (see Kubernetes: Single Pod)
- An **ExternalName Service** for PostgreSQL and the `n8n-db-credentials` Secret (see Kubernetes: External Database)
- **StorageClass** available for the Redis PVC (e.g., `standard`)

1. Deploy Redis as a StatefulSet

Create a headless Service and StatefulSet for Redis, enabling data persistence:

```yaml
# redis-statefulset.yaml
apiVersion: v1
kind: Service
metadata:
  name: redis
  namespace: n8n
  labels:
    app: redis
spec:
  ports:
    - port: 6379
      name: redis
  clusterIP: None   # headless service for StatefulSet
  selector:
    app: redis

---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
  namespace: n8n
spec:
  serviceName: redis
  replicas: 1
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
        - name: redis
          image: redis:7-alpine
          command: ["redis-server", "--appendonly", "yes"]
          ports:
            - containerPort: 6379
              name: redis
          volumeMounts:
            - name: redis-data
              mountPath: /data
          readinessProbe:
            exec:
              command:
                - redis-cli
                - ping
            initialDelaySeconds: 10
            periodSeconds: 10
          livenessProbe:
            exec:
              command:
                - redis-cli
                - ping
            initialDelaySeconds: 30
            periodSeconds: 30
  volumeClaimTemplates:
    - metadata:
        name: redis-data
      spec:
        accessModes: ["ReadWriteOnce"]
        storageClassName: standard
        resources:
          requests:
            storage: 1Gi
```

Apply:

```
kubectl apply -f redis-statefulset.yaml
```

Verify:

```
kubectl -n n8n get statefulset,svc -l app=redis
```

---

2. Configure Environment Secrets

Make sure the `n8n-auth` and `n8n-db-credentials` Secrets from earlier chapters exist. If this is a fresh namespace, create them now:

```
# Encryption key and runners auth token (never basic auth — n8n has none)
kubectl -n n8n create secret generic n8n-auth \
  --from-literal=N8N_ENCRYPTION_KEY="$(openssl rand -hex 32)" \
  --from-literal=N8N_RUNNERS_AUTH_TOKEN="$(openssl rand -hex 32)"

# Database credentials (ExternalName service)
kubectl -n n8n create secret generic n8n-db-credentials \
  --from-literal=DB_PORT=5432 \
  --from-literal=DB_NAME=n8n \
  --from-literal=DB_USER=n8nuser \
  --from-literal=DB_PASSWORD=SuperSecretExternal
```

---

3. Deploy the Trigger Deployment

The trigger handles incoming requests and enqueues executions. It gets the full URL variable set so webhook and editor links resolve correctly behind your load balancer, plus its own runners sidecar:

```yaml
# n8n-trigger-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: n8n-trigger
  namespace: n8n
spec:
  replicas: 1
  selector:
    matchLabels:
      app: n8n-trigger
  template:
    metadata:
      labels:
        app: n8n-trigger
    spec:
      terminationGracePeriodSeconds: 90
      containers:
      - name: n8n
        image: n8nio/n8n:2.38.1
        ports:
          - containerPort: 5678
        env:
          - name: DB_TYPE
            value: postgresdb
          - name: DB_POSTGRESDB_HOST
            value: n8n-external-db
          - name: DB_POSTGRESDB_PORT
            valueFrom:
              secretKeyRef:
                name: n8n-db-credentials
                key: DB_PORT
          - name: DB_POSTGRESDB_DATABASE
            valueFrom:
              secretKeyRef:
                name: n8n-db-credentials
                key: DB_NAME
          - name: DB_POSTGRESDB_USER
            valueFrom:
              secretKeyRef:
                name: n8n-db-credentials
                key: DB_USER
          - name: DB_POSTGRESDB_PASSWORD
            valueFrom:
              secretKeyRef:
                name: n8n-db-credentials
                key: DB_PASSWORD
          - name: EXECUTIONS_MODE
            value: queue
          - name: QUEUE_BULL_REDIS_HOST
            value: redis
          - name: QUEUE_BULL_REDIS_PORT
            value: "6379"
          - name: OFFLOAD_MANUAL_EXECUTIONS_TO_WORKERS
            value: "true"
          - name: N8N_DEFAULT_BINARY_DATA_MODE
            value: database
          - name: N8N_ENCRYPTION_KEY
            valueFrom:
              secretKeyRef:
                name: n8n-auth
                key: N8N_ENCRYPTION_KEY
          - name: N8N_HOST
            value: n8n.example.com
          - name: N8N_PROTOCOL
            value: https
          - name: WEBHOOK_URL
            value: https://n8n.example.com/
          - name: N8N_EDITOR_BASE_URL
```

```
            value: https://n8n.example.com/
          - name: N8N_PROXY_HOPS
            value: "1"
          - name: N8N_PORT
            value: "5678"
          - name: N8N_RUNNERS_MODE
            value: external
          - name: N8N_RUNNERS_BROKER_LISTEN_ADDRESS
            value: 0.0.0.0
          - name: N8N_RUNNERS_AUTH_TOKEN
            valueFrom:
              secretKeyRef:
                name: n8n-auth
                key: N8N_RUNNERS_AUTH_TOKEN
          - name: N8N_NATIVE_PYTHON_RUNNER
            value: "true"
          - name: N8N_GRACEFUL_SHUTDOWN_TIMEOUT
            value: "60"
        volumeMounts:
          - name: n8n-data
            mountPath: /home/node/.n8n
        readinessProbe:
          httpGet:
            path: /healthz
            port: 5678
          initialDelaySeconds: 30
          periodSeconds: 15
        livenessProbe:
          httpGet:
            path: /healthz
            port: 5678
          initialDelaySeconds: 60
          periodSeconds: 30
      - name: runners
        image: n8nio/runners:2.38.1
        env:
          - name: N8N_RUNNERS_TASK_BROKER_URI
            value: http://localhost:5679
          - name: N8N_RUNNERS_AUTH_TOKEN
            valueFrom:
              secretKeyRef:
                name: n8n-auth
                key: N8N_RUNNERS_AUTH_TOKEN
          - name: N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT
            value: "15"
      volumes:
        - name: n8n-data
          persistentVolumeClaim:
            claimName: n8n-pvc
```

> Because `OFFLOAD_MANUAL_EXECUTIONS_TO_WORKERS=true` routes even editor “Execute Workflow” runs to the worker pool, the trigger’s own runner mostly sits idle. Keep it anyway — expression evaluation and pinned-data previews in the editor still use it — it just won’t see production Code node traffic.

Apply:

```
kubectl apply -f n8n-trigger-deployment.yaml
```

Expose via a Service:

```
# n8n-trigger-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: n8n-trigger-svc
  namespace: n8n
spec:
  type: LoadBalancer
  selector:
    app: n8n-trigger
  ports:
```

```
    - port: 80
      targetPort: 5678
```

```
kubectl apply -f n8n-trigger-service.yaml
```

---

4. Deploy the Worker Deployment with HPA

Workers process queued executions and, in this book's reference stack, each needs its **own** runners sidecar — the broker connection is pod-local, not shared across replicas the way it would have to be over Swarm's overlay network. The worker container itself opens the broker (`N8N_RUNNERS_BROKER_LISTEN_ADDRESS=0.0.0.0` on port 5679); its sidecar dials `localhost:5679`, exactly like the trigger pod:

```yaml
# n8n-worker-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: n8n-worker
  namespace: n8n
spec:
  replicas: 3
  selector:
    matchLabels:
      app: n8n-worker
  template:
    metadata:
      labels:
        app: n8n-worker
    spec:
      terminationGracePeriodSeconds: 90
      containers:
      - name: n8n-worker
        image: n8nio/n8n:2.38.1
        command: ["n8n", "worker", "--concurrency=10"]
        ports:
          - containerPort: 5678
          - containerPort: 5679
        env:
          - name: DB_TYPE
            value: postgresdb
          - name: DB_POSTGRESDB_HOST
            value: n8n-external-db
          - name: DB_POSTGRESDB_PORT
            valueFrom:
              secretKeyRef:
                name: n8n-db-credentials
                key: DB_PORT
          - name: DB_POSTGRESDB_DATABASE
            valueFrom:
              secretKeyRef:
                name: n8n-db-credentials
                key: DB_NAME
          - name: DB_POSTGRESDB_USER
            valueFrom:
              secretKeyRef:
                name: n8n-db-credentials
                key: DB_USER
          - name: DB_POSTGRESDB_PASSWORD
            valueFrom:
              secretKeyRef:
                name: n8n-db-credentials
                key: DB_PASSWORD
          - name: EXECUTIONS_MODE
            value: queue
          - name: QUEUE_BULL_REDIS_HOST
            value: redis
          - name: QUEUE_BULL_REDIS_PORT
            value: "6379"
          - name: QUEUE_HEALTH_CHECK_ACTIVE
            value: "true"
          - name: N8N_DEFAULT_BINARY_DATA_MODE
            value: database
          - name: N8N_ENCRYPTION_KEY
            valueFrom:
              secretKeyRef:
                name: n8n-auth
                key: N8N_ENCRYPTION_KEY
          - name: N8N_HOST
```

```
            value: n8n.example.com
          - name: N8N_PROTOCOL
            value: https
          - name: WEBHOOK_URL
            value: https://n8n.example.com/
          - name: N8N_EDITOR_BASE_URL
            value: https://n8n.example.com/
          - name: N8N_PROXY_HOPS
            value: "1"
          - name: N8N_RUNNERS_MODE
            value: external
          - name: N8N_RUNNERS_BROKER_LISTEN_ADDRESS
            value: 0.0.0.0
          - name: N8N_RUNNERS_AUTH_TOKEN
            valueFrom:
              secretKeyRef:
                name: n8n-auth
                key: N8N_RUNNERS_AUTH_TOKEN
          - name: N8N_NATIVE_PYTHON_RUNNER
            value: "true"
          - name: N8N_GRACEFUL_SHUTDOWN_TIMEOUT
            value: "60"
        resources:
          requests:
            cpu: "250m"
            memory: "512Mi"
          limits:
            cpu: "500m"
            memory: "1Gi"
        readinessProbe:
          httpGet:
            path: /healthz/readiness
            port: 5678
          initialDelaySeconds: 10
          periodSeconds: 15
        livenessProbe:
          httpGet:
            path: /healthz
            port: 5678
          initialDelaySeconds: 30
          periodSeconds: 30
      - name: runners
        image: n8nio/runners:2.38.1
        env:
          - name: N8N_RUNNERS_TASK_BROKER_URI
            value: http://localhost:5679
          - name: N8N_RUNNERS_AUTH_TOKEN
            valueFrom:
              secretKeyRef:
                name: n8n-auth
                key: N8N_RUNNERS_AUTH_TOKEN
          - name: N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT
            value: "15"
```

> **Probe the worker on 5678, never on 5679.** Setting `QUEUE_HEALTH_CHECK_ACTIVE=true` makes a worker serve `/healthz` and `/healthz/readiness` on port 5678, the same port a main instance uses. `/healthz/readiness` additionally reports on the database and Redis connections, which is what readiness should mean for a worker. Port 5679 is the task runner broker, not a health endpoint: a TCP probe there only proves a socket is open, and if you ever disable external runners the port never opens at all and the pod stays unready forever. Neither probe catches a worker that is up but stuck on a stalled job — watch queue depth in Redis for that.

Apply:

```
kubectl apply -f n8n-worker-deployment.yaml
```

Create an HPA:

```
# n8n-worker-hpa.yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
```

```yaml
metadata:
  name: n8n-worker-hpa
  namespace: n8n
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: n8n-worker
  minReplicas: 2
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 60
```

Apply:

```
kubectl apply -f n8n-worker-hpa.yaml
```

Verify HPA:

```
kubectl -n n8n get hpa n8n-worker-hpa
```

---

5. Testing and Verification

- **Scale Test**: Run multiple workflows concurrently and watch the HPA scale workers:

```
kubectl -n n8n top pods
kubectl -n n8n get hpa
```

- **Load Balancer**: Note the external IP of `n8n-trigger-svc`:

```
kubectl -n n8n get svc n8n-trigger-svc
```

- **Logs**:

```
kubectl -n n8n logs -l app=n8n-worker -c n8n-worker --tail=50
kubectl -n n8n logs -l app=n8n-worker -c runners --tail=50
```

---

6. Backup and Restore

- **Redis**: Back up the PVC via host or CronJob:

```
kubectl -n n8n exec statefulset/redis -- \
  tar czf /data/redis_backup_$(date +%F).tar.gz -C /data .
```

- **Postgres**: Use external DB backup strategies (RDS snapshots, Cloud SQL backups) as covered in Kubernetes: External Database.

---

7. Troubleshooting Common Issues

Issue	Cause	Solution
Redis StatefulSet stuck in Pending	StorageClass not available or PVC binding failure	Verify the StorageClass exists, inspect `kubectl describe pvc redis-data-0 -n n8n`
Workers do not scale	Metrics Server missing or insufficient metrics	Install Metrics Server, verify `kubectl top pods` works, check HPA events
Trigger Service never acquires external IP	Cloud provider LoadBalancer quota or annotation missing	Check cloud LB quotas, ensure the Service type is LoadBalancer and the controller is running
High queue backlog	Too few workers or resource limits too low	Increase HPA `maxReplicas`, raise worker CPU/memory limits
Pod scheduling issues	NodeSelector or taints preventing placement	Inspect `kubectl describe pod`, remove or adjust node affinities
Worker pod ready but never processes jobs	Runner auth token mismatch, or worker can't reach Redis	Confirm the `n8n-worker` and `runners` containers share the same `N8N_RUNNERS_AUTH_TOKEN`, check `QUEUE_BULL_REDIS_HOST` connectivity
Pods killed mid-execution during a rollout	`terminationGracePeriodSeconds` shorter than `N8N_GRACEFUL_SHUTDOWN_TIMEOUT`	Keep the grace period longer than the shutdown timeout: 90 against 60
Worker pods never become ready	Probe pointed at port 5679, or `QUEUE_HEALTH_CHECK_ACTIVE` not set	Set `QUEUE_HEALTH_CHECK_ACTIVE=true` and probe `/healthz/readiness` on port 5678

Further reading

- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/enable-queue-mode
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/queue-mode
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/handle-binary-data

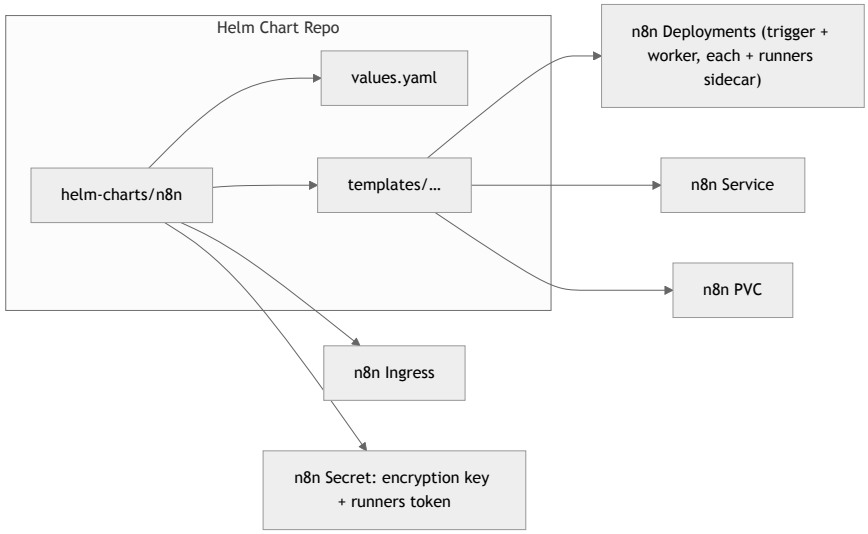
---

Chapter 21: Deploying with Helm

Overview

Packaging n8n as a **Helm chart** lets you deploy and manage complex Kubernetes configurations with a single command, parameterize settings for different environments, and version your deployments. In this chapter, you will learn to:

1. **Scaffold a Helm chart** for n8n
2. **Organize chart files** (`Chart.yaml`, `values.yaml`, `templates/…`)
3. **Define configurable values** for image, replicas, resources, persistence, the encryption key, task runners, database, Redis, and Ingress
4. **Use templates** (`_helpers.tpl`, `deployment.yaml`, `service.yaml`, `ingress.yaml`, `secrets.yaml`) with Go templating
5. **Package and install** the chart in your cluster
6. **Upgrade and rollback** releases
7. **Use subcharts** and **dependencies** for external services (Postgres, Redis)
8. **Implement hooks** for backups and migrations
9. **Debug and lint** your chart
10. **Publish** your chart to a Helm repository



Diagram

Prerequisites

- **Helm v3+** installed locally
- A **Kubernetes cluster** with `kubectl` context set
- Familiarity with **YAML**, **Go template syntax**, and **basic Helm commands**
- (Optional) A **chart repository** to publish to (GitHub Pages, ChartMuseum)

Should You Hand-Roll This Chart?

For most production deployments, install the actively maintained community chart at **https://github.com/8gears/n8n-helm-chart** instead of building your own. It already covers queue

mode, task runners, autoscaling, and Postgres/Redis subchart wiring, and has had far more real clusters exercise its edge cases than a chart you write in an afternoon.

This chapter still builds one from scratch, because understanding the moving pieces — the Secret, the runners sidecar, the queue-mode variables — is what lets you safely override the community chart's `values.yaml`, and some environments (an internal-only ChartMuseum, an air-gapped registry) require a chart your own team can audit line by line. Otherwise, run `helm repo add n8n https://8gears.github.io/n8n-helm-chart/` and use that instead.

---

1. Scaffold the Helm Chart

Use the Helm CLI to generate the chart skeleton:

```
helm create n8n
cd n8n
```

This creates:

```
n8n/
├── Chart.yaml
├── values.yaml
├── charts/
├── templates/
│   ├── deployment.yaml
│   ├── service.yaml
│   ├── ingress.yaml
│   ├── pvc.yaml
│   ├── secrets.yaml
│   ├── _helpers.tpl
│   └── NOTES.txt
└── .helmignore
```

---

2. Define Chart Metadata (`Chart.yaml`)

Edit `Chart.yaml` to reflect n8n-specific information. Pin `appVersion` to the n8n release the chart's defaults deploy — never leave it as `latest`:

```yaml
apiVersion: v2
name: n8n
description: A Helm chart for deploying n8n workflow automation
type: application
version: 0.1.0
appVersion: "2.38.1"
icon: https://raw.githubusercontent.com/n8n-io/n8n/master/assets/images/logo.png
keywords:
  - n8n
  - automation
  - workflow
sources:
  - https://github.com/n8n-io/n8n
maintainers:
  - name: YourName
    email: you@example.com
```

`version` above is the chart's own release number, required by Helm — unrelated to the obsolete `version:` key that used to appear in Compose files.

---

3. Configure Default Values (`values.yaml`)

Make `values.yaml` comprehensive and self-documenting. There is no basic-auth block — n8n has no basic auth — and the encryption key and runners auth token are first-class values, either supplied directly (for a quick test install) or, better, via an existing Secret you create out of band:

```yaml
# values.yaml

# Replica counts
replicaCount:
  trigger: 1
  worker: 3

# Image settings
image:
  repository: n8nio/n8n
  tag: "2.38.1"
  pullPolicy: IfNotPresent

# n8n container port
service:
  port: 5678
  type: LoadBalancer  # or ClusterIP, NodePort

# Public hostname n8n uses to build webhook and editor URLs
n8n:
  host: n8n.example.com
  protocol: https
  proxyHops: 1
  # Native Python in the Code node, executed by the external runners
  nativePythonRunner: true

# Encryption key and runners auth token.
# Either set these directly (fine for a throwaway test install) or set
# existingSecret to the name of a Secret you created yourself with keys
# N8N_ENCRYPTION_KEY and N8N_RUNNERS_AUTH_TOKEN — do that for production.
encryptionKey: ""
existingSecret: ""

# Task runners: every n8n process (trigger and worker) gets a sidecar
runners:
  enabled: true
  image:
    repository: n8nio/runners
    tag: "2.38.1"
  authToken: ""

# Graceful shutdown: keep terminationGracePeriodSeconds above
# N8N_GRACEFUL_SHUTDOWN_TIMEOUT so in-flight executions can finish before
# Kubernetes sends SIGKILL. These match the queue-mode chapters: 60 and 90.
gracefulShutdownTimeout: 60
terminationGracePeriodSeconds: 90

# Persistence (n8n workflow data)
persistence:
  enabled: true
  storageClass: standard
  size: 5Gi
  accessMode: ReadWriteOnce

# External Database (PostgreSQL)
database:
  type: postgresdb   # sqlite, postgresdb
  external:
    enabled: true
    host: external-db.example.com
    port: 5432
    database: n8n
  credentials:
    user: n8nuser
    password: SuperSecretExternal   # prefer existingSecret in production

# Redis queue mode
```

```yaml
redis:
  enabled: true
  external:
    enabled: false
    host: redis.example.com
    port: 6379

# Resources
resources:
  trigger:
    requests:
      cpu: "250m"
      memory: "512Mi"
    limits:
      cpu: "500m"
      memory: "1Gi"
  worker:
    requests:
      cpu: "250m"
      memory: "512Mi"
    limits:
      cpu: "500m"
      memory: "1Gi"

# Ingress configuration
ingress:
  enabled: true
  annotations:
    kubernetes.io/ingress.class: nginx
    cert-manager.io/cluster-issuer: letsencrypt-prod
    nginx.ingress.kubernetes.io/force-ssl-redirect: "true"
    nginx.ingress.kubernetes.io/proxy-body-size: "50m"
    nginx.ingress.kubernetes.io/proxy-read-timeout: "3600"
    nginx.ingress.kubernetes.io/proxy-send-timeout: "3600"
  hosts:
    - host: n8n.example.com
      paths:
        - /
  tls:
    - secretName: n8n-tls
      hosts:
        - n8n.example.com

# Horizontal Pod Autoscaler for workers
hpa:
  enabled: true
  minReplicas: 2
  maxReplicas: 10
  cpuUtilizationPercentage: 60
```

The `proxy-read-timeout` and `proxy-send-timeout` annotations keep the editor's websocket connection alive past NGINX's default 60-second timeout, so long executions don't appear to hang in the UI.

---

4. Create the Secret (`templates/secrets.yaml`)

```
{{- if not .Values.existingSecret }}
apiVersion: v1
kind: Secret
metadata:
  name: {{ include "n8n.fullname" . }}-secrets
type: Opaque
stringData:
  N8N_ENCRYPTION_KEY: {{ required "Set encryptionKey or existingSecret" .Values.encryptionKey | quote
      }}
  N8N_RUNNERS_AUTH_TOKEN: {{ required "Set runners.authToken or existingSecret"
      .Values.runners.authToken | quote }}
{{- end }}
```

`required` fails the install with a clear message instead of deploying n8n with an empty encryption key. Install with `--set encryptionKey=$(openssl rand -hex 32) --set runners.authToken=$(openssl rand -hex 32)` , or point `existingSecret` at a Secret you manage separately.

---

5. Template the Deployments (`templates/deployment.yaml`)

Use Go templating to generate two Deployments (trigger and worker), each with an `n8n` container and a `runners` sidecar container that talks to it over `localhost:5679`:

```
{{- $fullName := include "n8n.fullname" . -}}
{{- $secretName := .Values.existingSecret | default (printf "%s-secrets" $fullName) -}}
{{- $values := .Values -}}

apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ $fullName }}-trigger
  labels:
    app: {{ $fullName }}-trigger
spec:
  replicas: {{ $values.replicaCount.trigger }}
  selector:
    matchLabels:
      app: {{ $fullName }}-trigger
  template:
    metadata:
      labels:
        app: {{ $fullName }}-trigger
    spec:
      terminationGracePeriodSeconds: {{ $values.terminationGracePeriodSeconds }}
      containers:
        - name: n8n
          image: "{{ $values.image.repository }}:{{ $values.image.tag }}"
          imagePullPolicy: {{ $values.image.pullPolicy }}
          ports:
            - containerPort: {{ $values.service.port }}
          env:
            {{- include "n8n.triggerEnvVars" . | nindent 12 }}
          resources:
            {{- toYaml $values.resources.trigger | nindent 12 }}
          volumeMounts:
            - name: n8n-data
              mountPath: /home/node/.n8n
          livenessProbe:
            httpGet:
              path: /healthz
              port: {{ $values.service.port }}
            initialDelaySeconds: 60
            periodSeconds: 30
          readinessProbe:
            httpGet:
              path: /healthz
              port: {{ $values.service.port }}
            initialDelaySeconds: 30
            periodSeconds: 15
        {{- if $values.runners.enabled }}
        - name: runners
          image: "{{ $values.runners.image.repository }}:{{ $values.runners.image.tag }}"
          imagePullPolicy: {{ $values.image.pullPolicy }}
          env:
            {{- include "n8n.runnerEnvVars" . | nindent 12 }}
        {{- end }}
      volumes:
        - name: n8n-data
          persistentVolumeClaim:
            claimName: {{ $fullName }}-pvc

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ $fullName }}-worker
  labels:
    app: {{ $fullName }}-worker
spec:
  replicas: {{ $values.replicaCount.worker }}
  selector:
    matchLabels:
```

```
      app: {{ $fullName }}-worker
  template:
    metadata:
      labels:
        app: {{ $fullName }}-worker
    spec:
      terminationGracePeriodSeconds: {{ $values.terminationGracePeriodSeconds }}
      containers:
        - name: n8n-worker
          image: "{{ $values.image.repository }}:{{ $values.image.tag }}"
          imagePullPolicy: {{ $values.image.pullPolicy }}
          command: ["n8n", "worker", "--concurrency=10"]
          ports:
            - containerPort: {{ $values.service.port }}
            - containerPort: 5679
          env:
            {{- include "n8n.workerEnvVars" . | nindent 12 }}
          resources:
            {{- toYaml $values.resources.worker | nindent 12 }}
          readinessProbe:
            httpGet:
              path: /healthz/readiness
              port: {{ $values.service.port }}
            initialDelaySeconds: 10
            periodSeconds: 15
          livenessProbe:
            httpGet:
              path: /healthz
              port: {{ $values.service.port }}
            initialDelaySeconds: 30
            periodSeconds: 30
        {{- if $values.runners.enabled }}
        - name: runners
          image: "{{ $values.runners.image.repository }}:{{ $values.runners.image.tag }}"
          imagePullPolicy: {{ $values.image.pullPolicy }}
          env:
            {{- include "n8n.runnerEnvVars" . | nindent 12 }}
        {{- end }}
      # no extra volumes needed for the worker; workflow data lives in Postgres
```

Define helper partials in `templates/_helpers.tpl`:

```
{{- define "n8n.envVars" -}}
{{- $secretName := .Values.existingSecret | default (printf "%s-secrets" (include "n8n.fullname" .))
-}}
- name: DB_TYPE
  value: "{{ .Values.database.type }}"
{{- if .Values.database.external.enabled }}
- name: DB_POSTGRESDB_HOST
  value: "{{ .Values.database.external.host }}"
- name: DB_POSTGRESDB_PORT
  value: "{{ .Values.database.external.port }}"
- name: DB_POSTGRESDB_DATABASE
  value: "{{ .Values.database.external.database }}"
- name: DB_POSTGRESDB_USER
  value: "{{ .Values.database.credentials.user }}"
- name: DB_POSTGRESDB_PASSWORD
  value: "{{ .Values.database.credentials.password }}"
{{- end }}
{{- if .Values.redis.enabled }}
- name: EXECUTIONS_MODE
  value: queue
- name: QUEUE_BULL_REDIS_HOST
  value: "{{ if .Values.redis.external.enabled }}{{ .Values.redis.external.host }}{{ else }}redis{{
end }}"
- name: QUEUE_BULL_REDIS_PORT
  value: "{{ if .Values.redis.external.enabled }}{{ .Values.redis.external.port }}{{ else }}6379{{ end
}}"
- name: N8N_DEFAULT_BINARY_DATA_MODE
  value: database
{{- end }}
- name: N8N_ENCRYPTION_KEY
  valueFrom:
    secretKeyRef:
      name: {{ $secretName }}
      key: N8N_ENCRYPTION_KEY
- name: N8N_HOST
  value: "{{ .Values.n8n.host }}"
```

```
- name: N8N_PROTOCOL
  value: "{{ .Values.n8n.protocol }}"
- name: WEBHOOK_URL
  value: "{{ .Values.n8n.protocol }}://{{ .Values.n8n.host }}/"
- name: N8N_EDITOR_BASE_URL
  value: "{{ .Values.n8n.protocol }}://{{ .Values.n8n.host }}/"
- name: N8N_PROXY_HOPS
  value: "{{ .Values.n8n.proxyHops }}"
- name: N8N_PORT
  value: "{{ .Values.service.port }}"
- name: N8N_GRACEFUL_SHUTDOWN_TIMEOUT
  value: "{{ .Values.gracefulShutdownTimeout }}"
{{- if .Values.runners.enabled }}
- name: N8N_RUNNERS_MODE
  value: external
- name: N8N_RUNNERS_BROKER_LISTEN_ADDRESS
  value: "0.0.0.0"
- name: N8N_RUNNERS_AUTH_TOKEN
  valueFrom:
    secretKeyRef:
      name: {{ $secretName }}
      key: N8N_RUNNERS_AUTH_TOKEN
- name: N8N_NATIVE_PYTHON_RUNNER
  value: "{{ .Values.n8n.nativePythonRunner }}"
{{- end }}
{{- end }}

{{- define "n8n.triggerEnvVars" -}}
{{- include "n8n.envVars" . }}
{{- if .Values.redis.enabled }}
- name: OFFLOAD_MANUAL_EXECUTIONS_TO_WORKERS
  value: "true"
{{- end }}
{{- end }}

{{- define "n8n.workerEnvVars" -}}
{{- include "n8n.envVars" . }}
- name: QUEUE_HEALTH_CHECK_ACTIVE
  value: "true"
{{- end }}

{{- define "n8n.runnerEnvVars" -}}
{{- $secretName := .Values.existingSecret | default (printf "%s-secrets" (include "n8n.fullname" .)) 
-}}
- name: N8N_RUNNERS_TASK_BROKER_URI
  value: http://localhost:5679
- name: N8N_RUNNERS_AUTH_TOKEN
  valueFrom:
    secretKeyRef:
      name: {{ $secretName }}
      key: N8N_RUNNERS_AUTH_TOKEN
- name: N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT
  value: "15"
{{- end }}
```

`n8n.triggerEnvVars` adds `OFFLOAD_MANUAL_EXECUTIONS_TO_WORKERS` on top of the shared set so manual “Execute Workflow” runs from the editor go through the worker pool instead of the trigger, matching **Kubernetes: Scaled Queue Mode**.

`n8n.workerEnvVars` adds `QUEUE_HEALTH_CHECK_ACTIVE`, which makes each worker serve `/healthz` and `/healthz/readiness` on `service.port` (5678). That is what the worker probes above target. Do not probe port 5679: that is the task runner broker, and the earlier version of this chart probed it with a TCP check, so setting `runners.enabled: false` left every worker permanently unready because nothing ever opened the port. The health endpoints belong to the worker itself, so the probes now work either way, and `N8N_GRACEFUL_SHUTDOWN_TIMEOUT` has moved out of the `runners.enabled` block for the same reason — shutdown behaviour has nothing to do with runners.

---

6. Service, PVC, and Ingress Templates

`templates/service.yaml`

```
apiVersion: v1
kind: Service
metadata:
  name: {{ include "n8n.fullname" . }}-trigger-svc
spec:
  type: {{ .Values.service.type }}
  selector:
    app: {{ include "n8n.fullname" . }}-trigger
  ports:
    - port: 80
      targetPort: {{ .Values.service.port }}
      protocol: TCP
```

`templates/pvc.yaml`

```
{{- if .Values.persistence.enabled }}
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: {{ include "n8n.fullname" . }}-pvc
spec:
  accessModes:
    - {{ .Values.persistence.accessMode }}
  resources:
    requests:
      storage: {{ .Values.persistence.size }}
  storageClassName: {{ .Values.persistence.storageClass }}
{{- end }}
```

`templates/ingress.yaml`

```
{{- if .Values.ingress.enabled }}
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: {{ include "n8n.fullname" . }}-ingress
  annotations:
    {{- toYaml .Values.ingress.annotations | nindent 4 }}
spec:
  tls:
    {{- range .Values.ingress.tls }}
    - hosts:
        {{- range .hosts }}
        - {{ . | quote }}
        {{- end }}
      secretName: {{ .secretName }}
    {{- end }}
  rules:
    {{- range .Values.ingress.hosts }}
    - host: {{ .host | quote }}
      http:
        paths:
          {{- range .paths }}
          - path: {{ . }}
            pathType: Prefix
            backend:
              service:
                name: {{ include "n8n.fullname" $. | printf "%s-trigger-svc" | quote }}
                port:
                  number: 80
          {{- end }}
    {{- end }}
{{- end }}
```

7. Package and Install the Chart

Package:

```
helm package .
# produces n8n-0.1.0.tgz
```

Install, supplying the two required secrets inline for a quick test (use `existingSecret` instead in production):

```
helm install n8n-release n8n-0.1.0.tgz --namespace n8n --create-namespace \
  --set encryptionKey="$(openssl rand -hex 32)" \
  --set runners.authToken="$(openssl rand -hex 32)"
```

Monitor:

```
kubectl -n n8n get all
```

Check release status:

```
helm status n8n-release -n n8n
```

---

8. Upgrades and Rollbacks

Upgrade

Edit `values.yaml` or override values inline. Bump `image.tag` and `runners.image.tag` together — they must stay on the same n8n version:

```
helm upgrade n8n-release n8n-0.1.0.tgz \
  --namespace n8n \
  --set replicaCount.worker=5 \
  --set image.tag=2.39.0 \
  --set runners.image.tag=2.39.0
```

Rollback

```
helm rollback n8n-release 1 -n n8n
```

---

9. Dependencies and Subcharts

As an alternative to the external-database values used above, you can pull in Postgres and Redis as subcharts so the whole stack installs from one chart:

```
# Chart.yaml
dependencies:
  - name: postgresql
    version: 10.7.17
    repository: https://charts.bitnami.com/bitnami
  - name: redis
    version: 17.6.8
    repository: https://charts.bitnami.com/bitnami
```

Run:

```
helm dependency update
helm install n8n-release . -n n8n
```

Override subchart values under `postgresql:` and `redis:` keys in `values.yaml`. If you do this, point `database.external.host` and `redis.external.host` at the subchart's generated Service names instead of an outside host.

---

10. Hooks for Backups and Migrations

Implement pre-upgrade hooks:

```yaml
# templates/backup-hook.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: "{{ include "n8n.fullname" . }}-backup-preupgrade"
  annotations:
    "helm.sh/hook": pre-upgrade
    "helm.sh/hook-weight": "-5"
spec:
  template:
    spec:
      containers:
        - name: backup
          image: alpine
          command:
            - sh
            - -c
            - |
              tar czf /backup/n8n_data_preupgrade_$(date +%F).tar.gz -C /home/node/.n8n .
          volumeMounts:
            - name: n8n-data
              mountPath: /home/node/.n8n
            - name: backup
              mountPath: /backup
      volumes:
        - name: n8n-data
          persistentVolumeClaim:
            claimName: {{ include "n8n.fullname" . }}-pvc
        - name: backup
          persistentVolumeClaim:
            claimName: {{ include "n8n.fullname" . }}-backup-pvc
      restartPolicy: Never
  backoffLimit: 1
```

Both `volumeMounts` entries need a matching entry under `volumes:` — without it the API server rejects the Job with `volumeMounts[0].name: Not found: "n8n-data"` and the whole `helm upgrade` aborts before it changes anything. `volumes:` is a sibling of `containers:` under `spec.template.spec`, at the same indentation.

Add a second claim to `templates/pvc.yaml` for `{{ include "n8n.fullname" . }}-backup-pvc` so the archive outlives the Job. An `emptyDir` compiles, but the volume is deleted with the pod and the backup goes with it, which defeats the point of a pre-upgrade hook.

The `n8n-data` claim is `ReadWriteOnce`, so the hook Job and the trigger pod cannot mount it from different nodes at the same time. Either give the Job a node affinity matching the trigger pod, or take the backup from the database instead — for a Postgres deployment the workflow data is in Postgres anyway, and `pg_dump` is the backup that matters.

---

11. Linting and Testing

Lint your chart before deploying:

```
helm lint .
```

Run a dry-run install:

```
helm install n8n-release . --namespace n8n --dry-run --debug
```

---

12. Publishing Your Chart

- **GitHub Pages**: push the `.tgz` and the `index.yaml` generated by `helm repo index`.
- **ChartMuseum**: upload via its HTTP API.
- **Artifact Hub**: register your GitHub repo with a `charts/` directory.

Generate an index:

```
helm repo index ./ --url https://your.github.io/helm-charts
```

Users can then add your repo:

```
helm repo add my-charts https://your.github.io/helm-charts
helm repo update
helm install n8n my-charts/n8n -n n8n
```

---

13. Troubleshooting Common Issues

Issue	Cause	Solution
`helm install` fails with “Set encryptionKey or existingSecret”	Neither `encryptionKey` nor `existingSecret` was provided	Pass `--set encryptionKey=$(openssl rand -hex 32)` and `--set runners.authToken=...`, or set `existingSecret`
Worker pods never become ready	Readiness probe pointed at port 5679, or `QUEUE_HEALTH_CHECK_ACTIVE` missing	Probe `/healthz/readiness` on `service.port`; confirm `n8n.workerEnvVars` sets `QUEUE_HEALTH_CHECK_ACTIVE=true`
Every worker unready after setting `runners.enabled: false`	Probe depended on the runner broker port	The probe must not reference 5679; only the runners container and `N8N_RUNNERS_*` variables belong inside that conditional
`helm upgrade` aborts with `volumeMounts[0].name: Not found: "n8n-data"`	`templates/backup-hook.yaml` declares mounts with no `volumes:` block	Add the `volumes:` section shown in section 10, as a sibling of `containers:`
Hook Job stuck `Pending`, volume already attached	`ReadWriteOnce` PVC mounted by the trigger pod on another node	Give the Job matching node affinity, or back up from Postgres instead
Code nodes fail after an upgrade	`image.tag` and `runners.image.tag` drifted apart	Bump both together; the runner image must match the n8n version exactly
Pods killed mid-execution during `helm upgrade`	`terminationGracePeriodSeconds` at or below `gracefulShutdownTimeout`	Keep the defaults: 90 against 60
Editor websocket drops during long executions	NGINX default 60-second proxy timeout	Keep `proxy-read-timeout` and `proxy-send-timeout` in `ingress.annotations`

Further reading

- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/enable-queue-mode
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/queue-mode
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners
- https://docs.n8n.io/deploy/host-n8n/install-options/install-using-docker-compose

---

Chapter 22: AWS ECS Fargate

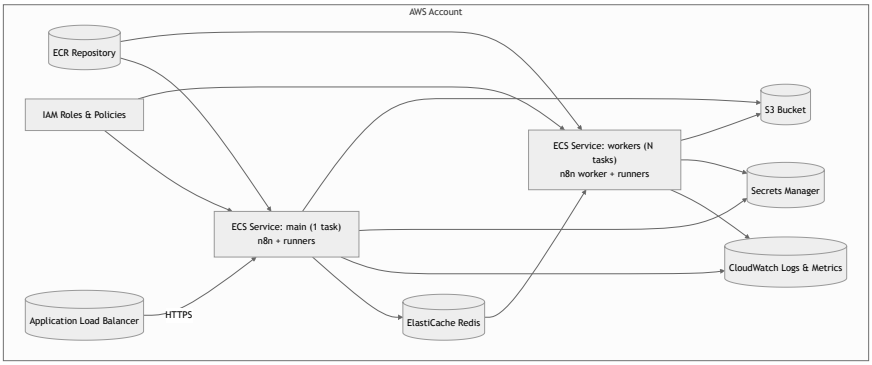
Overview

AWS Elastic Container Service (ECS) with Fargate runs containers without EC2 instances to manage. Fargate task storage is ephemeral and never shared, and ECS restarts tasks freely, so this chapter runs n8n in queue mode: one main task serving the editor and webhooks, a separate worker service running `n8n worker`, ElastiCache for Redis holding the queue, and binary data in Postgres or S3 rather than on a task's disk.

Community Edition runs exactly **one** main process. Multi-main is an Enterprise feature (`N8N_MULTI_MAIN_SETUP_ENABLED`), so the main service below stays at a desired count of 1 and you add capacity by scaling the worker service instead.

In this chapter, you will:

1. **Build and push** your n8n Docker image to Amazon ECR (Elastic Container Registry)
2. **Configure IAM Roles** for task execution and logging permissions
3. **Create an ECS Cluster** for Fargate tasks
4. **Provision ElastiCache for Redis** to hold the execution queue
5. **Set binary data mode** for an ephemeral, multi-task environment
6. **Define a main Task Definition** with the `n8n` container and its runners sidecar
7. **Define a worker Task Definition** running `n8n worker` with its own runners sidecar
8. **Set up a Load Balancer** (Application Load Balancer) with target groups and health checks
9. **Deploy two ECS Services**: one main task, N worker tasks
10. **Scale the worker service** via ECS Service Auto Scaling
11. **Enable CloudWatch Logs** for container output and metrics
12. **Configure secrets** from AWS Secrets Manager for the encryption key, runners auth token, and DB credentials



Diagram

Prerequisites

- **AWS CLI** installed and configured (`aws configure`)
- **Docker** installed locally
- **IAM permissions** for ECR, ECS, IAM roles, ALB, ElastiCache, CloudWatch, S3, Secrets Manager
- **A VPC** with two subnets in different AZs and an Internet Gateway or NAT
- **A PostgreSQL database** reachable from the VPC (RDS or self-managed)
- The **encryption key** and **runners auth token** you generated for the reference stack

1. Build and Push Docker Image to ECR

1.1 Create an ECR Repository

```
aws ecr create-repository \
  --repository-name n8n \
  --image-scanning-configuration scanOnPush=true \
  --region us-east-1
```

1.2 Authenticate Docker to ECR

```
aws ecr get-login-password --region us-east-1 \
  | docker login --username AWS --password-stdin <AWS_ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com
```

1.3 Build and Tag the Image

`FROM n8nio/n8n:2.38.1` in your `Dockerfile`. Tag the build with the same version, never `latest`, so a rollback is a one-line change:

```
docker build -t n8n:2.38.1 .
docker tag n8n:2.38.1 <AWS_ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/n8n:2.38.1
```

1.4 Push to ECR

```
docker push <AWS_ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/n8n:2.38.1
```

> **Scenario:** the CI/CD and GitOps chapter builds a full pipeline around bumping this pinned tag.

---

2. Define IAM Roles & Policies

2.1 Task Execution Role

ECS tasks need permission to pull images and write logs:

```
aws iam create-role \
  --role-name ecsTaskExecutionRole \
  --assume-role-policy-document file://ecs-execution-trust-policy.json
```

`ecs-execution-trust-policy.json`:

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Principal": { "Service": "ecs-tasks.amazonaws.com" },
    "Action": "sts:AssumeRole"
  }]
}
```

Attach managed policy:

```
aws iam attach-role-policy \
  --role-name ecsTaskExecutionRole \
  --policy-arn arn:aws:iam::aws:policy/service-role/AmazonECSTaskExecutionRolePolicy
```

2.2 Task Role (for S3 and Secrets)

For S3 or Secrets Manager access at runtime:

```
aws iam create-role \
  --role-name n8nTaskRole \
  --assume-role-policy-document file://ecs-task-trust-policy.json

aws iam attach-role-policy \
  --role-name n8nTaskRole \
  --policy-arn arn:aws:iam::aws:policy/AmazonS3FullAccess

aws iam attach-role-policy \
  --role-name n8nTaskRole \
  --policy-arn arn:aws:iam::aws:policy/SecretsManagerReadWrite
```

---

3. Create an ECS Cluster

```
aws ecs create-cluster --cluster-name n8n-cluster
```

Verify:

```
aws ecs describe-clusters --clusters n8n-cluster
```

---

4. Provision ElastiCache for Redis

Queue mode needs Redis. Create a single-node (or replicated) ElastiCache cluster in the same VPC as your tasks:

```
aws elasticache create-cache-cluster \
  --cache-cluster-id n8n-queue \
  --engine redis \
  --cache-node-type cache.t4g.small \
  --num-cache-nodes 1 \
  --cache-subnet-group-name n8n-cache-subnets \
  --security-group-ids sg-aaaa1111
```

Read the endpoint back once it is available:

```
aws elasticache describe-cache-clusters \
  --cache-cluster-id n8n-queue \
  --show-cache-node-info \
  --query "CacheClusters[0].CacheNodes[0].Endpoint"
```

That hostname becomes `QUEUE_BULL_REDIS_HOST` on both the main and worker task definitions; the port is `QUEUE_BULL_REDIS_PORT=6379`. If you enable encryption in transit, also set `QUEUE_BULL_REDIS_TLS=true`, and set `QUEUE_BULL_REDIS_PASSWORD` if you enable the Redis AUTH token. The task security group must allow outbound 6379 to the cache security group.

---

5. Decide How Binary Data Persists

Fargate task storage is ephemeral and never shared across tasks, and filesystem mode is not supported in queue mode at all, which rules out `N8N_DEFAULT_BINARY_DATA_MODE=filesystem`. Pick one:

- **Database mode** (used below): `N8N_DEFAULT_BINARY_DATA_MODE=database` sends binary data through the same Postgres connection. Simplest to operate; fine for moderate payloads.
- **S3 mode**: for heavier workloads, set `N8N_DEFAULT_BINARY_DATA_MODE=s3` and the `N8N_EXTERNAL_STORAGE_S3_*` variables (`_HOST` , `_BUCKET_NAME` , `_BUCKET_REGION` , `_ACCESS_KEY` , `_ACCESS_SECRET`) against a bucket you create:

  ```
  aws s3 mb s3://n8n-binary-data --region us-east-1
  ```

Whichever you choose, set the same value on the main and worker task definitions. Neither mode needs an EFS mount, so this chapter skips it.

---

6. Create the Main Task Definition

The task definition below runs **two containers in the same task**: `n8n` and its `runners` sidecar. Fargate tasks with `awsvpc` networking share a network namespace, so the two containers reach each other over `localhost` with no service discovery needed.

`n8n-task-definition.json`:

```
{
  "family": "n8n-task",
  "requiresCompatibilities": ["FARGATE"],
  "networkMode": "awsvpc",
  "cpu": "1024",
  "memory": "2048",
  "executionRoleArn": "arn:aws:iam::<AWS_ACCOUNT_ID>:role/ecsTaskExecutionRole",
  "taskRoleArn": "arn:aws:iam::<AWS_ACCOUNT_ID>:role/n8nTaskRole",
  "containerDefinitions": [
    {
      "name": "n8n",
      "image": "<AWS_ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/n8n:2.38.1",
      "portMappings": [{ "containerPort": 5678, "protocol": "tcp" }],
      "environment": [
        { "name": "DB_TYPE",                          "value": "postgresdb" },
        { "name": "DB_POSTGRESDB_HOST",               "value": "external-db.example.com" },
        { "name": "DB_POSTGRESDB_PORT",               "value": "5432" },
        { "name": "DB_POSTGRESDB_DATABASE",           "value": "n8n" },
        { "name": "DB_POSTGRESDB_USER",               "value": "n8nuser" },
        { "name": "DB_POSTGRESDB_SSL_ENABLED",        "value": "true" },
        { "name": "N8N_HOST",                         "value": "n8n.example.com" },
        { "name": "N8N_PROTOCOL",                     "value": "https" },
        { "name": "WEBHOOK_URL",                      "value": "https://n8n.example.com/" },
        { "name": "N8N_EDITOR_BASE_URL",              "value": "https://n8n.example.com/" },
        { "name": "N8N_PROXY_HOPS",                   "value": "1" },
        { "name": "EXECUTIONS_MODE",                  "value": "queue" },
        { "name": "QUEUE_BULL_REDIS_HOST",            "value": "n8n-
        queue.abc123.0001.use1.cache.amazonaws.com" },
        { "name": "QUEUE_BULL_REDIS_PORT",            "value": "6379" },
        { "name": "OFFLOAD_MANUAL_EXECUTIONS_TO_WORKERS", "value": "true" },
        { "name": "N8N_DEFAULT_BINARY_DATA_MODE",     "value": "database" },
        { "name": "N8N_RUNNERS_MODE",                 "value": "external" },
        { "name": "N8N_RUNNERS_BROKER_LISTEN_ADDRESS", "value": "0.0.0.0" },
        { "name": "N8N_NATIVE_PYTHON_RUNNER",         "value": "true" }
      ],
      "secrets": [
        { "name": "N8N_ENCRYPTION_KEY",       "valueFrom": "arn:aws:secretsmanager:us-east-1:
        <AWS_ACCOUNT_ID>:secret:n8n/encryption-key" },
        { "name": "N8N_RUNNERS_AUTH_TOKEN",   "valueFrom": "arn:aws:secretsmanager:us-east-1:
        <AWS_ACCOUNT_ID>:secret:n8n/runners-auth-token" },
        { "name": "DB_POSTGRESDB_PASSWORD",   "valueFrom": "arn:aws:secretsmanager:us-east-1:
        <AWS_ACCOUNT_ID>:secret:n8n/db-password" }
      ],
      "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
          "awslogs-group": "/ecs/n8n",
          "awslogs-region": "us-east-1",
          "awslogs-stream-prefix": "n8n"
        }
      }
    },
    {
      "name": "runners",
      "image": "<AWS_ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/n8n-runners:2.38.1",
      "environment": [
        { "name": "N8N_RUNNERS_TASK_BROKER_URI",       "value": "http://localhost:5679" },
        { "name": "N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT", "value": "15" }
      ],
      "secrets": [
        { "name": "N8N_RUNNERS_AUTH_TOKEN", "valueFrom": "arn:aws:secretsmanager:us-east-1:
        <AWS_ACCOUNT_ID>:secret:n8n/runners-auth-token" }
      ],
      "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
```

```
          "awslogs-group": "/ecs/n8n",
          "awslogs-region": "us-east-1",
          "awslogs-stream-prefix": "runners"
        }
      }
    }
  ]
}
```

Mirror `n8nio/runners:2.38.1` into ECR too, pinned to the same n8n version. `N8N_ENCRYPTION_KEY` must be **identical** everywhere this n8n deployment runs — store it once in Secrets Manager and reference it from every task definition. Pass `N8N_RUNNERS_AUTH_TOKEN` to the runners container directly: the runners image ignores `_FILE` variables, so a file-based indirection there silently fails broker authentication.

Register the task:

```
aws ecs register-task-definition --cli-input-json file://n8n-task-definition.json
```

---

7. Create the Worker Task Definition

Workers are what actually execute workflows in queue mode. Without them the queue fills and nothing runs. A worker task is the same image with `command` set to `worker`, no port mapping, no ALB registration — and **its own runners sidecar**, because each worker process needs a broker of its own.

`n8n-worker-task-definition.json`:

```
{
  "family": "n8n-worker-task",
  "requiresCompatibilities": ["FARGATE"],
  "networkMode": "awsvpc",
  "cpu": "1024",
  "memory": "2048",
  "executionRoleArn": "arn:aws:iam::<AWS_ACCOUNT_ID>:role/ecsTaskExecutionRole",
  "taskRoleArn": "arn:aws:iam::<AWS_ACCOUNT_ID>:role/n8nTaskRole",
  "containerDefinitions": [
    {
      "name": "n8n",
      "image": "<AWS_ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/n8n:2.38.1",
      "command": ["worker", "--concurrency=10"],
      "environment": [
        { "name": "DB_TYPE",                          "value": "postgresdb" },
        { "name": "DB_POSTGRESDB_HOST",               "value": "external-db.example.com" },
        { "name": "DB_POSTGRESDB_PORT",               "value": "5432" },
        { "name": "DB_POSTGRESDB_DATABASE",           "value": "n8n" },
        { "name": "DB_POSTGRESDB_USER",               "value": "n8nuser" },
        { "name": "DB_POSTGRESDB_SSL_ENABLED",        "value": "true" },
        { "name": "EXECUTIONS_MODE",                  "value": "queue" },
        { "name": "QUEUE_BULL_REDIS_HOST",            "value": "n8n-
        queue.abc123.0001.use1.cache.amazonaws.com" },
        { "name": "QUEUE_BULL_REDIS_PORT",            "value": "6379" },
        { "name": "N8N_DEFAULT_BINARY_DATA_MODE",     "value": "database" },
        { "name": "N8N_RUNNERS_MODE",                 "value": "external" },
        { "name": "N8N_RUNNERS_BROKER_LISTEN_ADDRESS", "value": "0.0.0.0" },
        { "name": "N8N_NATIVE_PYTHON_RUNNER",         "value": "true" },
        { "name": "N8N_GRACEFUL_SHUTDOWN_TIMEOUT",    "value": "60" }
      ],
      "secrets": [
        { "name": "N8N_ENCRYPTION_KEY",       "valueFrom": "arn:aws:secretsmanager:us-east-1:
        <AWS_ACCOUNT_ID>:secret:n8n/encryption-key" },
        { "name": "N8N_RUNNERS_AUTH_TOKEN",   "valueFrom": "arn:aws:secretsmanager:us-east-1:
        <AWS_ACCOUNT_ID>:secret:n8n/runners-auth-token" },
        { "name": "DB_POSTGRESDB_PASSWORD",   "valueFrom": "arn:aws:secretsmanager:us-east-1:
        <AWS_ACCOUNT_ID>:secret:n8n/db-password" }
      ],
      "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
          "awslogs-group": "/ecs/n8n",
          "awslogs-region": "us-east-1",
          "awslogs-stream-prefix": "worker"
        }
      }
    },
    {
      "name": "runners",
      "image": "<AWS_ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/n8n-runners:2.38.1",
      "environment": [
        { "name": "N8N_RUNNERS_TASK_BROKER_URI",       "value": "http://localhost:5679" },
        { "name": "N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT", "value": "15" }
      ],
      "secrets": [
        { "name": "N8N_RUNNERS_AUTH_TOKEN", "valueFrom": "arn:aws:secretsmanager:us-east-1:
        <AWS_ACCOUNT_ID>:secret:n8n/runners-auth-token" }
      ],
      "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
          "awslogs-group": "/ecs/n8n",
          "awslogs-region": "us-east-1",
          "awslogs-stream-prefix": "worker-runners"
        }
      }
```

```
    }
  ]
}
```

Register it:

```
aws ecs register-task-definition --cli-input-json file://n8n-worker-task-definition.json
```

The encryption key, database settings, and Redis endpoint are identical to the main task's. That is the point: main and workers are the same install, split by role.

---

8. Configure the Application Load Balancer

8.1 Create a Target Group

```
aws elbv2 create-target-group \
  --name n8n-tg \
  --protocol HTTP \
  --port 5678 \
  --vpc-id vpc-abcdefgh \
  --target-type ip \
  --health-check-protocol HTTP \
  --health-check-path /healthz/readiness \
  --health-check-interval-seconds 30 \
  --healthy-threshold-count 2
```

`/healthz/readiness` returns 200 only once the database is connected and migrated — a stronger signal than bare `/healthz`.

8.2 Create or Use an Existing ALB

```
aws elbv2 create-load-balancer \
  --name n8n-alb \
  --subnets subnet-11111111 subnet-22222222 \
  --security-groups sg-aaaa1111 \
  --scheme internet-facing
```

Set `N8N_HOST`, `WEBHOOK_URL`, and `N8N_EDITOR_BASE_URL` to this ALB's DNS name or custom domain — the editor's websockets and generated webhook URLs both depend on it matching what clients actually reach.

8.3 Create a Listener

```
# HTTP listener redirecting to HTTPS
aws elbv2 create-listener \
  --load-balancer-arn arn:aws:elasticloadbalancing:...:loadbalancer/app/n8n-alb/... \
  --protocol HTTP \
  --port 80 \
  --default-actions
        Type=redirect,RedirectConfig='{"Protocol":"HTTPS","Port":"443","StatusCode":"HTTP_301"}'

# HTTPS listener with ACM certificate
aws elbv2 create-listener \
  --load-balancer-arn arn:aws:elasticloadbalancing:...:loadbalancer/app/n8n-alb/... \
  --protocol HTTPS \
  --port 443 \
  --certificates CertificateArn=arn:aws:acm:...:certificate/... \
  --default-actions Type=forward,TargetGroupArn=arn:aws:elasticloadbalancing:...:targetgroup/n8n-
        tg/...
```

The ALB's HTTP/1.1 target group forwards `Upgrade` / `Connection` headers by default, which the editor's websocket traffic needs.

---

9. Deploy the ECS Services

9.1 The Main Service — Exactly One Task

```
aws ecs create-service \
  --cluster n8n-cluster \
  --service-name n8n-main \
  --task-definition n8n-task \
  --launch-type FARGATE \
  --platform-version LATEST \
  --desired-count 1 \
  --network-configuration "awsvpcConfiguration={subnets=[subnet-11111111,subnet-
        22222222],securityGroups=[sg-aaaa1111],assignPublicIp=ENABLED}" \
  --load-balancers "targetGroupArn=arn:aws:elasticloadbalancing:...:targetgroup/n8n-
        tg/...,containerName=n8n,containerPort=5678" \
  --deployment-configuration "maximumPercent=100,minimumHealthyPercent=0" \
  --deployment-controller type=ECS
```

`--desired-count 1` is deliberate. Community Edition supports one main process; running two would duplicate every cron and polling trigger. `maximumPercent=100` with `minimumHealthyPercent=0` makes ECS stop the old main task before starting the replacement, so two mains never overlap during a deployment — at the cost of a short gap in editor and webhook availability.

9.2 The Worker Service — Scale This One

```
aws ecs create-service \
  --cluster n8n-cluster \
  --service-name n8n-workers \
  --task-definition n8n-worker-task \
  --launch-type FARGATE \
  --platform-version LATEST \
  --desired-count 2 \
  --network-configuration "awsvpcConfiguration={subnets=[subnet-11111111,subnet-
        22222222],securityGroups=[sg-aaaa1111],assignPublicIp=ENABLED}" \
  --deployment-controller type=ECS
```

No load balancer and no target group: workers pull from Redis, they do not receive HTTP traffic.

Verify both:

```
aws ecs describe-services --cluster n8n-cluster --services n8n-main n8n-workers
aws ecs list-tasks --cluster n8n-cluster --service-name n8n-workers
```

One main, N workers, one Postgres, one Redis: `EXECUTIONS_MODE=queue` is what lets the main hand executions to workers instead of running them itself.

---

10. Enable Service Auto Scaling for Workers

Auto scaling applies to the **worker** service only. Never register the main service as a scalable target.

10.1 Create a Scalable Target

```
aws application-autoscaling register-scalable-target \
  --service-namespace ecs \
  --resource-id service/n8n-cluster/n8n-workers \
  --scalable-dimension ecs:service:DesiredCount \
  --min-capacity 2 \
  --max-capacity 10
```

10.2 Create a Scaling Policy

```
aws application-autoscaling put-scaling-policy \
  --policy-name worker-cpu-autoscale \
  --service-namespace ecs \
  --resource-id service/n8n-cluster/n8n-workers \
  --scalable-dimension ecs:service:DesiredCount \
  --policy-type TargetTrackingScaling \
  --target-tracking-scaling-policy-configuration file://scaling-policy.json
```

`scaling-policy.json`:

```
{
  "TargetValue": 60.0,
  "PredefinedMetricSpecification": {
    "PredefinedMetricType": "ECSServiceAverageCPUUtilization"
  },
  "ScaleInCooldown": 60,
  "ScaleOutCooldown": 60
}
```

---

11. Configure CloudWatch Logs & Metrics

Every container in both task definitions has a `logConfiguration` block, with distinct stream prefixes so main, worker, and runner output stay separable. Then:

- View logs:

```
aws logs describe-log-streams --log-group-name /ecs/n8n
aws logs get-log-events --log-group-name /ecs/n8n --log-stream-name <stream>
```

- Create **CloudWatch Alarms** on metrics (e.g., CPU > 80% for 5 min) to trigger scaling or SNS alerts.

---

12. Manage Secrets with AWS Secrets Manager

Store each secret once, and reference it from both task definitions:

```
aws secretsmanager create-secret \
  --name n8n/encryption-key \
  --secret-string "$(openssl rand -hex 32)"

aws secretsmanager create-secret \
  --name n8n/runners-auth-token \
  --secret-string "$(openssl rand -hex 32)"

aws secretsmanager create-secret \
  --name n8n/db-password \
  --secret-string "SuperSecretExternal"
```

These are the ARNs referenced in Sections 6 and 7. Changing a secret's value in place doesn't need a new task definition — just a new deployment on each service to pick it up:

```
aws ecs update-service --cluster n8n-cluster --service n8n-main --force-new-deployment
aws ecs update-service --cluster n8n-cluster --service n8n-workers --force-new-deployment
```

---

13. Troubleshooting Common Issues

Issue	Cause	Solution
Task stuck in `PENDING`	Subnets lack ENI capacity or SG misconfiguration	Ensure free IPs, SG allows 443 and 80
`CannotPullContainerError`	ECR permissions or tag mismatch	Verify execution role has `ecr:GetAuthorizationToken`, correct image URI
Health checks failing on ALB	Container not responding on `/healthz/readiness`	Confirm Postgres is reachable, increase timeout and thresholds
Fargate tasks repeatedly restarting	Missing env vars or container crash	Check CloudWatch logs; ensure DB, Redis, and encryption key vars are provided
Executions queue up and never run	No worker service, or workers can't reach Redis	Confirm `n8n-workers` has running tasks and the task SG allows outbound 6379 to the cache SG
Duplicate scheduled executions	Main service scaled above 1 task	Set `--desired-count 1` on `n8n-main`; multi-main is Enterprise-only
`n8n` starts but Code nodes fail	`runners` sidecar unreachable or auth token mismatch	Confirm both containers share `N8N_RUNNERS_AUTH_TOKEN`, passed directly and not via a `_FILE` variable
Binary data missing on a worker	Task restarted with filesystem mode	Use `database` or `s3`; filesystem mode is unsupported in queue mode
Secrets not injected	Incorrect ARN or IAM permissions	Verify secret ARN and that `secretsmanager:GetSecretValue` is allowed for the execution role

Further reading

- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/enable-queue-mode
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/queue-mode
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/handle-binary-data
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners

---

Chapter 23: Google Cloud Run

Overview

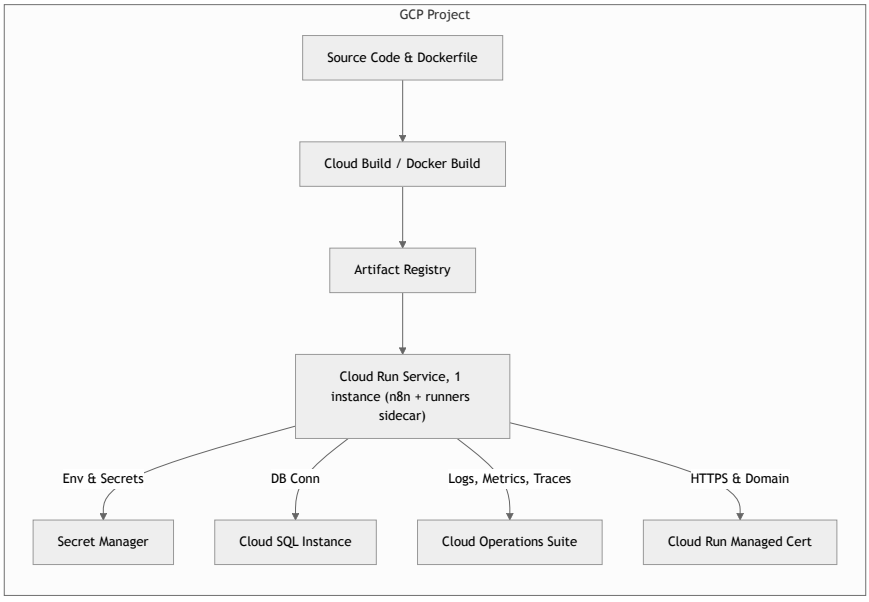
Google Cloud Run is a managed platform for containers, with per-revision traffic splitting, built-in HTTPS, and integration with Cloud SQL, Secret Manager, and IAM.

Cloud Run's request-driven, scale-to-zero model does not match how n8n runs. n8n needs a process that stays alive between requests to fire cron and polling triggers, a long-lived editor websocket, and — in queue mode — worker processes that are never the target of an HTTP request at all. Cloud Run gives you none of those for free.

This chapter therefore runs n8n on Cloud Run as a single always-on instance in regular (non-queue) mode: `--min-instances 1 --max-instances 1` with CPU always allocated, plus the task runners sidecar. That is a real, working deployment for a small instance, and it is the only safe shape here — a second instance on the same database would duplicate every schedule and poll. **If you need queue mode, workers, and horizontal scale, deploy on GKE or ECS instead**, following the Kubernetes: Scaled Queue Mode or AWS ECS Fargate chapters. Cloud Run is a fit for small always-on instances only.

In this chapter you will learn to:

1. **Containerize n8n** and push it to Artifact Registry
2. **Provision Cloud SQL** and configure Proxy connectivity
3. **Create a service account** with IAM roles for Cloud SQL and Secret Manager
4. **Store secrets** in Secret Manager
5. **Deploy a Cloud Run service** pinned to one always-on instance, with the runners sidecar
6. **Understand why queue mode does not belong here**
7. **Set up a custom domain and HTTPS**
8. **Configure logging, tracing, and alerts**, including native OpenTelemetry
9. **Perform traffic splitting** for canary deployments
10. **Run the task runners sidecar** correctly on this platform
11. **Troubleshoot** cold starts, connection timeouts, and scaling limits



Diagram

Prerequisites

- **Google Cloud SDK (`gcloud`)** installed and authenticated
- **GCP Project** with billing enabled
- **APIs enabled**: Cloud Run, Artifact Registry, Cloud SQL Admin, Secret Manager, Compute Engine, IAM
- **IAM permissions**: Owner or Editor, plus Service Account Admin and Secret Manager Admin

---

1. Containerize n8n and Push to Artifact Registry

1.1 Create an Artifact Registry Repository

```
gcloud artifacts repositories create n8n-repo \
  --repository-format=docker \
  --location=us-central1 \
  --description="n8n container registry"
```

1.2 Authenticate Docker

```
gcloud auth configure-docker us-central1-docker.pkg.dev
```

1.3 Build and Tag

`FROM n8nio/n8n:2.38.1` in your `Dockerfile`, and tag with the same version:

```
docker build -t us-central1-docker.pkg.dev/PROJECT_ID/n8n-repo/n8n:2.38.1 .
```

1.4 Push the Image

```
docker push us-central1-docker.pkg.dev/PROJECT_ID/n8n-repo/n8n:2.38.1
```

> **Scenario:** the CI/CD and GitOps chapter builds a `cloudbuild.yaml` around bumping this pinned tag.

---

2. Provision Cloud SQL for PostgreSQL

2.1 Create Cloud SQL Instance

```
gcloud sql instances create n8n-sql \
  --database-version=POSTGRES_18 \
  --tier=db-custom-2-4096 \
  --region=us-central1
```

2.2 Set Password and Database

```
gcloud sql users set-password postgres --instance=n8n-sql --password="SuperSecretPg"
gcloud sql databases create n8n --instance=n8n-sql
```

2.3 Enable Private IP (Optional)

If you prefer to reach Cloud SQL over a private IP instead of the Auth proxy, create a VPC connector and attach the instance to the network:

```
gcloud compute networks vpc-access connectors create n8n-connector \
  --network=default --region=us-central1 --range=10.8.0.0/28

gcloud sql instances patch n8n-sql --network=default
```

The deploy command in Section 5 uses the Auth proxy (`--add-cloudsql-instances`), so the connector is optional.

---

3. Create a Service Account and Grant IAM Roles

3.1 Create the Service Account

```
gcloud iam service-accounts create n8n-run-sa \
  --display-name="n8n Cloud Run service account"
```

3.2 Assign Roles

```
gcloud projects add-iam-policy-binding PROJECT_ID \
  --member="serviceAccount:n8n-run-sa@PROJECT_ID.iam.gserviceaccount.com" \
  --role="roles/cloudsql.client"

gcloud projects add-iam-policy-binding PROJECT_ID \
  --member="serviceAccount:n8n-run-sa@PROJECT_ID.iam.gserviceaccount.com" \
  --role="roles/secretmanager.secretAccessor"

gcloud projects add-iam-policy-binding PROJECT_ID \
  --member="serviceAccount:n8n-run-sa@PROJECT_ID.iam.gserviceaccount.com" \
  --role="roles/logging.logWriter"
```

---

4. Store Sensitive Data in Secret Manager

```
openssl rand -hex 32 | gcloud secrets create n8n-encryption-key --data-file=-
openssl rand -hex 32 | gcloud secrets create n8n-runners-auth-token --data-file=-
echo -n "SuperSecretPg" | gcloud secrets create n8n-db-pass --data-file=-
```

`n8n-encryption-key` must hold the same value as everywhere else this deployment runs — copy it in, don't regenerate it here.

---

5. Deploy to Cloud Run

The deploy command below uses two containers: the `n8n` ingress container and its `runners` sidecar. Containers in one Cloud Run service share a network namespace, so the sidecar reaches the broker on `localhost:5679`.

```
gcloud run deploy n8n-service \
  --region us-central1 \
  --platform managed \
  --allow-unauthenticated \
  --service-account n8n-run-sa@PROJECT_ID.iam.gserviceaccount.com \
  --container n8n \
  --image us-central1-docker.pkg.dev/PROJECT_ID/n8n-repo/n8n:2.38.1 \
  --port 5678 \
  --set-env-vars DB_TYPE=postgresdb,DB_POSTGRESDB_HOST=/cloudsql/PROJECT_ID:us-central1:n8n-
        sql,DB_POSTGRESDB_DATABASE=n8n,DB_POSTGRESDB_USER=postgres,N8N_HOST=n8n.example.com,N8N_PROTOC
        OL=https,WEBHOOK_URL=https://n8n.example.com/,N8N_EDITOR_BASE_URL=https://n8n.example.com/,N8N
        _PROXY_HOPS=1,N8N_DEFAULT_BINARY_DATA_MODE=database,N8N_RUNNERS_MODE=external,N8N_RUNNERS_BROK
        ER_LISTEN_ADDRESS=0.0.0.0,N8N_NATIVE_PYTHON_RUNNER=true \
  --set-secrets N8N_ENCRYPTION_KEY=n8n-encryption-key:latest,N8N_RUNNERS_AUTH_TOKEN=n8n-runners-auth-
        token:latest,DB_POSTGRESDB_PASSWORD=n8n-db-pass:latest \
  --container runners \
  --image us-central1-docker.pkg.dev/PROJECT_ID/n8n-repo/n8n-runners:2.38.1 \
  --set-env-vars
        N8N_RUNNERS_TASK_BROKER_URI=http://localhost:5679,N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT=15 \
  --set-secrets N8N_RUNNERS_AUTH_TOKEN=n8n-runners-auth-token:latest \
  --add-cloudsql-instances PROJECT_ID:us-central1:n8n-sql \
  --memory 2Gi --cpu 2 --concurrency 40 \
  --min-instances 1 --max-instances 1 \
  --no-cpu-throttling \
  --timeout 3600
```

Four flags matter more here than anywhere else in this book:

- **`--min-instances 1 --max-instances 1`**: Cloud Run scales to zero by default, and a stopped process fires no cron or polling trigger, so the floor must be 1. The ceiling must *also* be 1: a second instance against the same database is a second main process, and both would fire every schedule and poll independently. Community Edition runs one main. This gives up both scale-to-zero savings and horizontal scale — the tradeoff for running n8n here at all.
- **`--no-cpu-throttling`**: without it, Cloud Run only gives the container CPU while a request is in flight, and n8n's scheduler stops between requests. Timers and polling triggers need CPU allocated for the life of the instance.
- **`--timeout 3600`**: the default is 300s, the ceiling is 3600s (60 minutes). Any workflow invoked synchronously over HTTP that runs longer than that is cut off mid-flight. On a single-instance Cloud Run deployment there is no queue to absorb long executions, so this ceiling is a hard limit on synchronous workflow duration.
- No `--use-http2` flag: leaving it unset keeps Cloud Run speaking HTTP/1.1 end-to-end. The editor's websocket upgrade doesn't survive Cloud Run's HTTP/2-to-gRPC path.

Session affinity is not needed and not set: with one instance there is nothing to route between.

`--add-cloudsql-instances` runs the Cloud SQL Auth proxy alongside your containers; `--set-secrets` injects Secret Manager values as environment variables. The `:latest` suffix in `--set-secrets` is Secret Manager's version alias, not a container tag — the images above stay pinned to `2.38.1`. `N8N_DEFAULT_BINARY_DATA_MODE=database` keeps binary data off the instance's ephemeral disk, which Cloud Run discards on every revision and restart; use `s3` with the `N8N_EXTERNAL_STORAGE_S3_*` variables if payloads are large.

---

6. Why Queue Mode Does Not Belong on Cloud Run

Queue mode needs `n8n worker` processes: long-lived containers that receive no HTTP requests and pull jobs from Redis instead. Cloud Run only starts and keeps containers alive in response to request traffic, so a worker deployed as a Cloud Run service either never starts or is torn down between requests. There is no supported way to run an n8n worker fleet here.

That is why this chapter sets no `EXECUTIONS_MODE`, no `QUEUE_BULL_REDIS_HOST`, and no Memorystore instance. The service runs in regular mode, executing workflows in the main process.

When you outgrow one instance — sustained concurrent executions, long-running workflows, or Code-node-heavy load — move to queue mode on GKE (see Kubernetes: Scaled Queue Mode) or ECS (see AWS ECS Fargate). Do not attempt to bridge the two by raising `--max-instances`.

---

7. Custom Domain & HTTPS

Cloud Run services get a `.run.app` domain with HTTPS by default. To use your own domain:

```
gcloud run domain-mappings create \
  --service n8n-service \
  --domain n8n.example.com \
  --region us-central1
```

Add the DNS records the command output gives you, and set `N8N_HOST`, `WEBHOOK_URL`, and `N8N_EDITOR_BASE_URL` to this domain, not `.run.app` — they must match what callers actually use.

---

8. Configure Logging, Monitoring, and Tracing

Cloud Run auto-ships container logs to **Cloud Logging**:

```
gcloud logging read "resource.type=cloud_run_revision AND resource.labels.service_name=n8n-service" --
        limit 50 --format json
```

For metrics, set `N8N_METRICS=true` and let an external Prometheus with VPC egress pull `/metrics` from the service. For tracing, use n8n's native OpenTelemetry support rather than a hand-rolled collector sidecar:

```
--set-env-vars N8N_OTEL_ENABLED=true,N8N_OTEL_EXPORTER_OTLP_ENDPOINT=https://your-collector:4318
```

(or configure it from **Settings** → **OpenTelemetry**, available since n8n 2.27). This emits a `workflow.execute` span per execution with nested `node.execute` spans, and propagates the W3C `traceparent` header on inbound webhooks and outbound requests, so a trace started by a caller continues through the workflow. Point the endpoint at Cloud Trace's OTLP ingestion or your own collector.

Set up alerts on error-count and latency thresholds.

---

9. Traffic Splitting for Safe Deployments

```
gcloud run services update-traffic n8n-service \
  --to-revisions new=10,previous=90 \
  --region us-central1
```

Monitor the new revision before shifting traffic to it fully. With `--max-instances 1`, a traffic split briefly runs two revisions — and therefore two main processes — at once. Keep splits short, and prefer a straight replacement for any deployment where duplicated schedule firing would matter.

---

10. Task Runners on Cloud Run

Task runners execute every Code node since n8n 2.0, and the supported production mode is `external`: the separate `n8nio/runners` image, pinned to the same version as n8n. The deploy command in Section 5 already includes it as the `runners` container, sharing the service's network namespace and reaching the broker at `N8N_RUNNERS_TASK_BROKER_URI=http://localhost:5679`.

Two details keep it healthy on Cloud Run:

- **`--no-cpu-throttling`** (Section 5) also matters here. A throttled sidecar cannot keep its broker connection alive between requests.
- **`N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT=15`** lets an idle runner exit; Cloud Run restarts it with the instance. Do not set `_FILE` variables on the runners container — the runners image ignores them, and the broker handshake fails with no useful error.

Pass `N8N_RUNNERS_AUTH_TOKEN` to both containers from the same Secret Manager secret. A mismatch shows up as Code nodes failing while the service looks healthy.

---

11. Troubleshooting Common Issues

Issue	Cause	Solution
`permission denied` on deploy	Service account lacks IAM role	Grant `roles/cloudsql.client` and `roles/secretmanager.secretAccessor`
Cannot connect to Cloud SQL (`dial tcp`)	`--add-cloudsql-instances` omitted, or the socket path is wrong	Include `--add-cloudsql-instances` and set `DB_POSTGRESDB_HOST=/cloudsql/<connection-name>`
Editor stuck "Reconnecting"	HTTP/2 end-to-end enabled	Don't pass `--use-http2`; Cloud Run must speak HTTP/1.1 for the websocket upgrade
Cron/polling triggers silently stop firing	Service scaled to zero, or CPU throttled between requests	Set `--min-instances 1` and `--no-cpu-throttling`
Duplicate scheduled executions	More than one instance or two live revisions	Set `--max-instances 1`; finish traffic splits promptly
Long executions cut off partway through	Cloud Run request timeout	Raise `--timeout` toward the 3600s ceiling, or move to queue mode on GKE or ECS
Code nodes fail while the service is healthy	Runners auth token mismatch, or a `_FILE` variable on the runners container	Give both containers the same `N8N_RUNNERS_AUTH_TOKEN` value directly
Binary data disappears after a revision	Ephemeral instance disk	Use `N8N_DEFAULT_BINARY_DATA_MODE=database` or `s3`

Further reading

- https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/enable-queue-mode
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/handle-binary-data
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/deployment

---

Chapter 24: Azure Container Instances

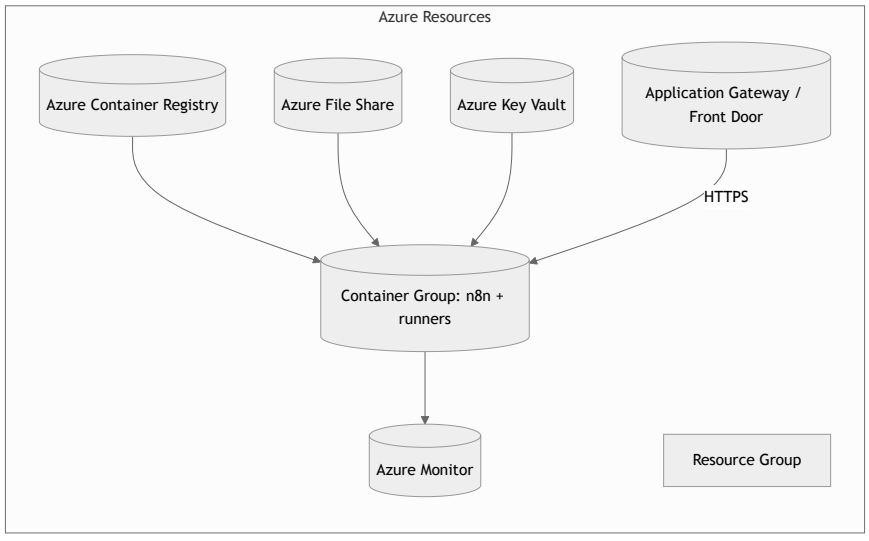
Overview

Azure Container Instances (ACI) run containers in Azure without managing VMs or a Kubernetes cluster. A container group is a single unit: it does not autoscale, and there is no way to run a second group of the same n8n install safely — two groups against one database would each fire every schedule and poll.

This chapter therefore deploys **one container group in regular (non-queue) mode**: the `n8n` container plus its runners sidecar, an Azure File Share for `~/.n8n`, and an Application Gateway or Front Door in front for HTTPS. Queue mode with `n8n worker` processes is out of scope here; for that, use AKS (see Kubernetes: Scaled Queue Mode) or AWS ECS Fargate.

In this chapter, you will:

1. **Create a resource group** and Azure Container Registry (ACR)
2. **Build and push** your n8n Docker image to ACR
3. **Provision an Azure File Share** for persistent storage
4. **Put an HTTPS gateway in front** before publishing anything
5. **Deploy n8n and its runners sidecar** as two containers in one container group
6. **Configure the encryption key and DB credentials** via Key Vault
7. **Monitor logs and metrics** via Azure Monitor
8. **Understand the scaling ceiling** of a single container group
9. **Automate deployments** with ARM/Bicep templates



Diagram

Prerequisites

- **Azure CLI** installed and logged in (`az login`)
- **Docker** installed locally
- **Resource Group** created or permission to create one
- A **PostgreSQL database** reachable from the container group (Azure Database for PostgreSQL, or your own)
- A **VNet and subnet** the container group and its gateway can share

- The **encryption key**, **runners auth token**, and database credentials you generated for the reference stack

---

1. Create Resource Group and Azure Container Registry

1.1 Resource Group

```
az group create \
  --name rg-n8n-aci \
  --location eastus
```

1.2 Container Registry

```
az acr create \
  --resource-group rg-n8n-aci \
  --name acrn8n \
  --sku Standard \
  --admin-enabled true
```

---

2. Build and Push n8n Image to ACR

2.1 Login to ACR

```
az acr login --name acrn8n
```

2.2 Build and Tag

`FROM n8nio/n8n:2.38.1` in your `Dockerfile`:

```
ACR_LOGIN_SERVER=$(az acr show --name acrn8n --query loginServer --output tsv)
docker build -t $ACR_LOGIN_SERVER/n8n:2.38.1 .
```

Mirror `n8nio/runners:2.38.1` the same way:

```
docker pull n8nio/runners:2.38.1
docker tag n8nio/runners:2.38.1 $ACR_LOGIN_SERVER/n8n-runners:2.38.1
docker push $ACR_LOGIN_SERVER/n8n-runners:2.38.1
```

2.3 Push the n8n Image

```
docker push $ACR_LOGIN_SERVER/n8n:2.38.1
```

> **Scenario:** automate this with Azure Pipelines or GitHub Actions, bumping the pinned tag on release.

---

3. Provision Azure File Share for Persistence

3.1 Storage Account

```
az storage account create \
  --resource-group rg-n8n-aci \
  --name stn8naci \
  --sku Standard_LRS \
  --kind StorageV2
```

3.2 File Share

```
STORAGE_KEY=$(az storage account keys list \
  --resource-group rg-n8n-aci \
  --account-name stn8naci \
  --query "[0].value" -o tsv)

az storage share create \
  --name n8n-data \
  --account-name stn8naci \
  --account-key $STORAGE_KEY \
  --quota 5
```

This deployment runs **one** container group, so exactly one process writes to the share. That, and only that, is why `N8N_DEFAULT_BINARY_DATA_MODE=filesystem` is safe here: the share being a network file system has nothing to do with it. Azure Files does not make `filesystem` mode safe for concurrent writers, and a second container group is not a supported way to scale this deployment (Section 8). If you expect large binary payloads, set `N8N_DEFAULT_BINARY_DATA_MODE=s3` with the `N8N_EXTERNAL_STORAGE_S3_*` variables instead.

---

4. Put the HTTPS Gateway in Front First

ACI itself terminates nothing: a container group with a public IP serves plain HTTP on whatever port you publish. Create the front end **before** you deploy the group, so the URL variables in Section 5 have a real hostname to point at.

Create an **Application Gateway** or **Azure Front Door**, give it a managed certificate for your domain, and point its backend pool at the container group's FQDN and port 5678. Then keep the container group off the public internet: deploy it into a VNet (`--vnet` / `--subnet` on `az container create`, or the `networkProfile` property in YAML) with a private IP, so the gateway is the only route in.

Everything downstream depends on this ordering. `N8N_PROTOCOL=https`, `WEBHOOK_URL`, and `N8N_EDITOR_BASE_URL` must name the **gateway's** hostname, not the raw `*.azurecontainer.io` FQDN; the editor's websockets and every generated webhook URL are built from those values.

---

5. Deploy n8n and Runners to Azure Container Instances

ACI containers in the same group share localhost networking, so `runners` reaches n8n's broker directly — the same pattern as the ECS task. Use a YAML container-group definition rather than a single long `az container create` command, since there are now two containers:

`n8n-aci.yaml`:

```yaml
apiVersion: 2021-09-01
location: eastus
name: n8n-aci
properties:
  osType: Linux
  restartPolicy: Always
  # Private IP inside the VNet the gateway sits in. The container group is
  # never published directly; Application Gateway or Front Door is the only
  # public entry point.
  networkProfile:
    id: /subscriptions/<SUB_ID>/resourceGroups/rg-n8n-
        aci/providers/Microsoft.Network/networkProfiles/n8n-netprofile
  ipAddress:
    type: Private
    ports:
      - port: 5678
        protocol: TCP
  containers:
    - name: n8n
      properties:
        image: acrn8n.azurecr.io/n8n:2.38.1
        ports:
          - port: 5678
        environmentVariables:
          - name: DB_TYPE
            value: postgresdb
          - name: DB_POSTGRESDB_HOST
            value: external-db.example.com
          - name: DB_POSTGRESDB_DATABASE
            value: n8n
          - name: DB_POSTGRESDB_USER
            value: n8nuser
          - name: DB_POSTGRESDB_SSL_ENABLED
            value: "true"
          # The gateway hostname from Section 4, never the raw ACI FQDN.
          - name: N8N_HOST
            value: n8n.example.com
          - name: N8N_PROTOCOL
            value: https
          - name: WEBHOOK_URL
            value: https://n8n.example.com/
          - name: N8N_EDITOR_BASE_URL
            value: https://n8n.example.com/
          - name: N8N_PROXY_HOPS
            value: "1"
          - name: N8N_DEFAULT_BINARY_DATA_MODE
            value: filesystem
          - name: N8N_RUNNERS_MODE
            value: external
          - name: N8N_RUNNERS_BROKER_LISTEN_ADDRESS
            value: 0.0.0.0
          - name: N8N_NATIVE_PYTHON_RUNNER
            value: "true"
        secureEnvironmentVariables:
          - name: N8N_ENCRYPTION_KEY
            secureValue: replace-with-64-hex-characters
          - name: N8N_RUNNERS_AUTH_TOKEN
            secureValue: replace-with-a-long-random-string
          - name: DB_POSTGRESDB_PASSWORD
            secureValue: SuperSecretExternal
        volumeMounts:
          - name: n8n-data
            mountPath: /home/node/.n8n
        resources:
          requests:
```

```
          cpu: 1
          memoryInGB: 2
    - name: runners
      properties:
        image: acrn8n.azurecr.io/n8n-runners:2.38.1
        environmentVariables:
          - name: N8N_RUNNERS_TASK_BROKER_URI
            value: http://localhost:5679
          - name: N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT
            value: "15"
        secureEnvironmentVariables:
          - name: N8N_RUNNERS_AUTH_TOKEN
            secureValue: replace-with-a-long-random-string
        resources:
          requests:
            cpu: 1
            memoryInGB: 1
  volumes:
    - name: n8n-data
      azureFile:
        shareName: n8n-data
        storageAccountName: stn8naci
        storageAccountKey: <STORAGE_KEY>
```

`secureEnvironmentVariables` keeps these values out of `az container show` output and ACI's plaintext logs — use it for the encryption key, runners auth token, and DB password. Note that the runners container gets `N8N_RUNNERS_AUTH_TOKEN` as a plain value: the runners image ignores `_FILE`-suffixed variables, so file-based indirection there does not work.

Deploy:

```
az container create --resource-group rg-n8n-aci --file n8n-aci.yaml
```

Then add the group's private IP to the gateway's backend pool and browse to `https://n8n.example.com`.

5.1 A Test Deployment Without a Gateway

If you only want to try ACI out and have no gateway yet, you can publish the group directly — but then the connection is plain HTTP, and n8n refuses to set its session cookie over HTTP on a non-localhost address unless you tell it to. Replace the network block with:

```
  ipAddress:
    type: Public
    dnsNameLabel: n8n-aci-app
    ports:
      - port: 5678
        protocol: TCP
```

and the URL variables with:

```
          - name: N8N_HOST
            value: n8n-aci-app.eastus.azurecontainer.io
          - name: N8N_PROTOCOL
            value: http
          - name: WEBHOOK_URL
            value: http://n8n-aci-app.eastus.azurecontainer.io:5678/
          - name: N8N_EDITOR_BASE_URL
            value: http://n8n-aci-app.eastus.azurecontainer.io:5678/
          - name: N8N_SECURE_COOKIE
            value: "false"
```

Drop `N8N_PROXY_HOPS`, since nothing is proxying. **This is not a production deployment.** Session cookies and every credential you enter travel unencrypted over the public internet. Use it to look around, then tear it down and rebuild behind the gateway.

---

6. Integrate with Azure Key Vault

Rather than embedding secrets in the YAML in source control, pull them from Key Vault at deploy time.

6.1 Create Key Vault and Secrets

```
az keyvault create \
  --name kv-n8n \
  --resource-group rg-n8n-aci \
  --location eastus

az keyvault secret set --vault-name kv-n8n --name "n8n-encryption-key" --value "$(openssl rand -hex
      32)"
az keyvault secret set --vault-name kv-n8n --name "n8n-runners-auth-token" --value "$(openssl rand -
      hex 32)"
az keyvault secret set --vault-name kv-n8n --name "n8n-db-password" --value "SuperSecretExternal"
```

6.2 Enable Managed Identity

```
az container identity assign \
  --resource-group rg-n8n-aci \
  --name n8n-aci
```

6.3 Grant Access Policy

```
MI_PRINCIPAL_ID=$(az container show \
  --resource-group rg-n8n-aci \
  --name n8n-aci \
  --query identity.principalId -o tsv)

az keyvault set-policy \
  --name kv-n8n \
  --object-id $MI_PRINCIPAL_ID \
  --secret-permissions get list
```

6.4 Reference Vault Secrets in ACI

ACI can't pull Key Vault references directly into `secureEnvironmentVariables`; resolve them in your deploy pipeline (`az keyvault secret show --query value` into the YAML before `az container create`), or fetch them via managed identity in an init step before n8n starts. Either way the resolved value still lands in `secureEnvironmentVariables`, never a plain one.

---

7. Monitor Logs and Metrics

View container logs:

```
az container logs --resource-group rg-n8n-aci --name n8n-aci --container-name n8n
```

Send both containers' logs to a Log Analytics workspace for retention and querying:

```
az monitor log-analytics query \
  --workspace <workspace-id> \
  --analytics-query "ContainerInstanceLog_CL | where ContainerGroup_s == 'n8n-aci'"
```

Configure **alerts** on container CPU, memory, or restart count in Azure Monitor.

---

8. The Scaling Ceiling

ACI does not autoscale within a container group, and **a second container group is not a supported way to scale this deployment**. Two groups against the same database are two main processes: every cron and polling trigger fires twice, and the two disagree about which executions are running. Running more than one n8n instance requires queue mode — Redis, `n8n worker` processes, one runners sidecar per worker, and binary data in the database or S3 — which a container group cannot provide, because ACI has no way to run worker processes as a scalable pool behind one queue.

That leaves two honest options:

- **Scale up**: raise the `cpu` and `memoryInGB` requests on the `n8n` container and restart the group. This is the whole of ACI's scaling story for n8n.
- **Move platforms**: for queue mode and horizontal scale, deploy on **AKS** (see Kubernetes: Scaled Queue Mode and Deploying with Helm) or on AWS ECS (see AWS ECS Fargate). AKS Virtual Nodes can still schedule burst pods onto ACI capacity, but the control plane and the worker pool live in Kubernetes.

---

9. Automate with ARM or Bicep

The template below is abbreviated to show structure — the full environment variable block, `secureEnvironmentVariables`, and volume mounts from Section 5 all belong in it. Note `type: 'Private'` on `ipAddress`: as in Section 4, the gateway is the public entry point, not the container group.

```
resource containerGroup 'Microsoft.ContainerInstance/containerGroups@2021-09-01' = {
  name: 'n8n-aci'
  location: resourceGroup().location
  properties: {
    osType: 'Linux'
    containers: [
      {
        name: 'n8n'
        properties: {
          image: '${acrLoginServer}/n8n:2.38.1'
          resources: { requests: { cpu: 1, memoryInGB: 2 } }
          environmentVariables: [
            { name: 'N8N_HOST', value: 'n8n.example.com' }
          ]
          volumeMounts: [
            { name: 'n8n-data', mountPath: '/home/node/.n8n' }
          ]
        }
      }
      {
        name: 'runners'
        properties: {
          image: '${acrLoginServer}/n8n-runners:2.38.1'
          resources: { requests: { cpu: 1, memoryInGB: 1 } }
        }
      }
    ]
    ipAddress: {
      type: 'Private'
      ports: [{ port: 5678, protocol: 'TCP' }]
    }
    volumes: [
      {
        name: 'n8n-data'
        azureFile: {
          shareName: 'n8n-data'
          storageAccountName: 'stn8naci'
          storageAccountKey: storageKey
        }
      }
    ]
  }
}
```

Deploy:

```
az deployment group create \
  --resource-group rg-n8n-aci \
  --template-file main.bicep
```

---

10. Troubleshooting Common Issues

Issue	Cause	Solution
ACI fails to pull image	ACR credentials incorrect or server name mismatch	Verify `az acr login`, check registry credentials
Volume mount error	File Share or storage key incorrect	Confirm the share exists, re-fetch the key, check share name case
Container restart loop	Missing env vars or unreachable DB	Check `az container logs`, verify DB credentials and the runners token
`runners` healthy but Code nodes fail	Auth token mismatch between containers	Confirm `N8N_RUNNERS_AUTH_TOKEN` is identical in both, and passed directly rather than via a `_FILE` variable
Editor rejects login, “secure cookie” error	Reaching n8n over plain HTTP on a non-localhost address	Put the gateway in front (Section 4); only a throwaway test deployment should set `N8N_SECURE_COOKIE=false`
Webhook URLs point at `*.azurecontainer.io`	URL variables set to the raw ACI FQDN	Set `N8N_HOST`, `WEBHOOK_URL`, and `N8N_EDITOR_BASE_URL` to the gateway hostname
Duplicate scheduled executions	A second container group running the same install	Run one group; use AKS or ECS with queue mode to scale out
HTTPS integration issues	Front Door or Application Gateway misconfigured	Verify the backend pool includes the container group’s address and port 5678
Key Vault access denied	Managed identity not granted `get` permission	Confirm the access policy includes the container’s principal ID

Further reading

- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/deployment
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/handle-binary-data
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/enable-queue-mode

---

Chapter 25: HashiCorp Nomad

Overview

Nomad is a cluster orchestrator that schedules containers and other workloads from declarative HCL job files. It runs as a single binary on servers and clients, supports Docker and other task drivers, integrates with Consul for service discovery and with Vault for secrets, and performs rolling updates on job resubmission.

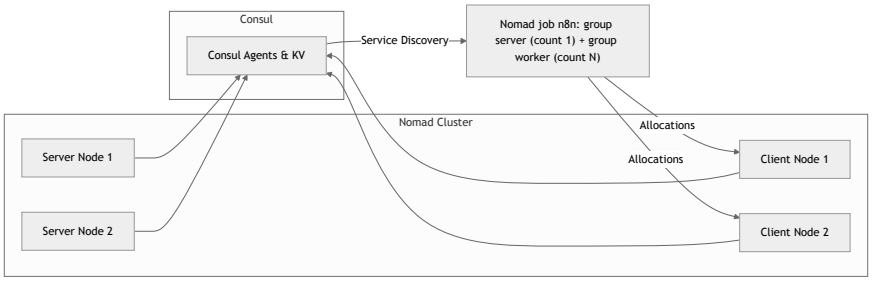
This chapter runs n8n in **queue mode**, which is what makes n8n scalable on Nomad. The job has two task groups:

- a **`server` group at `count = 1`** — the n8n main process, serving the editor and webhooks, paired with a runners task
- a **`worker` group** whose `count` you scale — each allocation runs `n8n worker` paired with its own runners task

Redis holds the queue, Postgres holds the data, and `N8N_DEFAULT_BINARY_DATA_MODE=database` keeps binary data off local disks. The `server` group stays at 1 because Community Edition runs one main process; multi-main is an Enterprise feature. Scaling means raising the `worker` group's count.

In this chapter you will learn to:

1. **Install and configure** a Nomad cluster (server and client nodes)
2. **Integrate with Consul** for service discovery and health checks
3. **Write a Nomad job specification** with a single-count `server` group and a scalable `worker` group, each task paired with its runners task
4. **Manage secrets** via Vault integration
5. **Configure volumes** for persistence, and why binary data goes to the database
6. **Deploy the job** and monitor allocations, logs, and health
7. **Perform rolling updates** by submitting new job versions
8. **Scale horizontally** by adjusting the worker group's count
9. **Use nomad-autoscaler** for dynamic scaling of the worker group
10. **Troubleshoot** allocation failures, network issues, and driver errors



Diagram

Prerequisites

- **Three or more machines** (servers and clients) with Linux
- **Nomad v1.5+** installed on each node
- **Consul v1.14+** installed for service discovery (optional but recommended)
- **Docker** installed on client nodes (for the Docker driver)
- **Vault** (optional) for secret injection

- **Basic network connectivity** among servers, clients, and Consul
- **PostgreSQL** and **Redis**, reachable from the client nodes (registered in Consul as `postgres` and `redis` below)
- The **encryption key** and **runners auth token** from the reference stack, stored in Vault

---

1. Install and Configure Nomad & Consul

1.1 Consul Setup

On each node:

```
wget https://releases.hashicorp.com/consul/1.14.0/consul_1.14.0_linux_amd64.zip
unzip consul_1.14.0_linux_amd64.zip && sudo mv consul /usr/local/bin/
```

Create `consul.hcl`:

```
datacenter = "dc1"
server = true         # on server nodes; false on clients
bootstrap_expect = 3  # number of servers
bind_addr = "{{ GetInterfaceIP \"eth0\" }}"
retry_join = ["provider=aws tag_key=ConsulCluster tag_value=prod"]
ui = true
```

Start Consul:

```
sudo consul agent -config-dir=/etc/consul.d &
```

1.2 Nomad Setup

Install Nomad:

```
wget https://releases.hashicorp.com/nomad/1.5.0/nomad_1.5.0_linux_amd64.zip
unzip nomad_1.5.0_linux_amd64.zip && sudo mv nomad /usr/local/bin/
```

Create `nomad.hcl` on **server** nodes:

```
server {
  enabled = true
  bootstrap_expect = 3
}
client {
  enabled = false
}
bind_addr = "0.0.0.0"
advertise {
  http = "{{ GetInterfaceIP \"eth0\" }}:4646"
  rpc  = "{{ GetInterfaceIP \"eth0\" }}:4647"
  serf = "{{ GetInterfaceIP \"eth0\" }}:4648"
}
consul {
  address = "127.0.0.1:8500"
  enable_tag_override = true
}
```

On **client** nodes, use:

```
server {
  enabled = false
}
client {
  enabled = true
  servers = ["10.0.0.1:4647","10.0.0.2:4647","10.0.0.3:4647"]
}
bind_addr = "0.0.0.0"
consul {
  address = "127.0.0.1:8500"
  enable_tag_override = true
}
```

Start Nomad:

```
sudo nomad agent -config=/etc/nomad.d/nomad.hcl &
```

Verify cluster:

```
nomad server members
nomad node status
```

---

2. Nomad Job Specification for n8n

The job below defines two task groups. Tasks inside one group share the group's network namespace, so each `runners` task reaches its own broker at `localhost:5679`, exactly as a second container would in an ECS task or ACI container group.

`n8n.nomad`:

```
job "n8n" {
  datacenters = ["dc1"]
  type = "service"

  # The main process. Community Edition runs exactly one; multi-main is an
  # Enterprise feature. Do not raise this count — scale the worker group.
  group "server" {
    count = 1

    network {
      mode = "bridge"
      port "http" {
        to = 5678
      }
    }

    volume "n8n_data" {
      type      = "host"
      read_only = false
      source    = "n8n_data"
    }

    task "server" {
      driver = "docker"

      config {
        image = "n8nio/n8n:2.38.1"
        ports = ["http"]
      }

      env {
        DB_TYPE                          = "postgresdb"
        DB_POSTGRESDB_HOST               = "postgres.service.consul"
        DB_POSTGRESDB_PORT               = "5432"
        DB_POSTGRESDB_DATABASE           = "n8n"
        DB_POSTGRESDB_USER               = "n8n"
        N8N_HOST                         = "n8n.example.com"
        N8N_PROTOCOL                     = "https"
        WEBHOOK_URL                      = "https://n8n.example.com/"
        N8N_EDITOR_BASE_URL               = "https://n8n.example.com/"
        N8N_PROXY_HOPS                   = "1"
        EXECUTIONS_MODE                  = "queue"
        QUEUE_BULL_REDIS_HOST             = "redis.service.consul"
        QUEUE_BULL_REDIS_PORT             = "6379"
        OFFLOAD_MANUAL_EXECUTIONS_TO_WORKERS = "true"
        N8N_DEFAULT_BINARY_DATA_MODE       = "database"
        N8N_RUNNERS_MODE                 = "external"
        N8N_RUNNERS_BROKER_LISTEN_ADDRESS = "0.0.0.0"
        N8N_NATIVE_PYTHON_RUNNER          = "true"
      }

      template {
        destination = "secrets/n8n.env"
        env         = true
        data        = <<EOH
N8N_ENCRYPTION_KEY={{ with secret "secret/data/n8n" }}{{ .Data.data.encryption_key }}{{ end }}
N8N_RUNNERS_AUTH_TOKEN={{ with secret "secret/data/n8n" }}{{ .Data.data.runners_auth_token }}{{ end }}
DB_POSTGRESDB_PASSWORD={{ with secret "secret/data/n8n" }}{{ .Data.data.db_password }}{{ end }}
EOH
      }

      resources {
        cpu    = 500  # MHz
        memory = 512  # MB
      }
```

```
      volume_mount {
        volume      = "n8n_data"
        destination = "/home/node/.n8n"
        read_only   = false
      }

      service {
        name = "n8n"
        port = "http"

        check {
          name     = "readiness"
          type     = "http"
          path     = "/healthz/readiness"
          interval = "10s"
          timeout  = "2s"
        }
      }

      vault {
        policies = ["n8n-policy"]
      }
    }

    task "runners" {
      driver = "docker"

      config {
        image = "n8nio/runners:2.38.1"
      }

      env {
        N8N_RUNNERS_TASK_BROKER_URI       = "http://localhost:5679"
        N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT = "15"
      }

      template {
        destination = "secrets/runners.env"
        env         = true
        data        = <<EOH
N8N_RUNNERS_AUTH_TOKEN={{ with secret "secret/data/n8n" }}{{ .Data.data.runners_auth_token }}{{ end }}
EOH
      }

      resources {
        cpu    = 500
        memory = 512
      }

      vault {
        policies = ["n8n-policy"]
      }
    }
  }

  # The execution capacity. Raise this count to scale; each allocation is one
  # worker process plus its own runners task.
  group "worker" {
    count = 2

    network {
      mode = "bridge"
    }

    task "worker" {
      driver = "docker"

      config {
        image   = "n8nio/n8n:2.38.1"
        command = "worker"
        args    = ["--concurrency=10"]
      }

      env {
        DB_TYPE                          = "postgresdb"
        DB_POSTGRESDB_HOST               = "postgres.service.consul"
        DB_POSTGRESDB_PORT               = "5432"
        DB_POSTGRESDB_DATABASE           = "n8n"
        DB_POSTGRESDB_USER               = "n8n"
```

```
        EXECUTIONS_MODE                   = "queue"
        QUEUE_BULL_REDIS_HOST              = "redis.service.consul"
        QUEUE_BULL_REDIS_PORT              = "6379"
        N8N_DEFAULT_BINARY_DATA_MODE       = "database"
        N8N_RUNNERS_MODE                  = "external"
        N8N_RUNNERS_BROKER_LISTEN_ADDRESS = "0.0.0.0"
        N8N_NATIVE_PYTHON_RUNNER          = "true"
        N8N_GRACEFUL_SHUTDOWN_TIMEOUT     = "60"
      }

      template {
        destination = "secrets/worker.env"
        env         = true
        data        = <<EOH
N8N_ENCRYPTION_KEY={{ with secret "secret/data/n8n" }}{{ .Data.data.encryption_key }}{{ end }}
N8N_RUNNERS_AUTH_TOKEN={{ with secret "secret/data/n8n" }}{{ .Data.data.runners_auth_token }}{{ end }}
DB_POSTGRESDB_PASSWORD={{ with secret "secret/data/n8n" }}{{ .Data.data.db_password }}{{ end }}
EOH
      }

      resources {
        cpu    = 1000
        memory = 1024
      }

      vault {
        policies = ["n8n-policy"]
      }
    }

    task "runners" {
      driver = "docker"

      config {
        image = "n8nio/runners:2.38.1"
      }

      env {
        N8N_RUNNERS_TASK_BROKER_URI       = "http://localhost:5679"
        N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT = "15"
      }

      template {
        destination = "secrets/runners.env"
        env         = true
        data        = <<EOH
N8N_RUNNERS_AUTH_TOKEN={{ with secret "secret/data/n8n" }}{{ .Data.data.runners_auth_token }}{{ end }}
EOH
      }

      resources {
        cpu    = 500
        memory = 512
      }

      vault {
        policies = ["n8n-policy"]
      }
    }
  }
}
```

- **`group "server"` at `count = 1`** is not a placeholder. A second allocation is a second main process: both would fire every cron and polling trigger.
- **`group "worker"`'s `count`** is the scaling knob. Each allocation pairs one `worker` task with one `runners` task, because a worker cannot share another process's broker.
- **`command` / `args`** turn the same image into a worker. Workers get no `port` block and no `service` block — they pull jobs from Redis and receive no HTTP traffic.
- **`volume` host** uses the local filesystem on the server group only; with `N8N_DEFAULT_BINARY_DATA_MODE=database`, it holds `~/.n8n` state, not binary data, so it doesn't need to be a shared CSI volume. Filesystem binary mode is not supported in queue mode at all.
- **`N8N_ENCRYPTION_KEY`, `DB_POSTGRESDB_*`, and the queue variables** appear in both groups. Every process that touches the database needs the same key and the same database.

- **`service`** registers the main process with Consul for discovery and health checks against `/healthz/readiness`.
- **`vault`** integrates with HashiCorp Vault to inject secrets via `template`. The runners tasks receive `N8N_RUNNERS_AUTH_TOKEN` as a rendered environment variable, not a `_FILE` path — the runners image ignores `_FILE` variables.

---

3. HashiCorp Vault for Secrets

3.1 Configure Vault

```
vault secrets enable -path=secret kv-v2
vault kv put secret/n8n \
  encryption_key="$(openssl rand -hex 32)" \
  runners_auth_token="$(openssl rand -hex 32)" \
  db_password="SuperSecretPg"
```

`encryption_key` here is the one value that must be identical across every allocation, every region, and every other platform this same n8n install runs on — generate it once, store it in Vault, and never regenerate it per environment.

Create policy `n8n-policy.hcl`:

```
path "secret/data/n8n" {
  capabilities = ["read"]
}
```

Apply policy:

```
vault policy write n8n-policy n8n-policy.hcl
```

3.2 Nomad Vault Config

Add to `nomad.hcl`:

```
vault {
  enabled          = true
  address          = "https://vault.service.consul:8200"
  create_from_role = "nomad-role"
}
```

Nomad injects secrets into tasks via the `template` blocks above.

---

4. Deploy the Job

```
nomad job run n8n.nomad
```

Check status:

```
nomad job status n8n
nomad alloc status <allocation_id>
nomad logs -stderr <allocation_id> server
nomad logs -stderr <allocation_id> worker
nomad logs -stderr <allocation_id> runners
```

`nomad job status n8n` should show one `server` allocation and as many `worker` allocations as the group's count. If the worker allocations are missing, executions queue in Redis and never run.

---

5. Rolling Updates

Bump the pinned `image` tag in all four task blocks (both `n8nio/n8n:2.38.1` and `n8nio/runners:2.38.1`, in both groups — they must stay on the same n8n version) in `n8n.nomad` and re-run:

```
nomad job run n8n.nomad
```

Nomad replaces worker allocations one at a time without downtime. The `server` group at `count = 1` is different: Nomad must stop the old main allocation before starting the new one, so expect a short editor and webhook outage on every server-group update. Add an `update` stanza with `max_parallel = 1` and a `healthy_deadline` so the deployment waits for the new main to pass its readiness check.

---

6. Scaling

Scale the **worker** group, never the server group:

```
group "worker" {
  count = 6  # bump to six worker allocations
  # ...
}
```

Then:

```
nomad job run n8n.nomad
```

Verify allocations:

```
nomad job status n8n
```

If throughput is still short after scaling the count, raise `--concurrency` on the worker command instead, and size each allocation's `resources` to match. Raising `group "server"`'s count is not a scaling option: it produces duplicate scheduled executions, which is why the job pins it at 1.

---

7. Autoscaling with nomad-autoscaler

Deploy the **nomad-autoscaler** plugin and target the `worker` group only:

```
autoscaler {
  enabled = true
  policy "n8n-worker-scale" {
    description = "Scale n8n workers based on CPU"
    target {
      job = "n8n"
      group = "worker"
      metric_source = job
    }
    scaling {
      min = 2
      max = 10
      metric {
        name = "cpu"
        threshold = 60
      }
    }
  }
}
```

Apply to the Nomad server and restart. Never write an autoscaling policy against the `server` group.

---

8. Monitoring and Logs

- **Nomad UI**: `http://<server>:4646` for jobs, allocations, and logs
- **Consul UI**: `http://<consul-server>:8500` for service discovery
- **Logging**: forward driver logs to a central system via a logging driver plugin (see the Logging, Metrics, and OpenTelemetry chapter for the shared-filesystem event log caveat that applies to every worker-style task here)

---

9. Troubleshooting Common Issues

Issue	Cause	Solution
Allocation fails with `driver not found`	Docker driver not enabled on client	Ensure the Docker plugin is enabled and the client has Docker installed
Volume mount error	Host path missing or permissions incorrect	Create the host volume path on all clients and `chown` it to the Nomad user
Health check failures	Service check path incorrect or network misconfigured	Verify `/healthz/readiness` responds, check firewall/iptables
`server` task up but Code nodes fail	`runners` task not reachable or auth token mismatch	Confirm both tasks in the group resolve the same `runners_auth_token` from Vault, check `bridge` networking
Executions queue but never run	No `worker` allocations, or workers can't reach Redis	Check `nomad job status n8n` for worker allocations; verify `redis.service.consul` resolves from the client
Duplicate scheduled executions	`group "server"` count raised above 1	Set it back to 1 and scale `group "worker"` instead
Credentials unreadable on workers only	Worker group missing `N8N_ENCRYPTION_KEY`	Render the same Vault key into both groups' `template` blocks
Vault secrets not injected	Vault integration misconfigured or token missing	Check Nomad logs for Vault errors, verify Vault policies and role mapping
Consul service registration missing	`service` block not defined or Consul unreachable	Ensure the Consul agent is running, check the `nomad.hcl` Consul stanza
Rolling update stuck	Outdated client version or resource constraints	Upgrade Nomad clients, increase CPU/memory limits, inspect node status

Further reading

- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/enable-queue-mode
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/queue-mode
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/handle-binary-data

---

Chapter 26: High Availability

Overview

High availability for n8n Community Edition means one main process that comes back fast, and an execution tier that survives losing any single node. It does **not** mean several main processes behind a load balancer: running more than one main at a time requires `N8N_MULTI_MAIN_SETUP_ENABLED`, which is an **Enterprise feature**. On Community Edition, a second main means every cron and polling trigger fires twice.

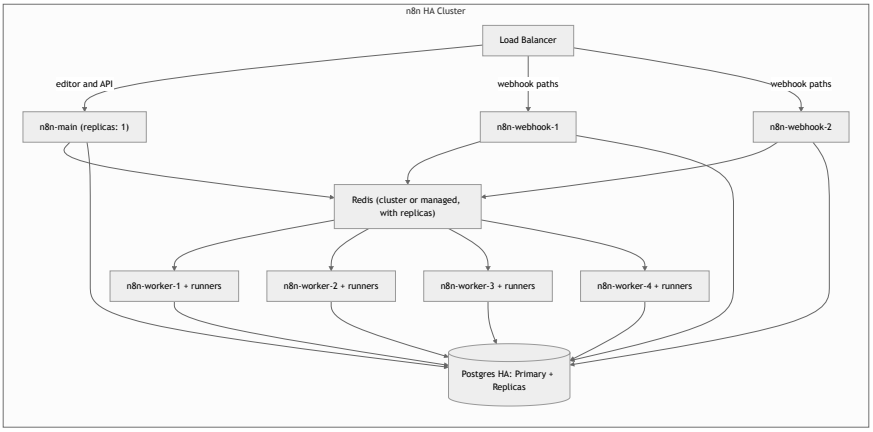
The topology this chapter builds:

- **One main** (`replicas: 1`), kept available by the orchestrator restarting it quickly — a `Recreate` update strategy and fast health checks, not a second copy.
- **N workers**, each running `n8n worker` with **its own runners sidecar**. This is where redundancy and throughput actually live: lose a worker and the queue redelivers its jobs.
- **Optional webhook processors** (`n8n webhook`), which *can* run multiple replicas behind the load balancer, so webhook ingress stays up even while the main pod is restarting.
- **Redis**, holding the queue.
- **Postgres**, holding workflows, credentials, and executions.

Sticky sessions are required only in an Enterprise multi-main setup, where the load balancer must pin an editor session to one main. With a single main there is nothing to pin, and with webhook processors there is no session to preserve — they serve stateless webhook requests.

In this chapter you will learn to:

1. **Design a Community Edition HA topology** across multiple nodes
2. **Deploy a single main** that restarts fast, with correct health checks
3. **Scale worker pools** horizontally against one Redis queue
4. **Add webhook processors** for webhook ingress availability
5. **Configure Redis and Postgres** for failure tolerance, including Redis Cluster
6. **Implement rolling upgrades** and understand where a gap is unavoidable
7. **Monitor service health** and implement alerting
8. **Understand the limits of this topology** before you try to stretch it across regions



Diagram

Prerequisites

- **Container orchestration** platform spanning multiple hosts or availability zones
- **Highly available Redis** in cluster mode or a managed service with replicas
- **Highly available Postgres** (Primary-Replica setup with failover)
- **Load balancer** supporting health checks and TLS termination
- The same `N8N_ENCRYPTION_KEY` value already generated, available to every pod as a Secret
- A `n8n-secrets` Secret holding `N8N_ENCRYPTION_KEY`, `N8N_RUNNERS_AUTH_TOKEN`, and `DB_PASSWORD`

---

1. Designing the Topology

1. **Main instance**
   - Exactly **one** replica. Multi-main is Enterprise-only; `N8N_MULTI_MAIN_SETUP_ENABLED` has no effect on Community Edition and running two mains anyway duplicates every schedule.
   - Availability comes from **fast replacement**: `strategy: Recreate`, a tight readiness probe, and a node pool with spare capacity so rescheduling is immediate.
   - Sets `N8N_ENCRYPTION_KEY`, the full `DB_POSTGRESDB_*` block, `EXECUTIONS_MODE=queue`, the queue variables, and `N8N_HOST` / `N8N_PROTOCOL=https` / `WEBHOOK_URL` / `N8N_EDITOR_BASE_URL` / `N8N_PROXY_HOPS=1` pointed at the load balancer's hostname — a mismatch breaks credential decryption or webhook URLs.
2. **Worker instances**
   - Deploy **4+** worker replicas, each running `n8n worker` with `EXECUTIONS_MODE=queue` and its **own** `runners` sidecar. Workers are the redundant tier.
3. **Webhook processors (optional)**
   - Deploy 2+ replicas running `n8n webhook` behind the load balancer, routed the webhook paths. They accept and enqueue webhook calls while the main is restarting.
4. **Redis**
   - A managed endpoint with replicas, or a Redis Cluster. Either way it is *one logical queue* — see Section 9 before you're tempted to spread it across regions.
5. **Postgres Primary-Replica**
   - Primary handles writes; read replicas serve read-only queries if needed.
6. **Infrastructure**
   - Place services in multiple fault domains (AZs, data centers) to mitigate hardware or network failures.

---

2. Deploying the Single Main

2.1 Kubernetes Example

Note what this manifest carries that a sketch usually omits: the encryption key, the whole database block, the URL variables, and a runners sidecar. Every one of them is required — this process touches the database, decrypts credentials, generates webhook URLs, and runs Code nodes.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: n8n-main
spec:
  replicas: 1
  strategy:
    type: Recreate          # never two mains at once, not even mid-rollout
  selector: { matchLabels: { app: n8n-main } }
  template:
    metadata: { labels: { app: n8n-main } }
    spec:
      containers:
        - name: n8n
          image: n8nio/n8n:2.38.1
          ports: [{ containerPort: 5678 }]
          env:
            - name: N8N_ENCRYPTION_KEY
              valueFrom: { secretKeyRef: { name: n8n-secrets, key: N8N_ENCRYPTION_KEY } }
            - name: DB_TYPE
              value: postgresdb
            - name: DB_POSTGRESDB_HOST
              value: postgres-primary
            - name: DB_POSTGRESDB_PORT
              value: "5432"
            - name: DB_POSTGRESDB_DATABASE
              value: n8n
            - name: DB_POSTGRESDB_USER
              value: n8n
            - name: DB_POSTGRESDB_PASSWORD
              valueFrom: { secretKeyRef: { name: n8n-secrets, key: DB_PASSWORD } }
            - name: N8N_HOST
              value: n8n.example.com
            - name: N8N_PROTOCOL
              value: https
            - name: WEBHOOK_URL
              value: https://n8n.example.com/
            - name: N8N_EDITOR_BASE_URL
              value: https://n8n.example.com/
            - name: N8N_PROXY_HOPS
              value: "1"
            - name: EXECUTIONS_MODE
              value: queue
            - name: QUEUE_BULL_REDIS_HOST
              value: redis
            - name: QUEUE_BULL_REDIS_PORT
              value: "6379"
            - name: OFFLOAD_MANUAL_EXECUTIONS_TO_WORKERS
              value: "true"
            - name: N8N_DEFAULT_BINARY_DATA_MODE
              value: database
            - name: N8N_RUNNERS_MODE
              value: external
            - name: N8N_RUNNERS_BROKER_LISTEN_ADDRESS
              value: 0.0.0.0
            - name: N8N_RUNNERS_AUTH_TOKEN
              valueFrom: { secretKeyRef: { name: n8n-secrets, key: N8N_RUNNERS_AUTH_TOKEN } }
            - name: N8N_NATIVE_PYTHON_RUNNER
              value: "true"
          readinessProbe:
            httpGet: { path: /healthz/readiness, port: 5678 }
            initialDelaySeconds: 10
            periodSeconds: 5
            failureThreshold: 3
          livenessProbe:
```

```
            httpGet: { path: /healthz, port: 5678 }
            initialDelaySeconds: 60
            periodSeconds: 10
        - name: runners
          image: n8nio/runners:2.38.1
          env:
            - name: N8N_RUNNERS_TASK_BROKER_URI
              value: http://localhost:5679
            - name: N8N_RUNNERS_AUTH_TOKEN
              valueFrom: { secretKeyRef: { name: n8n-secrets, key: N8N_RUNNERS_AUTH_TOKEN } }
            - name: N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT
              value: "15"
```

Expose via a **LoadBalancer** Service:

```
apiVersion: v1
kind: Service
metadata: { name: n8n-main-svc }
spec:
  type: LoadBalancer
  selector: { app: n8n-main }
  ports: [{ port: 5678, targetPort: 5678 }]
```

- **Health checks**: point the LB at `/healthz/readiness` every 5s, unhealthy after 3 failures. Short intervals are the availability mechanism here — the faster the failure is noticed, the faster the pod is replaced.
- **Session affinity**: leave it at `None`. There is one main, so there is nothing to pin. Sticky sessions become a requirement only in an Enterprise multi-main setup.
- **Recreate, not RollingUpdate**: a rolling update would briefly run two mains. Accept the few seconds of editor downtime instead.

---

3. Scaling Worker Pools Horizontally

Workers carry the same encryption key, the same database block, and the same queue settings as the main — and each pod pairs one `n8n worker` container with one `runners` sidecar. A worker cannot borrow another process's broker.

3.1 Kubernetes Deployment with HPA

```yaml
apiVersion: apps/v1
kind: Deployment
metadata: { name: n8n-worker }
spec:
  replicas: 4
  selector: { matchLabels: { app: n8n-worker } }
  template:
    metadata: { labels: { app: n8n-worker } }
    spec:
      containers:
        - name: n8n-worker
          image: n8nio/n8n:2.38.1
          command: ["n8n","worker","--concurrency=10"]
          env:
            - name: N8N_ENCRYPTION_KEY
              valueFrom: { secretKeyRef: { name: n8n-secrets, key: N8N_ENCRYPTION_KEY } }
            - name: DB_TYPE
              value: postgresdb
            - name: DB_POSTGRESDB_HOST
              value: postgres-primary
            - name: DB_POSTGRESDB_PORT
              value: "5432"
            - name: DB_POSTGRESDB_DATABASE
              value: n8n
            - name: DB_POSTGRESDB_USER
              value: n8n
            - name: DB_POSTGRESDB_PASSWORD
              valueFrom: { secretKeyRef: { name: n8n-secrets, key: DB_PASSWORD } }
            - name: N8N_HOST
              value: n8n.example.com
            - name: N8N_PROTOCOL
              value: https
            - name: WEBHOOK_URL
              value: https://n8n.example.com/
            - name: N8N_EDITOR_BASE_URL
              value: https://n8n.example.com/
            - name: EXECUTIONS_MODE
              value: queue
            - name: QUEUE_BULL_REDIS_HOST
              value: redis
            - name: QUEUE_BULL_REDIS_PORT
              value: "6379"
            - name: N8N_DEFAULT_BINARY_DATA_MODE
              value: database
            - name: N8N_RUNNERS_MODE
              value: external
            - name: N8N_RUNNERS_BROKER_LISTEN_ADDRESS
              value: 0.0.0.0
            - name: N8N_RUNNERS_AUTH_TOKEN
              valueFrom: { secretKeyRef: { name: n8n-secrets, key: N8N_RUNNERS_AUTH_TOKEN } }
            - name: N8N_NATIVE_PYTHON_RUNNER
              value: "true"
            - name: N8N_GRACEFUL_SHUTDOWN_TIMEOUT
              value: "60"
          readinessProbe:
            httpGet: { path: /healthz/readiness, port: 5678 }
            initialDelaySeconds: 10
        - name: runners
          image: n8nio/runners:2.38.1
          env:
            - name: N8N_RUNNERS_TASK_BROKER_URI
              value: http://localhost:5679
            - name: N8N_RUNNERS_AUTH_TOKEN
              valueFrom: { secretKeyRef: { name: n8n-secrets, key: N8N_RUNNERS_AUTH_TOKEN } }
```

```
            - name: N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT
              value: "15"
```

`N8N_GRACEFUL_SHUTDOWN_TIMEOUT=60` gives a terminating worker a minute to finish its in-flight executions before Kubernetes kills it; set the pod's `terminationGracePeriodSeconds` to at least the same value.

Horizontal Pod Autoscaler:

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata: { name: n8n-worker-hpa }
spec:
  scaleTargetRef: { apiVersion: apps/v1, kind: Deployment, name: n8n-worker }
  minReplicas: 4
  maxReplicas: 20
  metrics:
    - type: Resource
      resource: { name: cpu, target: { type: Utilization, averageUtilization: 70 } }
```

3.2 Webhook Processors for Ingress Availability

The one tier besides workers that Community Edition lets you run with several replicas is the webhook processor: `n8n webhook`. These processes accept webhook calls and enqueue them; they run no schedules and no polling triggers, so multiple replicas are safe.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata: { name: n8n-webhook }
spec:
  replicas: 2
  selector: { matchLabels: { app: n8n-webhook } }
  template:
    metadata: { labels: { app: n8n-webhook } }
    spec:
      containers:
        - name: n8n-webhook
          image: n8nio/n8n:2.38.1
          command: ["n8n","webhook"]
          ports: [{ containerPort: 5678 }]
          # Same env block as the worker above: N8N_ENCRYPTION_KEY, the full
          # DB_POSTGRESDB_* block, the URL variables, the queue variables,
          # and N8N_DEFAULT_BINARY_DATA_MODE.
```

Route the load balancer's `/webhook/*` and `/webhook-test/*` paths to this Deployment and everything else to `n8n-main-svc`. Webhook ingress then survives a main restart. Webhook processors do not need a runners sidecar: they enqueue rather than execute.

---

4. Redis for the Queue

A managed Redis endpoint with automatic failover is the simplest correct choice, and the one to prefer. Point every main, worker, and webhook processor at it with `QUEUE_BULL_REDIS_HOST` and `QUEUE_BULL_REDIS_PORT`, plus `QUEUE_BULL_REDIS_PASSWORD` and `QUEUE_BULL_REDIS_TLS=true` if the endpoint requires them.

4.1 Redis Cluster

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm install redis-cluster bitnami/redis-cluster \
  --set cluster.nodes=6,cluster.tls.enabled=false
```

Connecting to a Redis Cluster needs a different variable. `QUEUE_BULL_REDIS_HOST` opens a **standalone** client, which fails on the cluster's `MOVED` slot redirects. Use `QUEUE_BULL_REDIS_CLUSTER_NODES` instead — a comma-separated list of `host:port` seed nodes:

```
QUEUE_BULL_REDIS_CLUSTER_NODES=redis-cluster-0:6379,redis-cluster-1:6379,redis-cluster-2:6379
```

When `QUEUE_BULL_REDIS_CLUSTER_NODES` is set, `QUEUE_BULL_REDIS_HOST` is **ignored**. Set the cluster variable on every main, worker, and webhook processor, or the ones still on the standalone variable will silently fail to reach the queue.

- **Persistence**: each node has its own PVC for AOF files.
- **Seeds**: any subset of masters works as seeds; the client discovers the rest.

4.2 Redis Sentinel Is Not Supported by That Variable

There is no Sentinel-specific queue variable in n8n. `QUEUE_BULL_REDIS_CLUSTER_NODES` expects cluster nodes, not Sentinel addresses, and pointing it at Sentinels does not work. If your Redis is Sentinel-managed, put a **Sentinel-aware proxy** in front of it — one that resolves the current master and presents a single stable endpoint — and give n8n that proxy's address as `QUEUE_BULL_REDIS_HOST`. A **managed Redis endpoint** that handles failover behind one hostname solves the same problem with less to run.

Either way, this is still a single logical queue with one keyspace, made durable within one region. That distinction — many nodes, one queue — is exactly what breaks down once you try to run it across two regions; see Section 9.

---

5. Highly Available Postgres

Use a managed service (AWS RDS, Cloud SQL) with Multi-AZ failover:

- **Primary**: write endpoint
- **Read Replicas**: automatically promoted if primary fails

Or deploy your own cluster using Patroni via Helm:

```
helm install postgres-ha bitnami/postgresql-ha \
  --set replicaCount=2,postgresql.postgresqlPassword=SuperSecretPg
```

- **Patroni** handles leader election and failover.
- **Virtual IP** via a Kubernetes Service ensures clients always connect to the primary.
- **TLS**: for a managed database, set `DB_POSTGRESDB_SSL_ENABLED=true` and `DB_POSTGRESDB_SSL_CA_FILE=/certs/ca.pem` with the CA mounted read-only in every pod.

These mechanics don't change when you stretch the topology across regions — only the failover time and acceptable lag do, covered in the Multi-Region Deployment and Failover chapter.

---

6. Rolling Upgrades Without Downtime

When updating the n8n image:

1. **Update every Deployment** with the new pinned image tag — main, workers, webhook processors, and all their runners sidecars — together. The runners image must match the n8n version.
2. **Workers and webhook processors** roll one pod at a time with no gap; the queue holds work while a worker restarts.
3. **The main rolls with `Recreate`**, so there is a short window with no main process. Schedules do not fire during it, and the editor is briefly unavailable. This is the cost of a single-main topology, and it is smaller than the cost of running two mains.
4. **Load balancer** health checks keep traffic off pods that are not ready.

Run database migrations on the main only. A worker started against an unmigrated schema fails; roll the main first, wait for `/healthz/readiness`, then roll the workers.

For non-Kubernetes platforms:

- **Nomad**: re-submit the job; the worker group rolls incrementally, the single-count server group is replaced in place.

---

7. Monitoring and Alerting

- **Prometheus & Grafana**: scrape `/metrics` (`N8N_METRICS=true`) from main and worker instances — see the Logging, Metrics, and OpenTelemetry chapter for the full scrape configuration.
- **Alertmanager**: fire alerts if main replicas fall below the desired count or queue length exceeds a threshold.

Example Prometheus rule:

```
groups:
- name: n8n.alerts
  rules:
  - alert: MainDown
    expr: kube_deployment_status_replicas_unavailable{deployment="n8n-main"} > 0
    for: 2m
    labels: { severity: critical }
    annotations: { summary: "n8n main pods unavailable" }
```

---

8. Advanced Scenarios

8.1 Multi-AZ Kubernetes Cluster

- Ensure node pools span AZs; use `topologySpreadConstraints` to evenly distribute pods.
- For StatefulSets (Redis), set `podAffinity` to avoid co-location.

---

9. Why This Doesn’t Simply Extend to Multiple Regions

Everything above scales one logical queue and one logical database across many nodes *within one region*. Both assumptions stop holding across regions:

- **Redis**: cluster replication is designed for low-latency links between nodes. Stretch it across regions and you either accept added round-trip latency on every queue operation, or run two independent Redis deployments — two independent queues, where a job enqueued in one region is invisible to workers in the other. There’s no way to get one logical, low-latency, cross-region queue from a single Redis deployment.
- **Postgres**: cross-region replication is asynchronous by default, so a promoted secondary can be seconds behind at failover.

The consequence, covered in the next chapter: a multi-region n8n deployment is **active-passive for execution** unless you deliberately accept two independent, non-synchronized queues running disconnected sets of executions.

---

10. Troubleshooting Common Issues

Issue	Cause	Solution
LB not routing to the new main pod	Readiness probe failing	Increase `initialDelaySeconds`, confirm `/healthz/readiness` responds
Every schedule fires twice	More than one main process running	Set `replicas: 1` and `strategy: Recreate`; multi-main needs an Enterprise license
Worker pods stuck Pending	Insufficient resources or missing Secret mounts	Scale the node pool, verify volume mounts
Workers connected but no jobs run	Workers on `QUEUE_BULL_REDIS_HOST` while the main uses cluster nodes	Set `QUEUE_BULL_REDIS_CLUSTER_NODES` on every process, or none
Queue errors mentioning `MOVED`	Standalone Redis client pointed at a Redis Cluster	Replace `QUEUE_BULL_REDIS_HOST` with `QUEUE_BULL_REDIS_CLUSTER_NODES`
Redis failover leaves n8n stuck	Sentinel addresses given to n8n directly	Front Sentinel with a Sentinel-aware proxy, or use a managed endpoint
Postgres failover not detected	Service VIP pointed to the old primary	Use Patroni's HAProxy sidecar or an updated Service selector
Rolling update stalls	`maxUnavailable` or parallelism too high on the worker Deployment	Set `maxUnavailable: 1` and `maxSurge: 1`
Credentials fail to decrypt on workers	`N8N_ENCRYPTION_KEY` missing or different on the worker Deployment	Mount the same Secret key into every Deployment
Code nodes fail on workers only	Worker pods have no runners sidecar	Add the `runners` container to the worker pod spec, sharing `N8N_RUNNERS_AUTH_TOKEN`

Further reading

- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/enable-queue-mode
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/queue-mode
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/handle-binary-data

---

Chapter 27: Multi-Region Deployment and Failover

Overview

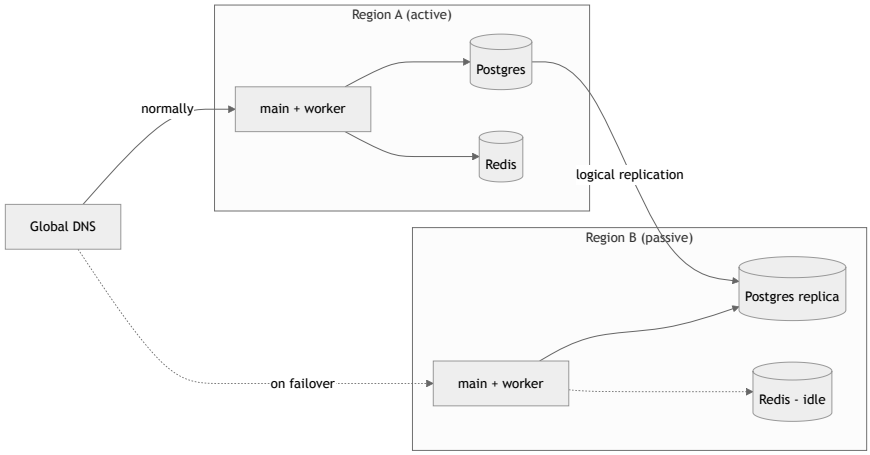
Deploying n8n across multiple geographic regions buys you two things: recovery if a whole region goes down, and data locality where regulation demands it.

It does not buy you an active-active execution tier. Each region's queue is one Redis deployment, and there is no way to make one logical queue both low-latency and consistent across a WAN. **A multi-region n8n deployment is active-passive for execution.** The passive region keeps a replicated database and idle compute; failover promotes it. The alternative — two independent queues — is active-active for traffic only, with two disconnected execution histories.

Each region also runs the Community Edition topology from the High Availability chapter: one main, N workers with a runners sidecar each, optional webhook processors, one Redis, one Postgres.

In this chapter you will learn to:

1. **Design a topology**, and why active-passive is the honest default for execution
2. **Synchronize PostgreSQL** with logical replication or managed cross-region replicas
3. **Decide how the queue works across regions**, since one Redis queue can't be active-active
4. **Deploy identical fleets**, including the same encryption key, per region
5. **Configure global DNS failover**
6. **Manage cross-region backup and restore**
7. **Automate region-aware CI/CD**
8. **Monitor globally** and **test failover**



Diagram

Prerequisites

- At least **two** region-level deployments, each following the High Availability chapter's topology
- **Global DNS provider** supporting health checks and failover routing (Route 53, Cloud DNS, Traffic Manager)
- **Cross-region networking** (VPC Peering, VPN, or PrivateLink) for database replication
- The same `N8N_ENCRYPTION_KEY` already in your secret store, ready to copy in rather than regenerate

1. Multi-Region Topology Options

1.1 Active-Passive (the default this chapter builds)

One region's main/worker fleet and Redis queue are authoritative; the other stands by with a synced database and idle compute. Failover promotes the standby's database and repoints DNS and the queue at it — simpler consistency, at the cost of a non-zero RTO.

1.2 Active-Active with Independent Queues

Both regions run their own main/worker fleet against their own Redis queue and, ideally, their own database — accepting that executions started in Region A are invisible to Region B. This is active-active for *traffic*, not for a single execution history: it only suits workloads where each region's workflows are genuinely independent, not one workflow whose steps might land on either region's queue.

There's no in-between: a single Redis-backed queue can't safely be both low-latency and consistent across two regions (see the High Availability chapter, Section 9). Pick the option your workflows actually require before building it.

---

2. Cross-Region PostgreSQL Replication

2.1 Managed Services

AWS RDS cross-region read replicas, **GCP Cloud SQL** asynchronous replicas, or **Azure Database** geo-replication all promote on failure the same way.

2.2 Self-Hosted

Logical Replication: publish/subscribe slots across regions.

```
-- On primary (Region A):
CREATE PUBLICATION n8n_pub FOR ALL TABLES;
-- On secondary (Region B):
CREATE SUBSCRIPTION n8n_sub
CONNECTION 'host=primary-a.example.com port=5432 dbname=n8n user=replicator password=pass'
PUBLICATION n8n_pub;
```

Failover: promote Region B's replica to primary and repoint n8n deployments via DNS. This is asynchronous replication — Region B can be seconds behind at the moment of failure, so failover always risks losing the last few writes. The mechanics are the same as the single-region Patroni/RDS setup in the High Availability chapter; only the achievable RPO changes, because the link is now a WAN hop instead of a rack.

---

3. The Queue Across Regions

One Redis deployment is one queue. That single fact is what makes cross-region execution active-passive.

3.1 Active-Passive: One Redis, One Region

Point every main, worker, and webhook processor — in both regions — at the Redis deployment in the currently-active region, using the same queue variables everywhere: `QUEUE_BULL_REDIS_HOST` and `QUEUE_BULL_REDIS_PORT` for a standalone or managed endpoint, or `QUEUE_BULL_REDIS_CLUSTER_NODES` (comma-separated `host:port`) if that region's Redis is a cluster. When `QUEUE_BULL_REDIS_CLUSTER_NODES` is set, `QUEUE_BULL_REDIS_HOST` is ignored, so pick one form and use it consistently across both regions' manifests.

The passive region's instances stay deployed but idle for execution until failover repoints them at the newly-promoted Redis. This keeps exactly one execution history.

3.2 Active-Active: Independent Queues Per Region

Each region runs its own Redis exactly as in the High Availability chapter, with no cross-region replication between them. Each region's fleet only talks to its own queue, and each region runs its own single main. Simpler than synchronizing Redis over a WAN, but a webhook landing in Region A can only be processed there — no failover of in-flight executions to Region B, only failover of *new* traffic via DNS.

Don't reach for cross-region Redis replication as a third option: it adds complexity without a queue that's both low-latency and consistent in both regions.

---

4. Deploying Fleets in Each Region

Follow the High Availability chapter's templates in each region:

- **One main per region**, at `replicas: 1` with a `Recreate` strategy. Multi-main is Enterprise-only, and "one main per region" is not an exception to that — in active-passive only the active region's main is serving; in active-active the two mains are separate installs against separate databases.
- **Worker Deployments**: identical in both regions, each pod carrying its own runners sidecar, down to the pinned `n8nio/n8n:2.38.1` and `n8nio/runners:2.38.1` tags.
- **`N8N_ENCRYPTION_KEY`**: the exact same value in both regions' secret stores, mounted into *every* pod — main, workers, and webhook processors alike. A region generating its own key can't decrypt credentials replicated in from the other's database, and neither can a worker that was never given the key.
- **`DB_POSTGRESDB_*`**: the full block on every pod, pointed at that region's database endpoint, with `DB_POSTGRESDB_SSL_ENABLED=true` and `DB_POSTGRESDB_SSL_CA_FILE` for a managed database.
- **Queue variables**: the same form in both regions — `QUEUE_BULL_REDIS_HOST` / `QUEUE_BULL_REDIS_PORT`, or `QUEUE_BULL_REDIS_CLUSTER_NODES` for a cluster, never a mix.
- **`N8N_HOST`** / **`WEBHOOK_URL`** / **`N8N_EDITOR_BASE_URL`**: point at the *global* hostname in front of your DNS load balancer, not a region-specific one, so failover doesn't change what callers need to know.

Automate with region-parameterized Helm values or Terraform modules that vary only the regional endpoints, never the encryption key or image tags.

---

5. Global DNS Load Balancing

5.1 AWS Route 53 Latency-Based or Failover Routing

```
# Create health check for Region A
aws route53 create-health-check --caller-reference "n8n-A" --health-check-config file://healthcheck-
        a.json
# Create record set with failover routing policy
aws route53 change-resource-record-sets --hosted-zone-id Z123... --change-batch file://failover-
        rules.json
```

For active-passive (Section 1.1), use **failover routing**, not latency-based routing — all traffic stays on the active region until a health check trips.

5.2 GCP Cloud DNS / Traffic Director

Define equivalent health checks and failover policy against each region's load balancer.

---

6. Failover and Failback

1. **Routine health checks** monitor region endpoints — for active-passive, specifically the active region's Redis and Postgres.
2. On failure, **global DNS** shifts traffic to the standby, whose Postgres replica is promoted and whose Redis becomes authoritative.
3. Once the original primary is restored, shift traffic back deliberately, not automatically — failback means re-establishing replication in the opposite direction first.
4. **Session state** lives in Postgres or Redis, never in-process, so failover doesn't strand it.

---

7. Stateful Backup & Restore Across Regions

Take automated daily snapshots of Postgres and Redis in each region (per the Backup, Restore, and Disaster Recovery chapter), copy them cross-region (RDS snapshot replication, geo-redundant storage), and keep a cold-start restore path ready in case logical replication has stalled.

---

8. Region-Aware CI/CD

Parameterize your pipelines (see the CI/CD and GitOps chapter) to deploy the same pinned tag to every region, canary each region before promoting globally, and vary only endpoint values — never the encryption key or image tags — via Terraform workspaces or `helmfile`.

---

9. Monitoring & Centralized Alerting

Push Prometheus metrics from every region to a central instance via federation or remote write, build unified Grafana dashboards with one data source per region, and route a global “critical” Alertmanager path for any region-wide outage.

---

10. Testing Failover

- **Simulate an outage**: disable health checks or block access to Region A's Redis and Postgres.
- **Observe**: DNS shifts to Region B, the replica promotes, and you confirm how much data (if any) was lost in flight.
- **Re-enable** Region A and verify failback re-establishes replication before traffic returns.

---

11. Troubleshooting Common Issues

Issue	Cause	Solution
DNS failover delays	DNS TTL too high	Lower TTL (e.g., 60s)
Data inconsistency after failback	Asynchronous replication lag	Monitor lag before promoting; accept the loss window or go synchronous
Executions “stuck” after failover	Workers still pointed at the old region’s Redis	Update `QUEUE_BULL_REDIS_HOST` (or `QUEUE_BULL_REDIS_CLUSTER_NODES`) on every process as part of the failover runbook
Some processes never pick up jobs after failover	A mix of `QUEUE_BULL_REDIS_HOST` and `QUEUE_BULL_REDIS_CLUSTER_NODES`	Use one form everywhere; the cluster variable makes the host variable a no-op
Duplicate scheduled executions after failover	Both regions’ mains active at once	In active-passive, scale the passive region’s main to zero until failover completes
Global load balancer misrouting	Health checks misconfigured or endpoints mislabeled	Verify health check paths, ports, and target endpoints
Credentials fail to decrypt after failover	Regions used different `N8N_ENCRYPTION_KEY` values, or workers lack it	Confirm both secret stores hold the identical key and every pod mounts it before promoting

Further reading

- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/enable-queue-mode
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/queue-mode
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/deployment

---

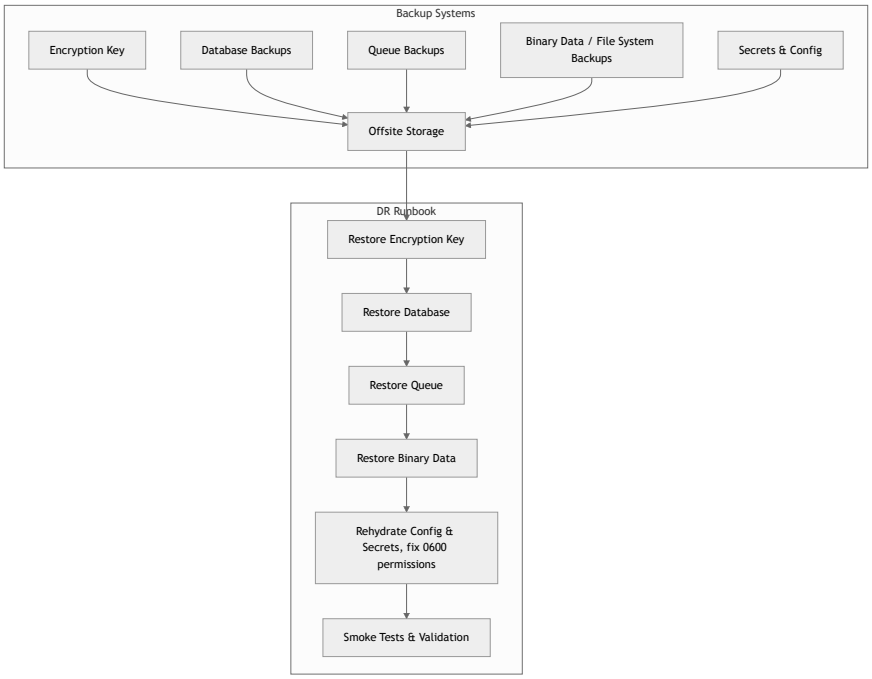
Chapter 28: Backup, Restore, and Disaster Recovery

Overview

A backup is only useful if you have restored from it at least once. This chapter covers what to back up, how to back it up on each platform, and — the part usually skipped — the exact restore commands, tested, in the order they have to run. The encryption key comes first every time: a database restored without its matching key contains workflows whose credentials cannot be decrypted.

This chapter covers:

1. **Identify critical data**, starting with the encryption key
2. **Back up databases** (Postgres, SQLite) across platforms
3. **Back up message queues** (Redis AOF/RDB)
4. **Back up binary data** — `~/.n8n` or your S3 bucket, depending on mode
5. **Back up secrets** and configuration
6. **Automate backups** via cron, CronJobs, periodic jobs, cloud snapshots
7. **Off-site replication and retention**
8. **Restore each component**, including the 2.0 file-permission step
9. **Full DR runbooks**: RTO/RPO, with the encryption key as restore step one
10. **Advanced DR**: pilot light, warm standby



Diagram

Prerequisites

- **Access** to all environments (SSH, kubectl, nomad, cloud consoles)
- **Sufficient storage** in object store or backup servers
- **Backup tools** installed (pg_dump, redis-cli, tar, cloud CLIs)
- Defined **RPO** and **RTO** targets

1. Identifying Critical Data

Component	Data to Back Up	Location / Tools
Encryption key	`N8N_ENCRYPTION_KEY` value	Secret store export, distinct from the DB backup
n8n Config	`~/.n8n/config`, settings file	File tar; Kubernetes PVC
Postgres	Database dumps (`pg_dump`)	pg_dump, cloud snapshot APIs
SQLite	`~/.n8n/database.sqlite`	File copy or tar
Redis	AOF/RDB files	redis-cli, tar
Binary data	`~/.n8n/binaryData` or the `N8N_EXTERNAL_STORAGE_S3_*` bucket	tar, or `aws s3 sync`
Secrets & Config	Env files, Vault / Secrets Manager	Encrypted export; `vault kv get`
Container Images	Custom Docker images	Push to registry
Logs	Execution and container logs	Cloud logging exports

The encryption key comes first, deliberately. Every workflow's stored credentials are encrypted with it; a database restored without the matching key is full of workflows that can authenticate to nothing. Back it up as its own artifact — a line in your secret manager's own backup, not just "somewhere in the `.env` tarball" — and make it restore step one in Section 9.

---

2. Database Backups

2.1 PostgreSQL

Local / VM:

```
# Full logical backup
PGPASSWORD=SuperSecretPg pg_dump -h localhost -U n8n -F c -b -v -f /backups/n8n_$(date +%F).dump n8n
```

Docker Compose:

Run `pg_dump` **inside** the Compose project, not in a detached `docker run`. A standalone container joins Docker's default bridge network, where the Compose service name `postgres` does not resolve, and the command fails with "could not translate host name". `docker compose exec` runs in the service's own container, on the project network, with the image's tools already present:

```
docker compose exec -T postgres \
  pg_dump -U n8n -F c -b n8n > ./backups/n8n_$(date +%F).dump
```

`-T` disables TTY allocation so the dump streams cleanly into the redirect. The password comes from the container's own `POSTGRES_PASSWORD`, so nothing sensitive appears on the host command line.

Kubernetes CronJob:

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: pg-backup
  namespace: n8n
spec:
  schedule: "0 2 * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: backup
            image: postgres:18
            env:
            - name: PGPASSWORD
              valueFrom:
                secretKeyRef:
                  name: n8n-db-credentials
                  key: DB_PASSWORD
            command:
            - sh
            - -c
            - |
              pg_dump -h postgres -U n8n -F c -b n8n > /backups/n8n_$(date +%F).dump
            volumeMounts:
            - name: backup-volume
              mountPath: /backups
          restartPolicy: OnFailure
          volumes:
          - name: backup-volume
            persistentVolumeClaim:
              claimName: backup-pvc
```

Managed Service Snapshots: - **AWS RDS**: automated snapshots plus cross-region copy. - **GCP Cloud SQL**: automated backups plus point-in-time recovery. - **Azure**: geo-redundant backup retention.

2.2 SQLite

```
cp ~/.n8n/database.sqlite /backups/database_$(date +%F).sqlite
```

For containerized:

```
docker cp n8n-app:/home/node/.n8n/database.sqlite ./backups/database_$(date +%F).sqlite
```

---

3. Message Queue Backups (Redis)

AOF/RDB Backup:

```
docker exec n8n-redis redis-cli save
docker run --rm \
  -v redis_data:/data \
  -v $(pwd)/backups:/backups \
  alpine sh -c "tar czf /backups/redis_$(date +%F).tar.gz -C /data dump.rdb appendonly.aof"
```

Kubernetes CronJob: same shape as the Postgres one, mounting the Redis PVC.

Redis holds in-flight and recent queue state, not workflow definitions or credentials — a restore recovers what was mid-flight, but a fresh empty Redis is also a valid recovery target if you're willing to lose in-flight executions.

---

4. Binary Data Backups

What you back up here depends on which binary data mode you're running (see AWS ECS Fargate, Section 5, for how that choice gets made):

Filesystem mode — back up the directory directly:

```
tar czvf n8n_binary_$(date +%F).tar.gz -C /home/node/.n8n binaryData
```

Database mode — nothing extra to do; binary data is already inside the Postgres dump from Section 2.

S3 mode — replicate or version the bucket itself rather than tarring it through a container:

```
aws s3api put-bucket-versioning --bucket n8n-binary-data --versioning-configuration Status=Enabled
aws s3 sync s3://n8n-binary-data s3://n8n-binary-data-dr --source-region us-east-1 --region us-west-2
```

Kubernetes CronJob and **Nomad periodic job**: same pattern as the database backup jobs, mounting whichever volume or bucket applies.

---

5. Secrets & Configuration Backups

5.1 Environment Files and the n8n Home Directory

Version-control `.env` files with secret values redacted, and rotate sensitive values via CI — never commit the encryption key to Git.

Back up the n8n home directory as its own artifact, since it holds the `config` settings file the restore runbook re-permissions:

```
docker run --rm -v n8n_data:/data -v "$(pwd)/backups:/backup" \
  alpine tar czf /backup/n8n_home_$(date +%F).tar.gz -C /data .
```

5.2 Vault / Secret Manager

- **HashiCorp Vault**: enable audit logging and snapshot periodically:

  ```
  vault operator raft snapshot save backup.snap
  ```

- **AWS Secrets Manager**: export via `aws secretsmanager get-secret-value`.
- **Kubernetes Secrets**: `kubectl get secret n8n-encryption-key -o yaml > secret-backup.yaml`, encrypted at rest and stored separately from the database backup.

---

6. Automation & Scheduling

Schedule DB, binary-data, and Redis backup scripts via cron on a backup server, Kubernetes CronJobs, Nomad's HCL `periodic` stanza, or cloud functions triggering snapshots.

---

7. Off-site Replication & Retention

1. **Object Storage**: upload backup artifacts to S3, GCS, or Azure Blob with lifecycle rules:

   ```
   aws s3 cp backups/ s3://n8n-backups/ --recursive --storage-class STANDARD_IA
   ```

2. **Cross-region copy**: replicate daily snapshots across regions for DR.

3. **Retention**: daily backups for 7 days, weekly for 4 weeks, monthly for 12 months, via bucket lifecycle rules.

---

8. Restore Procedures

8.1 Restore the Encryption Key First

Before touching the database, put `N8N_ENCRYPTION_KEY` back on every instance that will run against it — main, workers, and their runners sidecars alike. Restoring with a different key than the one the database was encrypted with leaves every credential unreadable; n8n won't crash, but every node using a credential fails at execution time.

8.2 Restore Postgres

Stop every n8n process first (`docker compose stop n8n n8n-worker`, a Kubernetes scale-down, or `nomad job stop`), leaving Postgres running.

Restore through the Compose service, for the same reason the dump ran that way — a detached `docker run` cannot resolve the service name `postgres`:

```
docker compose exec -T postgres \
  pg_restore -U n8n -d n8n --clean --if-exists -v < ./backups/n8n_2026-04-21.dump
```

`--clean --if-exists` drops the existing objects before recreating them, so a restore into a database n8n has already initialized does not fail on duplicates. Then restart n8n.

Kubernetes restore: use a **Job** manifest similar to the CronJob but running `pg_restore` against the database Service; scale the n8n Deployments to zero first and back up afterwards.

If you are also restoring the config directory (Section 8.4), n8n 2.0's `N8N_ENFORCE_SETTINGS_FILE_PERMISSIONS=true` default requires the settings file to be mode `0600`. A tar extraction or volume copy commonly resets it, and n8n refuses to start against looser permissions.

8.3 Restore Redis

```
# Stop the Redis service

docker run --rm \
  -v redis_data:/data \
  -v $(pwd)/backups:/backup \
  alpine sh -c "tar xzf /backup/redis_2026-04-21.tar.gz -C /data"

# Restart the Redis service
```

8.4 Restore Binary Data and Config

```
docker run --rm \
  -v n8n_data:/data \
  -v $(pwd)/backups:/backup \
  alpine sh -c "tar xzf /backup/n8n_binary_2026-04-21.tar.gz -C /data"
```

Now fix the config file's permissions. `/data` exists only inside a container that has the volume mounted — running `chmod 0600 /data/config` on the host does nothing, because the host has no `/data`. Run it in its own throwaway container with the volume attached:

```
docker run --rm -v n8n_data:/data alpine chmod 0600 /data/config
```

Verify before starting n8n:

```
docker run --rm -v n8n_data:/data alpine ls -l /data/config
```

For S3-mode binary data, point `N8N_EXTERNAL_STORAGE_S3_*` at the DR bucket — no local extraction step.

8.5 Restore Secrets

- **Vault**: `vault operator raft snapshot restore backup.snap`
- **Kubernetes**: `kubectl apply -f secret-backup.yaml` (after editing metadata)
- **AWS**: re-import secrets via `aws secretsmanager create-secret`

8.6 A Tested Restore Runbook for Docker Compose

Run this end to end on a scratch host at least once before you need it. It restores a Compose deployment whose volumes are `postgres_data`, `n8n_data`, and `redis_data`.

Case A — database binary mode (binary data lives inside the Postgres dump):

```
# 1. Encryption key. Put the backed-up value in .env before anything starts.
grep -q '^N8N_ENCRYPTION_KEY=' .env || echo "N8N_ENCRYPTION_KEY=<restored-key>" >> .env

# 2. Bring up Postgres alone.
docker compose up -d postgres
docker compose exec -T postgres pg_isready -U n8n -d n8n

# 3. Restore the dump through the service.
docker compose exec -T postgres \
  pg_restore -U n8n -d n8n --clean --if-exists < ./backups/n8n_2026-04-21.dump

# 4. Restore ~/.n8n state and fix the settings-file permission inside a container.
docker run --rm -v n8n_data:/data -v "$(pwd)/backups:/backup" \
  alpine sh -c "tar xzf /backup/n8n_home_2026-04-21.tar.gz -C /data"
docker run --rm -v n8n_data:/data alpine chmod 0600 /data/config

# 5. Start the rest of the stack.
docker compose up -d

# 6. Verify.
docker compose logs -f n8n            # expect "Editor is now accessible", no migration errors
docker compose exec -T postgres psql -U n8n -d n8n -c "select count(*) from workflow_entity;"
```

Then, in the editor: open a workflow that uses a stored credential and run it. A successful authenticated call is the only proof the encryption key matched. A credential that fails here means step 1 used the wrong key — stop and fix that before doing anything else, because re-saving the credential overwrites the ciphertext.

Case B — S3 binary mode: identical, minus nothing and plus one step. The dump no longer carries binary data, so after step 3 confirm the bucket:

```
aws s3 ls s3://n8n-binary-data --summarize | tail -3
```

Then set `N8N_DEFAULT_BINARY_DATA_MODE=s3` and the `N8N_EXTERNAL_STORAGE_S3_*` variables to the bucket you are restoring against — the DR bucket if the original region is gone — before step 5. Open a past execution that produced a file and download its output; if the bucket or credentials are wrong, the metadata renders but the download 404s.

Redis is optional in both cases. Restoring it recovers executions that were mid-flight; starting with an empty Redis is a valid recovery target if you accept losing them. If you skip it, note in the incident record which executions were in flight so they can be re-triggered.

Rehearse this quarterly, and time it. The number you get is your real RTO.

---

9. Disaster Recovery Runbooks

1. **Declare RPO/RTO** targets and **pre-stage** recovery infrastructure in standby mode.
2. **Failover**, in order: restore or confirm the encryption key on every instance (always step one) → restore the database from the latest backup or promote a replica → restore Redis, or accept an empty queue → restore binary data and config, fixing `chmod 0600` → deploy the n8n application.
3. **Validate**: run sample workflows, verify connectivity, confirm a credential decrypts.
4. **Failback**: once the primary is healthy, reverse the process.

Maintain the runbook as version-controlled documentation and rehearse quarterly — the encryption-key and file-permission steps are the two most commonly skipped in a rushed real recovery.

---

10. Advanced DR Scenarios

- **Pilot Light**: minimal footprint environment always running in the DR region.
- **Warm Standby**: application running at reduced capacity, ready for scale-up.
- **Immutable Backups**: WORM storage to protect against ransomware.
- **Database PITR**: continuous WAL archiving for point-in-time restores.

---

11. Troubleshooting Common Issues

Issue	Cause	Solution
`could not translate host name "postgres"`	`pg_dump` / `pg_restore` run in a detached `docker run` on the default network	Use `docker compose exec -T postgres ...` so the command runs on the project network
`pg_restore` fails on existing objects	Restoring into a database n8n already initialized	Add `--clean --if-exists`, or restore into a fresh database
`chmod: /data/config: No such file or directory`	`chmod` run on the host, where `/data` does not exist	`docker run --rm -v n8n_data:/data alpine chmod 0600 /data/config`
n8n refuses to start after a restore	Settings file looser than `0600` under `N8N_ENFORCE_SETTINGS_FILE_PERMISSIONS`	Re-run the containerized `chmod` above and verify with `ls -l`
Workflows load but every credential fails	Restored with a different `N8N_ENCRYPTION_KEY`	Restore the original key and restart; do not re-save the credentials first, that overwrites the ciphertext
Binary data missing from restored executions	Ran in `s3` mode; the dump never contained it	Point `N8N_EXTERNAL_STORAGE_S3_*` at the (DR) bucket and confirm object access
Executions marked running that never finish	Redis restored empty, or not restored	Accept the loss and re-trigger; record which executions were in flight
Restore succeeds but migrations then fail	Dump taken from a newer n8n version than the image being restored onto	Restore onto the same pinned version that produced the dump, then upgrade

Further reading

- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/handle-binary-data
- https://docs.n8n.io/deploy/host-n8n/install-options/install-using-docker-compose
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/deployment
- https://docs.n8n.io/changelog/v20-breaking-changes

Chapter 29: CI/CD and GitOps

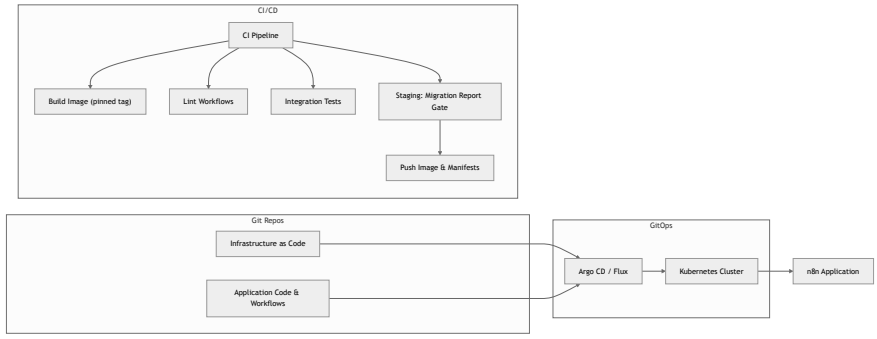
Overview

Continuous Integration and Continuous Deployment (CI/CD) combined with GitOps practices let you automate build/test/deploy, version both code and infrastructure in Git, promote through environments with minimal manual intervention, and roll back easily on regression.

The unit of deployment throughout this chapter is a **pinned tag bump** — `n8nio/n8n:2.38.1` to `n8nio/n8n:2.38.2`, say — committed to Git and promoted through environments, never a `latest` tag that silently moves underneath you.

In this chapter you will learn to:

1. **Structure your Git repositories** for infrastructure and workflows
2. **Write CI pipelines** (GitHub Actions, GitLab CI/CD) that build a pinned image, lint workflows, and run integration tests
3. **Use GitOps tools** (Argo CD, Flux) to synchronize Git with your Kubernetes or Nomad clusters
4. **Manage environment promotion** via branch-based triggers or pull-request workflows
5. **Add a staging gate that checks the n8n 2.0 migration report** before any major-version bump reaches production
6. **Securely handle secrets** in CI/CD pipelines and GitOps
7. **Implement preview environments** for pull requests
8. **Automate rollback**, including publishing and unpublishing workflows via the current n8n CLI
9. **Integrate policy checks** to enforce best practices on your workflows and infrastructure
10. **Monitor pipeline health** and alert on failures



Diagram

Prerequisites

- **Git** hosting (GitHub, GitLab, Bitbucket) for two repositories:
  - `n8n-app`: workflow JSON exports, custom extensions, Dockerfile, application code
  - `n8n-infra`: Helm charts, Kubernetes manifests, Nomad job files, or Terraform code
- **CI/CD platform** enabled on your repository
- **GitOps tool** installed in your target cluster (Argo CD v2.x or Flux v2.x)
- **Secrets management**: access to HashiCorp Vault, a cloud provider secret store, or CI secrets
- **Docker registry** credentials for pushing images
- A **staging environment** that mirrors production closely enough to trust its migration report

1. Git Repository Structure

1.1 `n8n-app` Repository

```
n8n-app/
├── workflows/
│   ├── my-first-workflow.json
│   └── ...
├── extensions/
│   └── custom-nodes.js
├── Dockerfile
├── .github/
│   └── workflows/ci.yaml
└── README.md
```

- **`workflows/`**: version-controlled exports of your n8n workflows
- **`extensions/`**: any custom JS nodes or utilities
- **`Dockerfile`**: `FROM n8nio/n8n:2.38.1` (or whichever pinned version this repo currently tracks)

1.2 `n8n-infra` Repository

```
n8n-infra/
├── helm-chart/
│   └── n8n/
├── k8s/
│   ├── deployment.yaml
│   ├── service.yaml
│   └── ingress.yaml
├── nomad/
│   └── n8n.nomad
├── .argo/
│   └── n8n-app.yaml
├── .flux/
│   └── kustomization.yaml
└── README.md
```

---

2. CI Pipeline Example: GitHub Actions

`.github/workflows/ci.yaml` in `n8n-app`:

```yaml
name: CI

on:
  push:
    branches: [ main, release/* ]
  pull_request:
    branches: [ main ]

jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Install JSONLint
        run: npm install -g jsonlint
      - name: Lint Workflows
        run: |
          for file in workflows/*.json; do
            jsonlint "$file"
          done

  build-and-push:
    needs: lint
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'
    steps:
      - uses: actions/checkout@v4
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3
      - name: Log in to Registry
        uses: docker/login-action@v3
        with:
          registry: ${{ secrets.REGISTRY_URL }}
          username: ${{ secrets.REGISTRY_USER }}
          password: ${{ secrets.REGISTRY_PASSWORD }}
      - name: Build and Push Pinned Image
        uses: docker/build-push-action@v6
        with:
          context: .
          push: true
          tags: ${{ secrets.REGISTRY_URL }}/n8n:2.38.1
      - name: Deploy to Staging
        uses: appleboy/ssh-action@v1
        with:
          host: ${{ secrets.STAGING_HOST }}
          username: ${{ secrets.SSH_USER }}
          key: ${{ secrets.SSH_KEY }}
          script: |
            kubectl set image deployment/n8n n8n-container=${{ secrets.REGISTRY_URL }}/n8n:2.38.1 -n
        n8n-staging
```

- **Lint** ensures all workflow exports are valid JSON.
- **Build & Push** tags the image with the pinned n8n version, never `latest`.
- **Deploy to Staging** updates the staging Deployment first — production only gets this tag after Section 5's gate passes.

---

3. GitLab CI/CD Example

`n8n-app/.gitlab-ci.yml`:

```yaml
stages:
  - lint
  - build
  - staging
  - deploy

variables:
  IMAGE: $CI_REGISTRY_IMAGE:2.38.1

lint:
  stage: lint
  script:
    - npm install -g jsonlint
    - jsonlint workflows/*.json

build:
  stage: build
  image: docker:27
  services:
    - docker:27-dind
  before_script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
  script:
    - docker build -t $IMAGE .
    - docker push $IMAGE
  only:
    - main

staging:
  stage: staging
  image: bitnami/kubectl:1.31
  script:
    - kubectl set image deployment/n8n n8n-container=$IMAGE -n n8n-staging
  only:
    - main

deploy:
  stage: deploy
  image: bitnami/kubectl:1.31
  script:
    - kubectl set image deployment/n8n n8n-container=$IMAGE -n n8n
  when: manual
  only:
    - main
```

- **Manual deploy** to production protects it from an unreviewed rollout.
- **`staging`** runs before `deploy` unconditionally, giving you a place to check the migration report (Section 5) before the manual production gate.

---

4. GitOps with Argo CD

4.1 Define an Argo CD Application

`n8n-infra/.argo/n8n-app.yaml`:

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: n8n
  namespace: argocd
spec:
  project: default
  source:
    repoURL: https://github.com/your-org/n8n-infra.git
    targetRevision: HEAD
    path: k8s
  destination:
    server: https://kubernetes.default.svc
    namespace: n8n
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
    syncOptions:
      - CreateNamespace=true
```

4.2 Deploy Argo CD

```
kubectl create namespace argocd
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-
        cd/stable/manifests/install.yaml
```

---

5. GitOps with Flux v2

5.1 Bootstrap Flux

```
flux bootstrap github \
  --owner=your-org \
  --repository=n8n-infra \
  --branch=main \
  --path=./k8s \
  --personal
```

```
# k8s/kustomization.yaml
apiVersion: kustomize.toolkit.fluxcd.io/v1beta2
kind: Kustomization
metadata:
  name: n8n
  namespace: flux-system
spec:
  interval: 5m0s
  path: "./"
  prune: true
  sourceRef:
    kind: GitRepository
    name: infra
```

---

6. The Migration Report Gate for Major Upgrades

A minor bump (`2.38.1` to `2.38.2`) can go through the pipeline above unattended. A **major** bump — the next one being n8n 3.0 — needs a human to check **Settings** → **Migration Report** on staging first: it lists Workflow and Instance Issues by severity, and there's no documented API to script the check, so this is a manual gate.

Add it as a required step between "deploy to staging" and "promote to production":

```
staging-migration-gate:
  needs: build-and-push
  runs-on: ubuntu-latest
  environment:
    name: production
    # requires a reviewer approval configured in repo settings
  steps:
    - name: Reminder
      run: |
        echo "Before approving: open Settings > Migration Report on staging."
        echo "Confirm zero Critical issues under both Workflow Issues and Instance Issues."
        echo "Refresh the report after any workflow fix and re-check before approving."
```

Wiring this as a GitHub Actions `environment` with required reviewers (or an equivalent gate in GitLab / Argo CD) turns "someone remembered to check" into "the pipeline won't proceed until someone checks." Treat any Critical issue as a pipeline failure, not a note for later.

---

7. Managing Secrets Securely

- **GitHub Actions/GitLab CI**: store secrets in the platform's encrypted secrets store, never commit plaintext.
- **Sealed Secrets** (Bitnami) or Flux's SOPS integration: encrypt Kubernetes Secrets in Git.
- **Vault integration**: use the Vault CSI driver or Helm chart to fetch runtime secrets, including `N8N_ENCRYPTION_KEY` and `N8N_RUNNERS_AUTH_TOKEN`.

---

8. Preview Environments for Pull Requests

1. **CI Step** creates a temporary namespace (`pr-123`), deploys the Helm chart with a PR-scoped image tag.
2. **Comments** back the preview URL.
3. **Cleanup** when the PR is closed via a webhook.

```yaml
jobs:
  preview:
    if: github.event_name == 'pull_request'
    runs-on: ubuntu-latest
    steps:
      - name: Helm Deploy Preview
        uses: azure/helm@v1
        with:
          namespace: pr-${{ github.event.number }}
          release-name: n8n-pr-${{ github.event.number }}
          chart-path: ./n8n-infra/helm-chart
          values: image.tag=2.38.1,ingress.host=n8n-pr-${{ github.event.number }}.example.com
```

Note the image tag here is still the pinned release version, not a PR-specific build tag — only the workflow JSON and infra values change per preview; the n8n binary itself stays on the version you're testing against.

---

9. Publishing and Unpublishing Workflows

Older pipelines used the n8n CLI's `update:workflow` to flip a workflow's active state. That's gone; n8n 2.0's Publish/Unpublish model with versioning replaces it:

```
n8n import:workflow --input=workflows/my-first-workflow.json
n8n publish:workflow <workflow-id> --versionId <version-id>
```

and a rollback or emergency stop uses:

```
n8n unpublish:workflow <workflow-id>
# or, for an incident affecting many workflows:
n8n unpublish:workflow --all
```

Wire `publish:workflow` into the same CI job that promotes infrastructure, and `unpublish:workflow` into your rollback job (Section 10), so a bad workflow export can be pulled independently of the n8n image.

---

10. Automated Rollback Procedures

- Use `helm rollback` or `kubectl rollout undo deployment/n8n` in CI if smoke tests fail against the new pinned tag.
- Use `n8n unpublish:workflow` (Section 9) when the regression is in a workflow export, not the image.
- Configure GitOps tools to automatically revert on failed sync (Argo CD's rollback on sync failure).

---

11. Policy Enforcement

Integrate **OPA Gatekeeper** or **Kyverno** to:

- **Validate** that workloads have resource limits and liveness/readiness probes.
- **Deny** deployments that use a `latest` image tag.

```yaml
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sRequiredLabels
metadata:
  name: require-n8n-labels
spec:
  enforcementAction: deny
  match:
    kinds:
      - apiGroups: ["apps"]
        kinds: ["Deployment"]
    labelSelector:
      matchLabels:
        app: n8n
  parameters:
    labels:
      - "app"
      - "environment"
```

---

12. Monitoring Pipeline Health

- **GitHub Actions**: configure notifications on failure via Slack or email.
- **Dashboards**: use built-in Actions or GitLab CI Analytics to track pipeline duration and failure rates.
- **Alerts**: integrate with PagerDuty or Opsgenie for critical failures, including a failed migration-report gate.

For the workflow-side half of this story — how versioning, publishing, and rollback work from inside n8n itself — see Workflow Versioning and Publishing.

---

13. Troubleshooting Common Issues

Issue	Cause	Solution
Deploy pushes a tag that already exists	Image tag not bumped with the version	Tag with the pinned n8n version and bump it in the same commit; never reuse a tag
Argo CD reports `OutOfSync` immediately after a deploy	`kubectl set image` applied outside Git	Change the tag in the manifest repo and let the GitOps controller apply it
Pods roll but still run the old image	Same tag repushed with new content	Use a distinct pinned tag per release so the digest changes
Major-version deploy reaches production unreviewed	The migration-report gate is a step, not an environment	Attach the gate job to a protected environment with required reviewers
Migration report shows Critical issues after “fixing” them	Report not refreshed since the workflow edit	Press Refresh in Settings → Migration Report and re-read both issue lists
`n8n update:workflow` not found	Command removed in 2.0	Use `publish:workflow <id> [--versionId]` and `unpublish:workflow <id> | --all`
Workers run an older image than the main	Only the main Deployment’s tag was bumped	Bump main, workers, webhook processors, and every runners sidecar together
Secrets missing in a preview environment	Namespace-scoped Secret not created for the PR namespace	Sync the Secret into each preview namespace, or use a cluster-wide secret store
Kyverno or Gatekeeper blocks the deploy	Policy denies unpinned tags or missing probes	Fix the manifest rather than the policy — the policy is the point

Further reading

- https://docs.n8n.io/changelog/v20-migration-tool
- https://docs.n8n.io/changelog/v20-breaking-changes
- https://docs.n8n.io/changelog/v30-breaking-changes
- https://docs.n8n.io/deploy/host-n8n/install-options/install-using-docker-compose

---

Chapter 30: Logging, Metrics, and OpenTelemetry

Overview

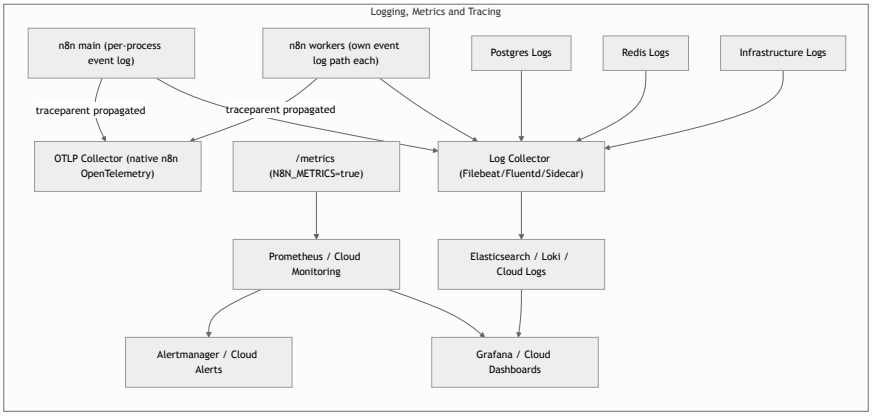
Logs, metrics, and traces are how you find out what a running n8n deployment is doing. This chapter sets up all three on Community Edition, using only what Community Edition has: container logs plus n8n's own log files, the built-in Prometheus endpoint, and n8n's native OpenTelemetry support. Streaming the event bus to external destinations is an Enterprise feature and is not part of this setup — Section 3 explains what still applies locally.

Centralized collection lets you:

- **Aggregate logs** from all components (n8n application, workers, database, queue, infrastructure) into a single pane
- **Correlate events** across services and time to diagnose root causes
- **Visualize metrics** such as workflow execution rates, error counts, queue lengths, and resource usage
- **Set alerts** on key indicators (high error rate, slow response, resource saturation)
- **Trace a single execution end-to-end**, across nodes and across a boundary crossed by a webhook call

In this chapter you will learn to:

1. **Define logging formats** and log levels in n8n and infrastructure
2. **Collect application logs** via sidecars or log drivers (Docker, Kubernetes, Nomad)
3. **Avoid event-log collisions** when multiple workers share a filesystem
4. **Ship logs** to centralized systems (ELK/EFK, Splunk, CloudWatch, Azure Monitor, GCP Logging)
5. **Expose Prometheus metrics** with `N8N_METRICS=true` and scrape them
6. **Create dashboards** in Grafana or cloud vendor portals
7. **Configure alerting** via Alertmanager, cloud alerts, or external services
8. **Trace executions end-to-end with n8n's native OpenTelemetry support**
9. **Integrate log aggregation** across Docker, Kubernetes, Nomad, and serverless environments
10. **Best practices**: log retention, index management, cost control, data privacy



Diagram

Prerequisites

- A running n8n deployment you can set environment variables on — main, workers, and any webhook processors

- A **log collector** you can deploy alongside it (Filebeat, Fluentd, Fluent Bit, Promtail) or a platform log driver (awslogs, Cloud Logging, Azure Monitor)
- A **log store and viewer**: Elasticsearch and Kibana, Loki and Grafana, or your cloud provider's equivalent
- **Prometheus** (or a cloud metrics service that scrapes Prometheus endpoints) with network access to the n8n pods on port 5678
- An **OTLP-compatible tracing backend**: a self-hosted OpenTelemetry Collector forwarding to Jaeger or Tempo, or a vendor OTLP endpoint
- Network paths that keep `/metrics` and `/healthz` off the public listener

---

1. Defining Log Formats and Levels

n8n's own logging is configured with environment variables, not a Winston config file you edit directly:

```
N8N_LOG_LEVEL=info        # error | warn | info | debug
N8N_LOG_OUTPUT=console,file
N8N_LOG_FILE_LOCATION=/home/node/.n8n/logs/n8n.log
N8N_LOG_FILE_SIZE_MAX=50  # MB per file
N8N_LOG_FILE_COUNT_MAX=60
```

For custom Docker images, set the same in `Dockerfile`:

```
ENV N8N_LOG_LEVEL=info \
    N8N_LOG_OUTPUT=console,file
```

---

2. Collecting Application Logs

2.1 Docker Compose (Filebeat)

```
services:
  filebeat:
    image: elastic/filebeat:8.17.5
    volumes:
      - /var/lib/docker/containers:/var/lib/docker/containers:ro
      - /etc/filebeat/filebeat.yml:/usr/share/filebeat/filebeat.yml:ro
    depends_on:
      - n8n
```

`filebeat.yml` snippet:

```
filebeat.inputs:
- type: container
  paths:
    - /var/lib/docker/containers/*/*.log
```

2.2 Kubernetes (Fluentd DaemonSet)

```
apiVersion: apps/v1
kind: DaemonSet
metadata: { name: fluentd }
spec:
  template:
    spec:
      containers:
      - name: fluentd
        image: fluent/fluentd-kubernetes-daemonset:v1.16-debian-elasticsearch8-1
        env:
          - name: FLUENT_ELASTICSEARCH_HOST
            value: "elasticsearch.logging.svc.cluster.local"
          - name: FLUENT_ELASTICSEARCH_PORT
            value: "9200"
        volumeMounts:
          - name: varlog
            mountPath: /var/log
          - name: varlibdockercontainers
            mountPath: /var/lib/docker/containers
            readOnly: true
      volumes:
        - name: varlog
          hostPath: { path: /var/log }
        - name: varlibdockercontainers
          hostPath: { path: /var/lib/docker/containers }
```

2.3 Nomad (Sidecar Logging)

In your Nomad job `task "server"` add:

```
logs {
  max_files      = 5
  max_file_size  = 10
}
```

Ship via a GELF-compatible driver to Logstash:

```
driver = "docker"
config {
  image = "n8nio/n8n:2.38.1"
  logging {
    type = "gelf"
    config {
      gelf-address = "udp://logstash.service.consul:12201"
    }
```

```
  }
}
```

---

3. Per-Process Event Log Files on Shared Filesystems

First, what is and is not available here. n8n's event bus can **stream logs to external destinations** — syslog, a webhook, Sentry — and that streaming is an **Enterprise feature**, unavailable on Community Edition. Do not build a pipeline around it.

What still applies on Community Edition is the **local event log file**. The event bus persists events to disk before anything else happens to them, and that file is also what n8n reads to recover executions after a crash. The path is `<n8n-user-folder>/n8nEventLog.log` by default, with a process-specific suffix, and `N8N_EVENTBUS_LOGWRITER_LOGFULLPATH` overrides it. Both the file and the variable work regardless of licence; only forwarding to an external destination is gated.

The default is fine when each process has its own home directory volume, which is the common case for main and worker Deployments with per-pod volumes.

It stops being fine once several worker processes **share** a writable filesystem — a host volume in Nomad, an NFS-backed PVC, or a reused EFS mount. Concurrent writes to the same file corrupt it and drop events, which means a crashed worker's executions may not recover. Give every worker its own explicit path:

```
# On worker 1
N8N_EVENTBUS_LOGWRITER_LOGFULLPATH=/shared/n8n-events/worker-1.log
# On worker 2
N8N_EVENTBUS_LOGWRITER_LOGFULLPATH=/shared/n8n-events/worker-2.log
```

Derive the suffix from the allocation ID, pod name, or hostname in your deployment templates rather than hardcoding it, so it stays unique as you scale the worker count. `N8N_EVENTBUS_LOGWRITER_MAXTOTALMESSAGESPERFILE` bounds how many lines are parsed during recovery, which matters once several of these files are being tailed by the same log shipper.

Since external streaming is not available, point your log collector (Section 4) at these files and at container stdout. That gets the same events into Elasticsearch, Loki, or CloudWatch without depending on an Enterprise feature.

---

4. Shipping Logs to Central Systems

Platform	Log Collector	Storage	Visualization
On-prem / VM	Filebeat, Fluentd	Elasticsearch, Loki	Kibana, Grafana
Kubernetes	Fluentd, Fluent Bit	EFK stack, Loki	Kibana, Grafana
Nomad	GELF/Logstash driver	ELK, Splunk	Kibana, Splunk UI
AWS ECS/Fargate	awslogs	CloudWatch Logs	CloudWatch Console
GCP Cloud Run	Cloud Logging	Cloud Logging	Cloud Console
Azure ACI	Azure Monitor	Log Analytics	Azure Portal

4.1 ELK Stack Example

- **Elasticsearch** cluster (3 nodes for HA)
- **Logstash** pipelines to parse JSON and add metadata
- **Kibana** for dashboarding

4.2 Grafana Loki + Promtail

- **Promtail** collects container logs, pushes to **Loki**
- **Grafana** queries both **Loki** (logs) and **Prometheus** (metrics)

---

5. Prometheus Metrics

n8n exposes Prometheus-compatible metrics natively — no sidecar or middleware needed:

```
N8N_METRICS=true
```

Set it on every main and worker instance you want visibility into; each exposes its own `/metrics`.

`prometheus.yml`:

```
scrape_configs:
  - job_name: n8n
    metrics_path: /metrics
    static_configs:
      - targets:
          - n8n-main-svc.n8n.svc.cluster.local:5678
          - n8n-worker-svc.n8n.svc.cluster.local:5678
```

Optional `N8N_METRICS_INCLUDE_*` variables narrow or widen what's exposed — a name prefix, default Node.js metrics, and per-feature detail (API counts, cache hits, queue depth). Because `/metrics` reveals operational detail, keep it off the public listener — scrape it over the same internal network or VPC connector used for Redis and Postgres.

---

6. Creating Dashboards

In **Grafana**, build dashboards for:

- **Workflow throughput** and **failure rate** from the n8n-exposed counters
- **Queue length** via a Redis exporter
- **Resource utilization**: CPU, memory, disk I/O per pod/node

Use templating variables to switch between namespaces, regions, or clusters.

---

7. Configuring Alerting

7.1 Alertmanager (Prometheus)

```yaml
groups:
- name: n8n_alerts
  rules:
  - alert: HighWorkflowErrorRate
    expr: |
      increase(n8n_workflow_errors_total[5m]) / increase(n8n_workflow_executions_total[5m]) > 0.05
    for: 10m
    labels:
      severity: critical
    annotations:
      summary: "High error rate in n8n workflows"
      description: "Error rate > 5% over 10 minutes"
```

Configure `alertmanager.yml` to send to Slack, PagerDuty, or email.

7.2 Cloud Vendor Alerts

- **CloudWatch Alarm**: on an `Error` log metric filter or CPU > 80%
- **GCP Alerting**: on a Logging metric count or a Prometheus alert
- **Azure Alerts**: Log Analytics query and metric alerts on container insights

---

8. Distributed Tracing with Native OpenTelemetry

Earlier editions had you hand-roll tracing with a Node.js instrumentation sidecar and a Jaeger collector. n8n now ships OpenTelemetry support directly, replacing the sidecar entirely.

Enable it with environment variables on every instance:

```
N8N_OTEL_ENABLED=true
N8N_OTEL_EXPORTER_OTLP_ENDPOINT=http://otel-collector:4318
N8N_OTEL_EXPORTER_OTLP_HEADERS_FILE=/mnt/otel-headers   # if your collector requires auth
```

or, since n8n 2.27, configure it interactively from **Settings** → **OpenTelemetry** in the editor (environment variables take precedence over the UI setting on self-hosted instances, so pick one source of truth per environment).

Once enabled, each execution emits a root **`workflow.execute`** span carrying the workflow ID, status, and mode, with a nested **`node.execute`** span per node recording item counts. n8n propagates the W3C **`traceparent`** header both ways: an inbound webhook's `traceparent` becomes the parent of the execution's spans, and outbound HTTP nodes inject their own — so a trace from a caller continues through the workflow and beyond, with no manual header wiring.

Point `N8N_OTEL_EXPORTER_OTLP_ENDPOINT` at whatever OTLP-compatible backend you already run — a self-hosted collector forwarding to Jaeger or Tempo, or a cloud vendor's OTLP ingestion endpoint (Cloud Trace, X-Ray via an OTel-compatible bridge, Azure Monitor). A minimal collector config:

```
receivers:
  otlp:
    protocols:
      http:
      grpc:
exporters:
  otlp/tempo:
    endpoint: "tempo:4317"
service:
  pipelines:
    traces:
      receivers: [otlp]
      exporters: [otlp/tempo]
```

---

9. Multi-Platform Integration

Deployment Type	Log Shipping	Metrics Scraping	Tracing
Kubernetes	Fluentd/Promtail to EFK	Prometheus Operator scraping `/metrics`	Native OTel to a collector Deployment
Nomad	GELF/Logstash driver	Nomad Prometheus plugin scraping `/metrics`	Native OTel to a collector job
AWS ECS/Fargate	awslogs to CloudWatch	CloudWatch Metrics or self-scraped `/metrics`	Native OTel to an OTLP endpoint
GCP Cloud Run	Cloud Logging	Cloud Monitoring	Native OTel to Cloud Trace's OTLP endpoint
Azure ACI	Azure Monitor	Azure Monitor	Native OTel to Application Insights's OTLP endpoint

---

10. Best Practices

1. **Structured logging**: consistent fields (`timestamp`, `level`, `component`, `workflowId`, `nodeName`) across every instance.
2. **Correlation**: the OpenTelemetry `traceparent` (Section 8) is now the correlation ID — don't invent a separate one.
3. **Retention policies**: balance cost and compliance — store high-volume logs for 7-14 days, metrics for 30-90 days.
4. **Index management**: for Elasticsearch, use time-based indices and ILM policies for rollover.
5. **Security & privacy**: scrub or mask sensitive data (credentials, PII) before logging; never enable `N8N_METRICS` or `/healthz` on a public listener without review.
6. **Resource limits**: give logging and monitoring agents appropriate CPU/memory requests and limits.
7. **Alert fatigue**: tune thresholds and use multi-level alerts (warning vs. critical).
8. **Per-process event logs**: on any shared filesystem, confirm every worker's `N8N_EVENTBUS_LOGWRITER_LOGFULLPATH` is actually unique before you scale past one worker — this is the single most common cause of silently missing events in a queue-mode deployment.

---

11. Troubleshooting Common Issues

Issue	Cause	Solution
`/metrics` returns 404	`N8N_METRICS` not set on that process	Set `N8N_METRICS=true` on every main, worker, and webhook processor you scrape
Metrics only from the main, none from workers	Prometheus targets only the main Service	Add a headless Service or pod-level scrape config covering the worker pods
Log streaming to syslog or a webhook does nothing	External event-bus destinations are an Enterprise feature	Collect container stdout and the local event log file instead (Sections 2-4)
Events missing after a worker crash	Several workers writing one shared event log file	Give each worker a unique `N8N_EVENTBUS_LOGWRITER_LOGFULLPATH`
Log files fill the volume	`N8N_LOG_OUTPUT` includes `file` with no rotation bound	Set `N8N_LOG_FILE_SIZE_MAX` and `N8N_LOG_FILE_COUNT_MAX`
No spans reach the collector	`N8N_OTEL_ENABLED` unset, or the OTLP endpoint unreachable	Set it on every process; confirm the collector's HTTP receiver on port 4318
Traces start fresh at n8n instead of continuing	Caller not sending `traceparent`, or a proxy stripping it	Pass the header through the proxy; n8n propagates it in both directions
UI OpenTelemetry setting appears to be ignored	Environment variables take precedence on self-hosted instances	Pick one source of truth per environment: env vars or Settings → OpenTelemetry
`/metrics` or `/healthz` reachable from the internet	Endpoints exposed on the public listener	Scrape over the internal network only; block both paths at the reverse proxy

Further reading

- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/deployment
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/enable-queue-mode
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/queue-mode

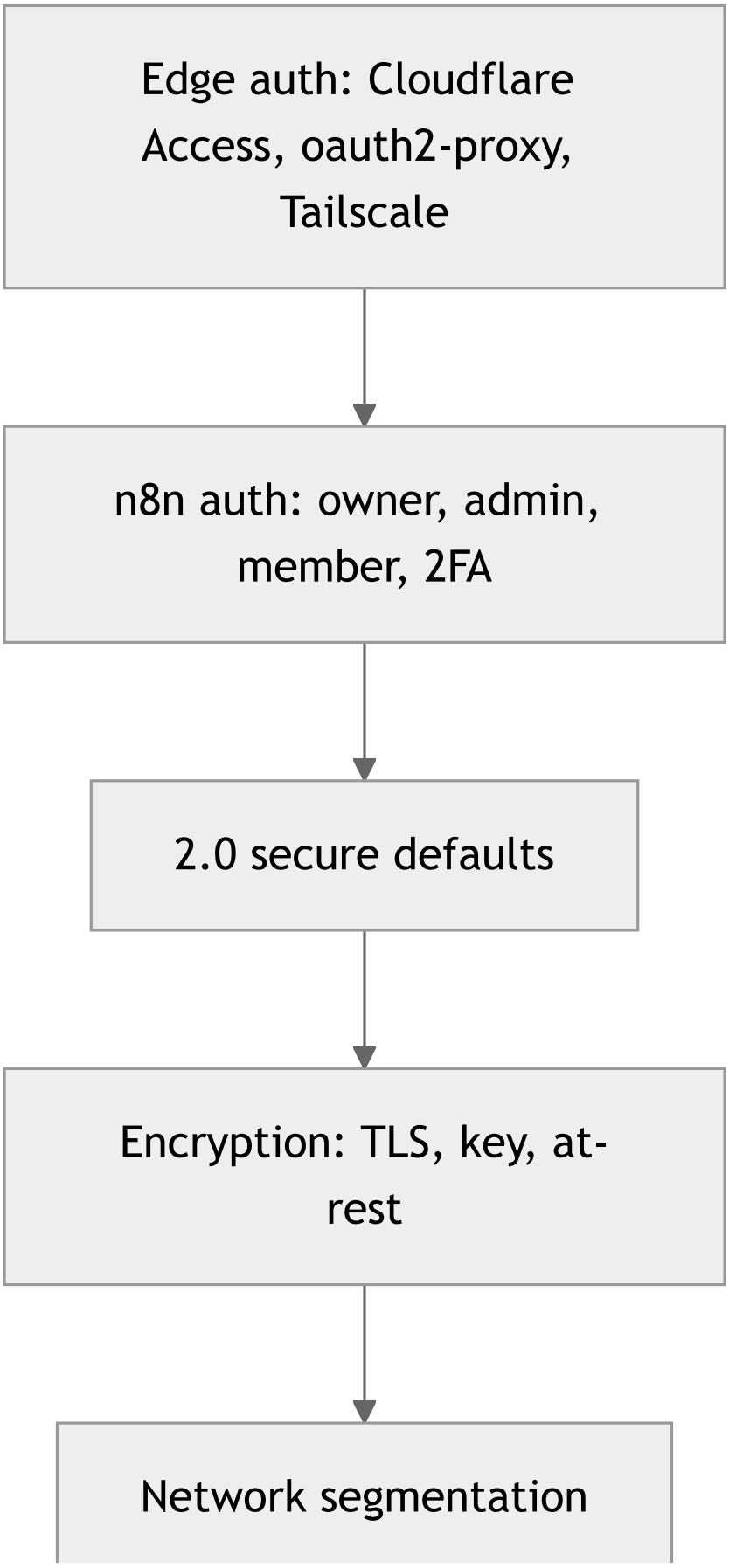
---

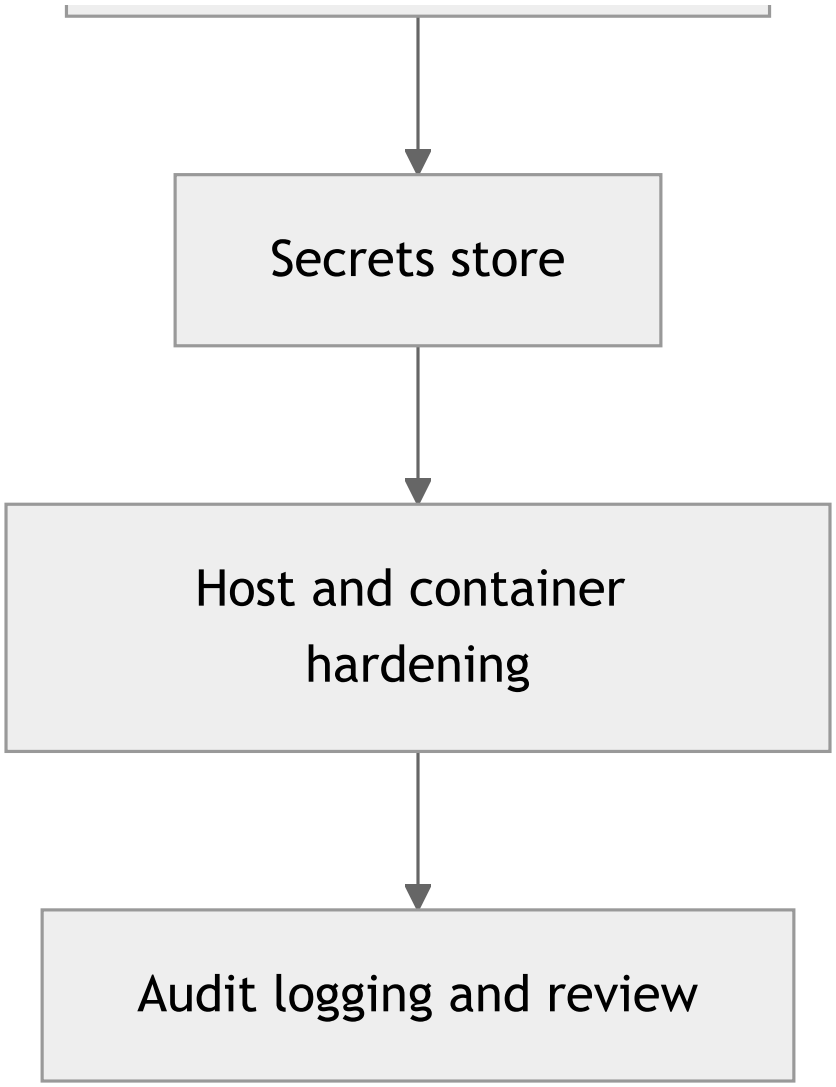
Chapter 31: Security Hardening for n8n 2.x

Overview

n8n runs arbitrary code, holds credentials for every service you connect, and usually sits on a public hostname. That combination makes it a high-value target. n8n 2.0 changed the picture considerably: several settings that used to be opt-in are now secure by default.

This chapter covers authentication and users, the 2.0 secure defaults and the cost of relaxing each, encryption, and then the infrastructure layers around n8n: network, secrets, hosts, identities, scanning, logging, CI/CD, and compliance.


Edge auth: Cloudflare Access, oauth2-proxy, Tailscale
n8n auth: owner, admin, member, 2FA
2.0 secure defaults
Encryption: TLS, key, at-rest
Network segmentation



Diagram

Prerequisites

- A running n8n 2.x deployment on the reference stack: `n8nio/n8n:2.38.1`, `n8nio/runners:2.38.1`, PostgreSQL, and an external task runner sidecar.
- TLS already terminating in front of n8n. See the chapter “Reverse Proxy and HTTPS: Caddy, Nginx, and Traefik” or “Kubernetes: Ingress and Let’s Encrypt”.
- Permission to change firewall rules, cloud IAM, Kubernetes RBAC, and CI/CD settings.

---

1. Authentication and Users

1.1 What n8n's authentication is, and what it is not

n8n has no basic auth: `N8N_BASIC_AUTH_USER` and `N8N_BASIC_AUTH_PASSWORD` were removed in n8n 1.0, so a config that sets them predates user management and should not be copied.

What n8n has instead is built-in user management: per-user accounts with email-and-password login, roles, and two-factor authentication. That is real authentication, and it is in the Community Edition.

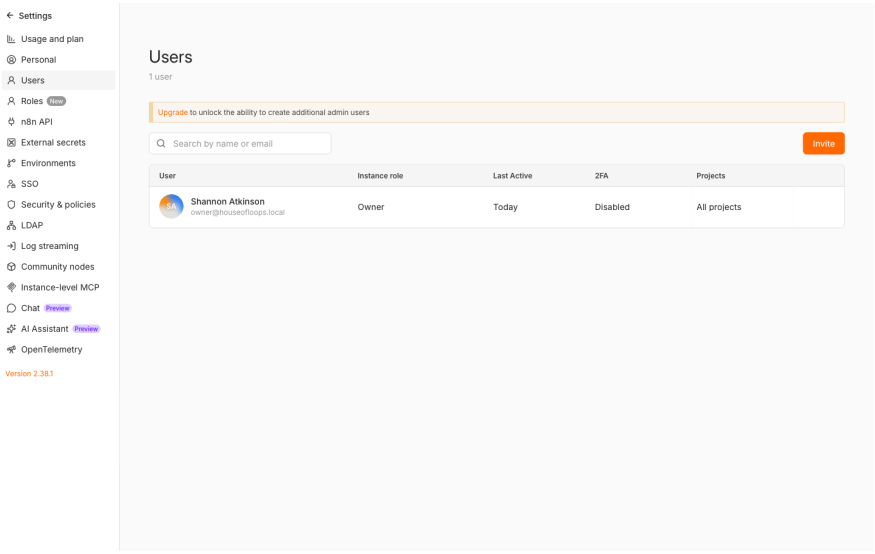
1.2 The owner account

The first person to open the editor on a fresh instance is prompted to create the **instance owner**. Until that happens, the setup screen is reachable by anyone who can hit the URL, so create the owner account immediately after the first start — before you point DNS at the instance or open a firewall port. There is exactly one owner; the account cannot be deleted and controls instance settings.

You can also pre-provision the owner so a fresh deployment never has an open setup screen. Set `N8N_INSTANCE_OWNER_MANAGED_BY_ENV=true` and supply `N8N_INSTANCE_OWNER_EMAIL`, `N8N_INSTANCE_OWNER_FIRST_NAME`, `N8N_INSTANCE_OWNER_LAST_NAME`, and `N8N_INSTANCE_OWNER_PASSWORD_HASH` — a bcrypt hash, because a plaintext password there breaks login. n8n then rewrites the owner record on every start and locks the matching UI control.

1.3 Inviting users

Go to **Settings** → **Users** → **Invite**. Invites go out over SMTP, so configure it:



Settings → Users on a fresh 2.38 instance: one owner, invite button, roles per user.

```
N8N_EMAIL_MODE=smtp
N8N_SMTP_HOST=smtp.example.com
N8N_SMTP_PORT=587
N8N_SMTP_USER=n8n@example.com
N8N_SMTP_PASS=replace-me
N8N_SMTP_SENDER=n8n <n8n@example.com>
N8N_SMTP_SSL=false
N8N_SMTP_STARTTLS=true
```

Without SMTP you can still invite people: n8n shows a copyable invite link to hand over through a channel you trust. Treat that link as a credential — anyone holding it can create an account on your

instance. `N8N_INVITE_LINKS_EMAIL_ONLY=true` stops invite URLs being readable through the API at all, closing the path where a high-privilege API token harvests pending invites, at the cost of making SMTP mandatory.

1.4 Roles in the Community Edition

The Community Edition ships three instance-level roles:

- **Owner** — one per instance, full control including instance settings and user management.
- **Admin** — user management and most instance settings, without owner-only controls.
- **Member** — builds and runs workflows in their own space, plus anything shared with them.

That is genuine role separation and it is free. Give people the lowest role that lets them work, and keep the owner account for a small number of operators rather than making it a shared login.

Three things are **not** in the Community Edition and are all paid-plan features: **SSO via SAML or OIDC**, **LDAP directory sync**, and **Projects with fine-grained RBAC** (project-scoped roles and per-project credential sharing). Do not plan a Community Edition rollout around any of them.

1.5 Enforcing two-factor authentication

2FA is available in the Community Edition and enabled as a capability by default (`N8N_MFA_ENABLED` is `true`); individual users switch it on in their personal settings. That per-user, opt-in 2FA is genuinely Community Edition.

To require it for everyone rather than hoping people opt in, use the environment-managed security policy. Both variables are needed — the first turns on env-managed policy and locks the matching UI controls, the second sets the value:

```
N8N_SECURITY_POLICY_MANAGED_BY_ENV=true
N8N_MFA_ENFORCED_ENABLED=true
```

Instance-wide enforcement through these two variables is a separate claim from user-level 2FA, and it may require a paid plan rather than Community Edition — licensing for environment-managed security policies has moved between releases. Confirm current availability on the two-factor authentication docs page before you build a rollout around it; do not assume it's free just because per-user 2FA is.

Enforce this before you invite anyone. Turning it on later locks out every user until they re-enroll, which is survivable but noisy.

1.6 Sessions and cookies

n8n issues a JWT in a cookie. Three settings matter:

```
# Persist sessions across restarts and share them across containers.
N8N_USER_MANAGEMENT_JWT_SECRET=replace-with-a-long-random-string
# Session lifetime in hours. Default 168 (one week).
N8N_USER_MANAGEMENT_JWT_DURATION_HOURS=24
# Hours before expiry to auto-refresh. 0 means 25% of the duration.
# -1 never refreshes, forcing a real login every duration window.
N8N_USER_MANAGEMENT_JWT_REFRESH_TIMEOUT_HOURS=-1
```

If you do not set `N8N_USER_MANAGEMENT_JWT_SECRET`, n8n generates one at startup. On a single container that only logs everyone out on restart. In queue mode, where the main instance and webhook processors all validate cookies, a per-process secret breaks sessions unpredictably. Set it explicitly and identically everywhere.

`N8N_SECURE_COOKIE` defaults to `true`, marking the session cookie `Secure` so browsers only send it over HTTPS. Keep it that way. `N8N_SECURE_COOKIE=false` is the documented workaround for reaching n8n over plain HTTP on a non-localhost address, and it means the session cookie travels in clear text

where anyone on the path can capture it and become that user. Acceptable for a few minutes of testing on a private network, never in production.

`N8N_SAMESITE_COOKIE` defaults to `lax`. Set it to `strict` if nothing legitimately navigates into n8n from another site, or `none` only if you embed the editor — `none` requires HTTPS.

1.7 The second auth layer belongs in front of n8n

People often want “another password on the whole thing”. A second shared password inside n8n is the wrong shape: no audit trail, one more secret to rotate.

The correct pattern is an identity-aware proxy in front of n8n. n8n keeps its own user accounts; the proxy decides who reaches the port at all. Three practical choices: **Cloudflare Access**, which needs no components of your own and locks the origin to Cloudflare; **oauth2-proxy**, self-hosted between your reverse proxy and n8n against any OIDC provider; and **Tailscale**, where the instance is never exposed publicly and access is per-device.

Here is oauth2-proxy in front of the reference stack. Only the proxy is published on the host:

```yaml
services:
  oauth2-proxy:
    image: quay.io/oauth2-proxy/oauth2-proxy:v7.7.1
    restart: unless-stopped
    ports:
      - "127.0.0.1:4180:4180"
    environment:
      OAUTH2_PROXY_HTTP_ADDRESS: "0.0.0.0:4180"
      OAUTH2_PROXY_PROVIDER: oidc
      OAUTH2_PROXY_OIDC_ISSUER_URL: https://idp.example.com/
      OAUTH2_PROXY_CLIENT_ID: ${OAUTH_CLIENT_ID}
      OAUTH2_PROXY_CLIENT_SECRET: ${OAUTH_CLIENT_SECRET}
      OAUTH2_PROXY_COOKIE_SECRET: ${OAUTH_COOKIE_SECRET}
      OAUTH2_PROXY_EMAIL_DOMAINS: example.com
      OAUTH2_PROXY_UPSTREAMS: http://n8n:5678
      OAUTH2_PROXY_REVERSE_PROXY: "true"
      # The editor uses websockets; the proxy must not buffer or strip them.
      OAUTH2_PROXY_SKIP_PROVIDER_BUTTON: "true"
    depends_on:
      - n8n
```

Your TLS-terminating reverse proxy then points at `127.0.0.1:4180` instead of `127.0.0.1:5678`, and the n8n service drops its `ports:` block entirely.

Two caveats. Webhooks and OAuth callbacks from third-party services cannot pass an interactive login, so exempt `/webhook`, `/webhook-test`, and `/rest/oauth2-credential/callback` at the proxy, or route them through a separate hostname that skips the identity layer. And the editor uses websockets, so whatever sits in front must pass `Upgrade` and `Connection` headers through.

---

2. The 2.0 Secure Defaults and What Relaxing Each Costs

n8n 2.0 flipped a set of defaults from permissive to safe. You can turn any of them off; this section says what you give up when you do.

Task runners in external mode. Since 2.0 every Code node executes inside a task runner rather than the main n8n process, and external mode runs that runner as a separate `n8nio/runners` container. This is the only isolation boundary between user-authored code and your n8n process, its memory, and its filesystem. `internal` mode runs the runner as a child process on the same host: fine for a laptop, not for production. The chapter “Task Runners and Safe Code Execution” covers setup, the shared `N8N_RUNNERS_AUTH_TOKEN`, and the rule that in queue mode every worker needs its own sidecar. Beyond that, run the runner as the unprivileged `nobody` user (UID/GID 65532), give it a read-only root filesystem with a small `emptyDir` at `/tmp`, use the `-distroless` image variant, and apply an AppArmor profile denying reads of per-process `/proc` files:

```
audit deny @{PROC}/[0-9]*/{environ,mounts} rwl,
```

Those files would otherwise leak environment variables — including secrets — to code running in the container.

`N8N_BLOCK_ENV_ACCESS_IN_NODE` now defaults to `true`, so Code nodes and expressions can no longer read the container’s environment. That environment holds `N8N_ENCRYPTION_KEY`, database passwords, and the runner auth token — anyone who could author a workflow could previously exfiltrate all of them in three lines. Setting it to `false` restores that exposure to every workflow author. If a workflow needs a value, put it in a credential.

`NODES_EXCLUDE` disables `ExecuteCommand` and `LocalFileTrigger` by default. `ExecuteCommand` runs shell commands as the n8n process; `LocalFileTrigger` watches arbitrary paths. Re-enabling either turns “can build a workflow” into “has a shell on the host”. If you must have one, exclude only the other rather than clearing the list:

```
# Re-enable LocalFileTrigger only. Do not set NODES_EXCLUDE="[]".
NODES_EXCLUDE=["n8n-nodes-base.executeCommand"]
```

`N8N_RESTRICT_FILE_ACCESS_TO` defaults to `~/.n8n-files`, so the Read/Write File and Read Binary Files nodes can only touch that directory. Widening it — it takes a semicolon-separated list — lets a workflow author read anything the n8n process can read, which usually includes the volume holding the encryption key. Mount a dedicated volume at `~/.n8n-files` and leave the variable alone. `N8N_BLOCK_FILE_ACCESS_TO_N8N_FILES` defaults to `true` and blocks the `.n8n` directory specifically; leave that alone too.

`N8N_SKIP_AUTH_ON_OAUTH_CALLBACK` flipped from `true` to `false`, so OAuth callback endpoints now require authentication instead of accepting unauthenticated callers. If an OAuth connection breaks after upgrading, complete it in a browser session where you are logged in rather than setting this back to `true`.

`N8N_ENFORCE_SETTINGS_FILE_PERMISSIONS` defaults to `true` and requires `0600` on n8n’s settings file, the same discipline SSH applies to private keys. That file holds the instance’s encryption key when you have not supplied one by environment variable. Set it to `false` only on filesystems that cannot express Unix permissions, such as a Windows bind mount, where any local process can then read the key.

`N8N_GIT_NODE_DISABLE_BARE_REPOS` defaults to `true`. Bare repositories can carry hooks and config that execute on clone. Setting it to `false` re-opens that path; if you do, confirm `N8N_GIT_NODE_ENABLE_HOOKS` stays at its `false` default.

`N8N_RUNNERS_INSECURE_MODE` re-enables the JavaScript features the runner disables, principally string-to-code evaluation. It exists because `$evaluateExpression()` stopped working in the Code node in 2.0. Never set it in production: write the logic in JavaScript directly, or evaluate the expression in an Edit Fields (Set) node before the Code node and read the result from the incoming item.

`N8N_DIAGNOSTICS_ENABLED=false` is not a 2.0 default but belongs here: it stops outbound usage telemetry. On an instance handling regulated data, or in an egress-restricted network, turn it off deliberately rather than discovering the traffic in a firewall log.

2.1 What n8n 3.0 changes

n8n 3.0 is scheduled for October 2026 and tightens security defaults again. Two changes need planning now:

- **Key rotation is on by default.** Plan how your deployment handles rotated keys before you upgrade; do not assume your current key handling survives untouched.
- **Compression node limits drop.** `N8N_COMPRESSION_NODE_MAX_DECOMPRESSED_SIZE_BYTES` falls from 2 GiB to 256 MiB and `N8N_COMPRESSION_NODE_MAX_ZIP_ENTRIES` from 5,000 to 1,000. These are zip-bomb defences. If you legitimately decompress larger archives, set the old values explicitly before upgrading:

```
N8N_COMPRESSION_NODE_MAX_DECOMPRESSED_SIZE_BYTES=2147483648
N8N_COMPRESSION_NODE_MAX_ZIP_ENTRIES=5000
```

Raise them only as far as your real archives require. A 256 MiB ceiling turns a decompression bomb into a failed node instead of an out-of-memory kill.

---

3. Encryption

3.1 `N8N_ENCRYPTION_KEY`

This key encrypts every credential in the database. Get its handling right and most credential-loss incidents never happen.

Generate it once:

```
openssl rand -hex 32
```

Store it in a secrets manager, not in a Compose file committed to Git. Vault, AWS Secrets Manager, GCP Secret Manager, and Azure Key Vault are all fine; a password manager entry is the minimum for a small deployment. n8n supports `_FILE`-suffixed variables, so `N8N_ENCRYPTION_KEY_FILE` can point at a mounted secret file instead of putting the value in the environment.

Set it identically on every container that touches the database — the main instance, every worker, every webhook processor. A worker with a different key decrypts nothing.

Back it up alongside the database. A restore without the matching key gives you a complete set of unreadable credentials, which is the same as having no backup. See the chapter “Backup, Restore, and Disaster Recovery”.

Never rotate it by editing the variable. Changing the key does not re-encrypt anything; it orphans every existing credential. If you must rotate after a suspected exposure, follow n8n’s documented rotation procedure for your version. n8n 3.0 enables key rotation by default, which changes this picture; check the release notes at upgrade time.

3.2 TLS everywhere

Terminate TLS at the edge — ingress controller, load balancer, or reverse proxy — and never expose port 5678 directly. In the reference stack the port is published as `127.0.0.1:5678:5678` so nothing off-host can reach it. Restrict protocols and ciphers at the terminator:

```
ssl_protocols TLSv1.2 TLSv1.3;
ssl_ciphers 'HIGH:!aNULL:!MD5';
ssl_prefer_server_ciphers on;
add_header Strict-Transport-Security "max-age=31536000; includeSubDomains" always;
```

Behind any proxy, set `N8N_HOST`, `N8N_PROTOCOL=https`, `WEBHOOK_URL`, `N8N_EDITOR_BASE_URL`, and `N8N_PROXY_HOPS=1` so n8n builds correct URLs and reads the real client IP.

3.3 mTLS and encryption at rest

In a service mesh (Istio, Linkerd), enable mutual TLS between n8n, its workers, PostgreSQL, and Redis; sidecars handle certificate issuance and rotation. Without a mesh, at minimum require TLS on the database connection and set `QUEUE_BULL_REDIS_TLS=true` for Redis.

- **PostgreSQL** — use the managed service’s encryption, or LUKS on the volume for self-managed instances.
- **Redis** — TLS in transit plus disk encryption; queue payloads can contain workflow data.
- **Kubernetes Secrets** — enable envelope encryption with a KMS provider. Base64 is not encryption.
- **Object and file storage** — enable at-rest encryption on the bucket or file share used for binary data in `s3` mode.

---

4. Network Segmentation and Policies

Put n8n, PostgreSQL, and Redis in private subnets with no public IPs. Reach them through a bastion host or VPN. The only thing with a public address should be the load balancer or reverse proxy that terminates TLS.

4.1 Kubernetes NetworkPolicies

Default-deny, then allow only what is needed. This is an allowlist pattern, not a finished policy — a real n8n deployment needs more than Postgres and DNS on egress: queue mode needs Redis, OpenTelemetry needs the OTLP collector, and any workflow with an HTTP Request node, an OAuth credential, or an outbound webhook needs the internet on 443. Enumerate your own destinations rather than copying the block below as-is; it shows the shape, with placeholders commented out for the egress rules most deployments end up needing:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: restrict-n8n-network
  namespace: n8n
spec:
  podSelector:
    matchLabels:
      app: n8n
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              role: ingress
      ports:
        - protocol: TCP
          port: 5678
    - from:
        - podSelector:
            matchLabels:
              app: n8n-runners
      ports:
        - protocol: TCP
          port: 5679
  egress:
    # DNS - needed for any outbound name resolution at all.
    - to:
        - namespaceSelector: {}
      ports:
        - protocol: UDP
          port: 53
        - protocol: TCP
          port: 53
    # PostgreSQL
    - to:
        - podSelector:
            matchLabels:
              app: postgres
      ports:
        - protocol: TCP
          port: 5432
    # Redis - uncomment if you're running queue mode, pointed at your Redis pod.
    # - to:
    #     - podSelector:
    #         matchLabels:
    #           app: redis
    #   ports:
    #     - protocol: TCP
    #       port: 6379
    # OTLP collector - uncomment if you export traces or metrics, pointed at
    # your collector.
    # - to:
```

```
#     - podSelector:
#         matchLabels:
#           app: otel-collector
#   ports:
#     - protocol: TCP
#       port: 4317
# The internet on 443, for HTTP Request nodes, OAuth token exchanges, and
# any workflow that calls out to a third-party API — uncomment and scope
# this as tightly as your workflows actually require.
# - to:
#     - ipBlock:
#         cidr: 0.0.0.0/0
#   ports:
#     - protocol: TCP
#       port: 443
```

List every egress destination your workflows genuinely need before you turn this on in a namespace already running production traffic — a default-deny policy that's missing a rule doesn't fail loudly, it just makes a node's HTTP call hang until it times out. Give task runner pods their own policy allowing egress only to the broker port on n8n. A runner that cannot reach the internet cannot exfiltrate what it reads.

On VMs and bare metal, use `iptables`, `nftables`, or `firewalld` to allow only 80, 443, and SSH from known addresses. Do not open 5678, 5679, 5432, or 6379 to anything outside the deployment.

---

5. Secrets Management

Store credentials in HashiCorp Vault's KV engine and inject them at deploy time — the Kubernetes Vault CSI driver mounts them as files, which pairs neatly with n8n's `_FILE` variables. Use dynamic secrets with a TTL for database credentials so a leaked value expires on its own. AWS Secrets Manager or Parameter Store, GCP Secret Manager, and Azure Key Vault work the same way: grant the n8n task or pod identity read access to exactly the secrets it needs and nothing else.

Never commit `.env` files or Kubernetes Secret manifests. Use SOPS-encrypted YAML or Sealed Secrets, add `.env` to `.gitignore` on day one, and add a pre-commit secret scanner (`gitleaks`, `detect-secrets`) so the rule is enforced rather than remembered.

---

6. Host and Container Hardening

Pin exact versions — `n8nio/n8n:2.38.1`, `n8nio/runners:2.38.1`, `postgres:18`. Never deploy a floating tag: you cannot roll back to a version you cannot name, and you cannot tell an auditor what is running. Keep the n8n image and the runners image on the same version; they speak a versioned protocol. For task runners, prefer the `-distroless` variant — `2.38.1-distroless` — which ships the runtime without a shell or package manager.

Run containers unprivileged, read-only, and with no capabilities. In Kubernetes, apply the `restricted` Pod Security Standard to the namespace and set a matching `securityContext`:

```
securityContext:
  runAsNonRoot: true
  runAsUser: 65532
  allowPrivilegeEscalation: false
  readOnlyRootFilesystem: true
  capabilities:
    drop: ["ALL"]
  seccompProfile:
    type: RuntimeDefault
```

Keep the host patched and minimal, disable unused services, and enable AppArmor or SELinux. If workflows run untrusted code, put task runners on dedicated nodes, optionally under gVisor.

---

7. Least Privilege at the Infrastructure Layer

n8n's own roles govern who can do what *inside* n8n. Infrastructure roles govern who can reach the container, read the database, or exec into the pod. Both are needed.

Grant the n8n cloud identity only what it uses: read on specific secrets, connect on the specific database, read/write on the specific bucket. No wildcards, no administrator roles. In Kubernetes, bind a narrow namespaced Role:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: n8n
  name: n8n-operator
rules:
  - apiGroups: [""]
    resources: ["pods", "services"]
    verbs: ["get", "list"]
```

Bind it to a dedicated ServiceAccount — every workload gets its own, never the namespace default — and restrict `pods/exec` tightly. A `kubectl exec` into an n8n pod reads the encryption key straight out of the environment.

---

8. Vulnerability Scanning

Scan images in CI and fail the build above your threshold:

```
trivy image --severity HIGH,CRITICAL --exit-code 1 n8nio/n8n:2.38.1
```

Audit any custom nodes with `npm audit --audit-level=high`, and treat n8n security releases as expedited changes. Because you pin exact tags, upgrading is a one-line diff and a `docker compose up -d`.

---

9. Audit Logging and Monitoring

Run n8n's built-in security audit from the CLI or API to surface unused credentials, abandoned workflows, and risky node usage; `N8N_SECURITY_AUDIT_DAYS_ABANDONED_WORKFLOW` (default 90) sets the abandonment threshold. Enable Kubernetes API server audit logging to capture who exec'd into what, and PostgreSQL audit logging (`pgaudit`) for DDL and DML on the n8n database. Ship everything to your own centralized logging and monitoring stack, and alert on failed-login spikes, new credential creation, and workflow changes outside business hours.

---

10. CI/CD Security

Scan Dockerfiles and Compose files for misconfiguration — root users, unpinned tags, published database ports. Run static analysis on custom nodes, run CI jobs in ephemeral minimally privileged runners rather than on the n8n host, and require reviews and signed commits on the branch that deploys:

```
required_status_checks:
  strict: true
  contexts:
    - "CI build"
    - "Trivy Scan"
    - "Secret scan"
require_pull_request_reviews:
  required_approving_review_count: 2
enforce_admins: true
```

See the chapter “CI/CD and GitOps” for the pipeline itself.

---

11. Compliance and Regular Reviews

Write down a security policy covering patch management, incident response, and access control. Then schedule quarterly reviews that actually happen: rotate credentials and certificates (rotating `N8N_ENCRYPTION_KEY` only through the documented procedure); prune the n8n user list; re-check the 2.0 defaults in Section 2 against the running configuration, because drift accumulates; review IAM and RBAC bindings; restore a backup and confirm credentials decrypt; and run the vulnerability scan. Export logs and change history as evidence for HIPAA, GDPR, or PCI DSS audits.

---

12. Troubleshooting Common Issues

Everyone is logged out after every restart, or login works on some requests and not others in queue mode. `N8N_USER_MANAGEMENT_JWT_SECRET` is unset, so each process generates its own. Set it explicitly and identically on the main instance, every worker, and every webhook processor.

The browser refuses to keep you logged in over HTTP. `N8N_SECURE_COOKIE` defaults to `true` and the browser drops a `Secure` cookie on an insecure origin. Put TLS in front. Only for short local testing, set `N8N_SECURE_COOKIE=false` and accept that the session cookie is now readable on the wire.

Credentials show as invalid after a restore. The encryption key does not match the one used when they were saved. Restore the key from your secrets manager; there is no recovery path without it.

A Code node throws on `$evaluateExpression()`. Expected since 2.0. Rewrite the logic in JavaScript, or evaluate in an Edit Fields (Set) node upstream. Do not reach for `N8N_RUNNERS_INSECURE_MODE`.

A Read/Write File node reports access denied. The path is outside `N8N_RESTRICT_FILE_ACCESS_TO`, which defaults to `~/.n8n-files`. Move the file, or mount your data volume at that path.

The Execute Command node is missing from the palette. Disabled by default in 2.0 through `NODES_EXCLUDE`. Re-enable it only with a clear justification, and re-exclude `LocalFileTrigger` explicitly rather than clearing the whole list.

An OAuth connection fails at the callback. `N8N_SKIP_AUTH_ON_OAUTH_CALLBACK` now defaults to `false`. Complete the connection from a logged-in browser session, and exempt the callback path at any identity-aware proxy.

Users bypass 2FA. `N8N_MFA_ENFORCED_ENABLED=true` has no effect without `N8N_SECURITY_POLICY_MANAGED_BY_ENV=true`. Both are required.

A Compression node fails on a large archive after upgrading to 3.0. The limits dropped to 256 MiB and 1,000 entries. Raise the two variables only if your archives are genuinely larger; otherwise the defence is working.

Further reading

- https://docs.n8n.io/changelog/v20-breaking-changes
- https://docs.n8n.io/changelog/v30-breaking-changes
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/security
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/user-management-and-2fa
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/deployment
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/security/harden-task-runners
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/user-management
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/manage-settings-using-environment-variables

---

Chapter 32: Workflow Versioning and Publishing

Overview

n8n 2.0 replaced the active/inactive toggle with Publish and Unpublish. The distinction matters for anything you do with Git: a workflow is no longer “on” the moment you save it. You edit freely, and every time you click Publish, n8n stores that state as a new version and makes it the live one. Workflow History keeps every version, and since n8n 2.11 you can compare two versions side by side directly in the editor — a lot of what this chapter used to need Git for just to answer “what changed” is now a built-in view.

None of that replaces Git for the reasons this book cares about: a record that survives outside n8n’s own database, a review step before a change reaches production, and a way to promote the same workflow across separate instances. This chapter builds a DIY export-to-Git pipeline that does all three, using n8n’s CLI and REST API, and updates every command for 2.0’s publish/version model.

This chapter covers:

1. Exporting workflows to JSON, by hand and by CLI
2. Structuring a Git repository for workflow files
3. Automating exports into commits
4. CI validation on pull requests
5. Branching strategies
6. Tagging and release notes
7. Workflow metadata
8. Secrets and credentials, kept out of Git
9. Merge conflicts in JSON
10. Where n8n’s own Source Control (Git) and Environments features fit instead of this chapter

Diagram

Prerequisites

- Shell access to the n8n container, to run CLI commands, or an API key for the REST API.
- A Git repository dedicated to workflow JSON.
- A CI pipeline capable of running a container and a short script.
- Familiarity with JSON, Git branches, and pull requests.

1. Exporting Workflows to JSON

1.1 Manual export via the editor

Open a workflow, use the menu's Download option, and save the file under your repository, for example `workflows/customer-onboarding.json`. This is fine for a one-off but doesn't scale past a handful of workflows.

1.2 CLI export

```
docker compose exec n8n n8n export:workflow --all --output=/home/node/exports/
```

`export:workflow` is unchanged by the 2.0 publishing model — it exports whatever is currently saved on a workflow, published or not, along with its `versionId` in the JSON. To export a single workflow by ID:

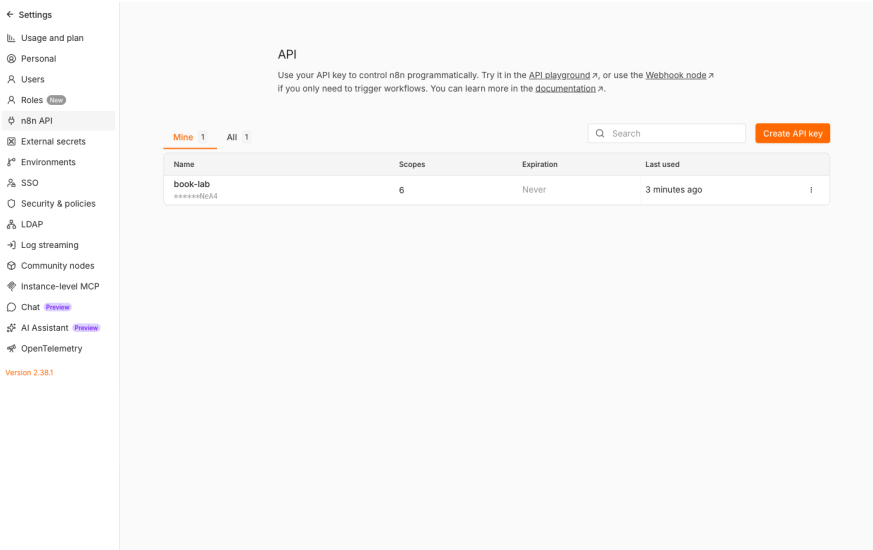
```
docker compose exec n8n n8n export:workflow --id=<workflow-id> --output=/home/node/exports/<workflow-
        id>.json
```

Copy the files out of the container (or mount `/home/node/exports` as a volume) before your CI step commits them.

1.3 Export via the REST API

```
curl -H "X-N8N-API-KEY: ${N8N_API_KEY}" \
  "https://n8n.example.com/api/v1/workflows" \
  | jq '.data[] | {id, name, versionId}' \
  > workflows/index.json
```

Use an n8n API key (Settings → API), never the owner account's password, for anything a script runs unattended. Fetch a single workflow's full definition at `/api/v1/workflows/:id`.



Settings → API: create a scoped key for scripts and pipelines. Keys carry scopes and an optional expiry.

2. Structuring Your Git Repository

```
n8n-workflows/
├── workflows/
│   ├── customer-onboarding.json
│   ├── invoice-processing.json
│   └── marketing-email.json
├── credentials/
│   └── README.md          # names and provisioning notes only, never secret values
├── scripts/
│   └── export-workflows.sh
├── .github/
│   └── workflows/ci.yaml
└── README.md
```

Store only workflow definitions under `workflows/`. Never commit credential values — n8n encrypts them with `N8N_ENCRYPTION_KEY` in the database, and an exported workflow JSON references credentials by ID and name, not by secret value, so there's nothing sensitive in a clean export unless someone hand-edits one in.

---

3. Automating Workflow Exports

3.1 Scheduled CI job

```yaml
# .github/workflows/cron-export.yaml
name: Scheduled Workflow Export

on:
  schedule:
    - cron: "0 3 * * *"

jobs:
  export:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Export workflows
        run: |
          docker run --rm \
            -e N8N_API_KEY=${{ secrets.N8N_API_KEY }} \
            -v ${{ github.workspace }}/workflows:/out \
            n8nio/n8n:2.38.1 \
            n8n export:workflow --all --output=/out/
      - name: Commit and push
        run: |
          git config user.name "n8n Automation"
          git config user.email "n8n@ci"
          git add workflows/
          git commit -m "Automated workflow export [skip ci]" || echo "No changes"
          git push
```

This captures whatever is currently saved, including unpublished drafts — treat this repository as a mirror of what's saved in the instance, not a record of what's live. Section 4 below is what actually gates production.

---

4. CI Validation and Pull Requests

4.1 Validate JSON on every pull request

```yaml
# .github/workflows/ci.yaml
jobs:
  validate:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Validate workflow JSON
        run: |
          for file in workflows/*.json; do
            jq empty "$file" || { echo "Invalid JSON: $file"; exit 1; }
            docker run --rm -v "$PWD/$file:/tmp/wf.json" n8nio/n8n:2.38.1 \
              n8n import:workflow --input=/tmp/wf.json --dry-run
          done
```

`import:workflow --dry-run` checks that the file parses as a valid n8n workflow without writing it to any database.

4.2 Publishing after merge

Once a pull request merges to `main`, the deploy step imports and publishes the changed workflows on the target instance:

```
docker compose exec n8n n8n import:workflow --input=/home/node/exports/invoice-processing.json
docker compose exec n8n n8n publish:workflow --id=<workflow-id>
```

To publish a specific version rather than whatever was just imported — useful if the target instance already has the workflow and you're rolling forward to a version you exported earlier:

```
docker compose exec n8n n8n publish:workflow --id=<workflow-id> --versionId=<version-id>
```

There is no `--all` flag on `publish:workflow`, deliberately — n8n 2.0 removed it so a script can't mass-publish every workflow on a production instance by accident. If you need to publish several workflows, loop over their IDs explicitly. `unpublish:workflow` keeps `--all`, since turning everything off in an emergency is a legitimate thing to want:

```
docker compose exec n8n n8n unpublish:workflow --id=<workflow-id>
docker compose exec n8n n8n unpublish:workflow --all
```

The old `update:workflow` command, which both edited and (with `--all`) activated workflows in one step, is gone in 2.0. Anything in a CI pipeline still calling it needs to move to the `import:workflow` / `publish:workflow` pair above.

4.3 Preview instances

For a change worth testing before it reaches production, deploy the pull request's branch to a throwaway n8n instance (a separate Compose stack, or an ephemeral deploy on one of the platforms from "Platform-as-a-Service: Render, Railway, and Fly.io") and import the changed workflows there, unpublished, so a reviewer can open them and inspect the diff in the editor before approving.

---

5. Branching Strategies

5.1 GitFlow-like

- `main` — production-ready workflows, exactly what's published.
- `develop` — integration branch for work in progress.
- `feature/*` — one branch per new or significantly changed workflow.
- `release/*`, `hotfix/*` — as needed for staged rollouts and urgent fixes.

5.2 Trunk-based

`main` only, small workflow changes, short-lived feature branches merged within a day or two. Because n8n itself now keeps every published version in Workflow History, trunk-based teams get a usable safety net even without a `release/*` branch — reverting a bad publish can mean picking a prior version in the editor rather than reverting Git and redeploying.

---

6. Tagging and Releasing

```
git tag -a v1.3.0 -m "Release workflow set 1.3.0"
git push origin v1.3.0
```

Use semantic versioning for what the tag communicates about the workflow set, not for n8n's own per-workflow `versionId`, which increments independently every time a workflow is published:

- **Major** — a workflow's behavior changes in a way that breaks callers or downstream data.
- **Minor** — a new workflow is added.
- **Patch** — a bugfix within an existing workflow.

```
# Changelog

## [1.3.0] - 2026-09-05
### Added
- New "Customer Re-engagement" workflow.
### Fixed
- Corrected error handling in "Invoice Processing".
```

---

7. Workflow Metadata

n8n's own version history already records who published what and when — Workflow History shows the author and timestamp for every version, and 2.11's compare view shows exactly what changed between two of them. Use the workflow JSON's `settings` object only for metadata n8n doesn't track itself:

```
{
  "name": "Invoice Processing",
  "nodes": [ ],
  "settings": {
    "owningTeam": "finance-eng",
    "environment": "production"
  }
}
```

Don't duplicate a `version` field here — it invites drift against the real `versionId` n8n assigns, and against the git tag.

---

8. Secrets and Credentials

Credentials never leave n8n's encrypted store through a normal export — a workflow JSON references a credential by ID and name only. Two things to keep out of Git regardless:

1. `N8N_ENCRYPTION_KEY` and any API keys used by your export/publish scripts — store these as CI secrets, not repository files.
2. A written record, in `credentials/README.md`, of what each referenced credential is and how it's provisioned on a fresh instance — so restoring from this repository onto a new instance is a checklist, not archaeology.

---

9. Merging and Conflict Resolution

Two people editing the same workflow produce a JSON diff that's hard to read and easy to merge wrong by hand. Two mitigations:

```
workflows/*.json merge=json
```

in `.gitattributes`, paired with a merge driver in `.git/config`:

```
[merge "json"]
  name = JSON merge driver
  driver = jq -s '.[0] * .[1] * .[2]' %O %A %B > %A
```

This is a shallow object-merge, good enough for non-overlapping metadata changes and nothing more — a real structural conflict (both sides editing the same node) still needs a human to open both versions in the editor's own compare view and reconcile visually. That comparison view, not a JSON diff tool, is the fastest way to actually understand what a workflow-level conflict is.

---

10. This Chapter vs. n8n's Paid Source Control and Environments

Everything above is a DIY pipeline built from the CLI, the REST API, and a Git repository you own — it works on Community Edition with no additional license. n8n also sells two features that overlap with parts of it: **Source Control (Git)**, which connects an instance directly to a Git repository and pushes/pulls workflow changes through the editor UI without a custom export script, and **Environments**, which promotes a set of workflows between separate instances (development, staging, production) with built-in variable substitution per environment. Both are paid-plan features, not part of Community Edition. If your team outgrows the scripts in this chapter — reviewers want in-editor diffs, promotion across instances happens often enough that scripting it feels like reinventing a product — that's the point to evaluate paying for Source Control and Environments rather than extending this pipeline further.

---

11. Troubleshooting Common Issues

Issue	Cause	Solution
`publish:workflow` fails with "command not found" or unexpected arguments	Still calling the removed `update:workflow`	Replace with `import:workflow` followed by `publish:workflow --id=<id>`
A workflow imports but never goes live	`import:workflow` only saves the workflow; it does not publish it	Run `publish:workflow --id=<id>` as a separate step
CI publishes the wrong version	`--versionId` omitted or pointing at a stale export	Confirm the `versionId` field in the exported JSON matches the version you intend to publish
Two branches both touch the same workflow and the JSON merge driver produces garbage	The shallow `jq` merge driver can't reconcile overlapping node edits	Resolve manually using the editor's version compare view, then re-export
Credentials missing after importing onto a fresh instance	Credentials are never included in an export; only references to them are	Recreate credentials on the target instance first, using `credentials/README.md`, before importing workflows
`unpublish:workflow --all` used as a routine deploy step	Treating an emergency command as a normal part of the pipeline	Publish/unpublish specific IDs in normal operation; reserve `--all` for incident response
External hook `workflow.activeChange` never fires	Hook removed in 2.0	Rewrite the hook to listen for `workflow.published`, which fires whenever any version of a workflow is published

---

Further reading

- https://docs.n8n.io/changelog/v20-breaking-changes
- https://docs.n8n.io/build/understand-workflows/save-and-publish-workflows
- https://docs.n8n.io/api/
- https://docs.n8n.io/source-control-environments/source-control/

Chapter 33: Upgrading to n8n 2.0

Overview

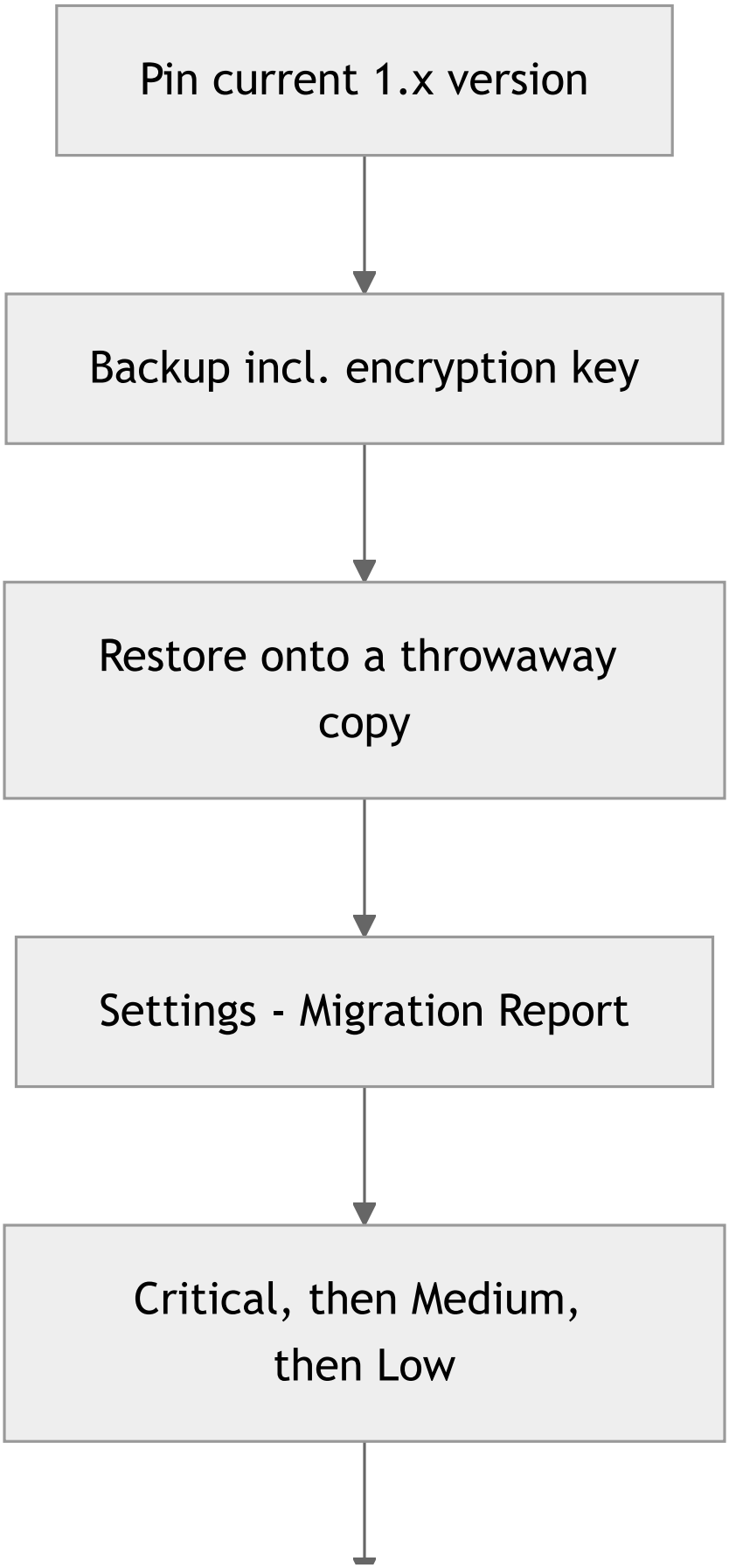
n8n 2.0 shipped on 8 December 2025. If you are still on any 1.x release, this chapter takes you from where you are to a pinned 2.x release on the book's reference Docker Compose stack.

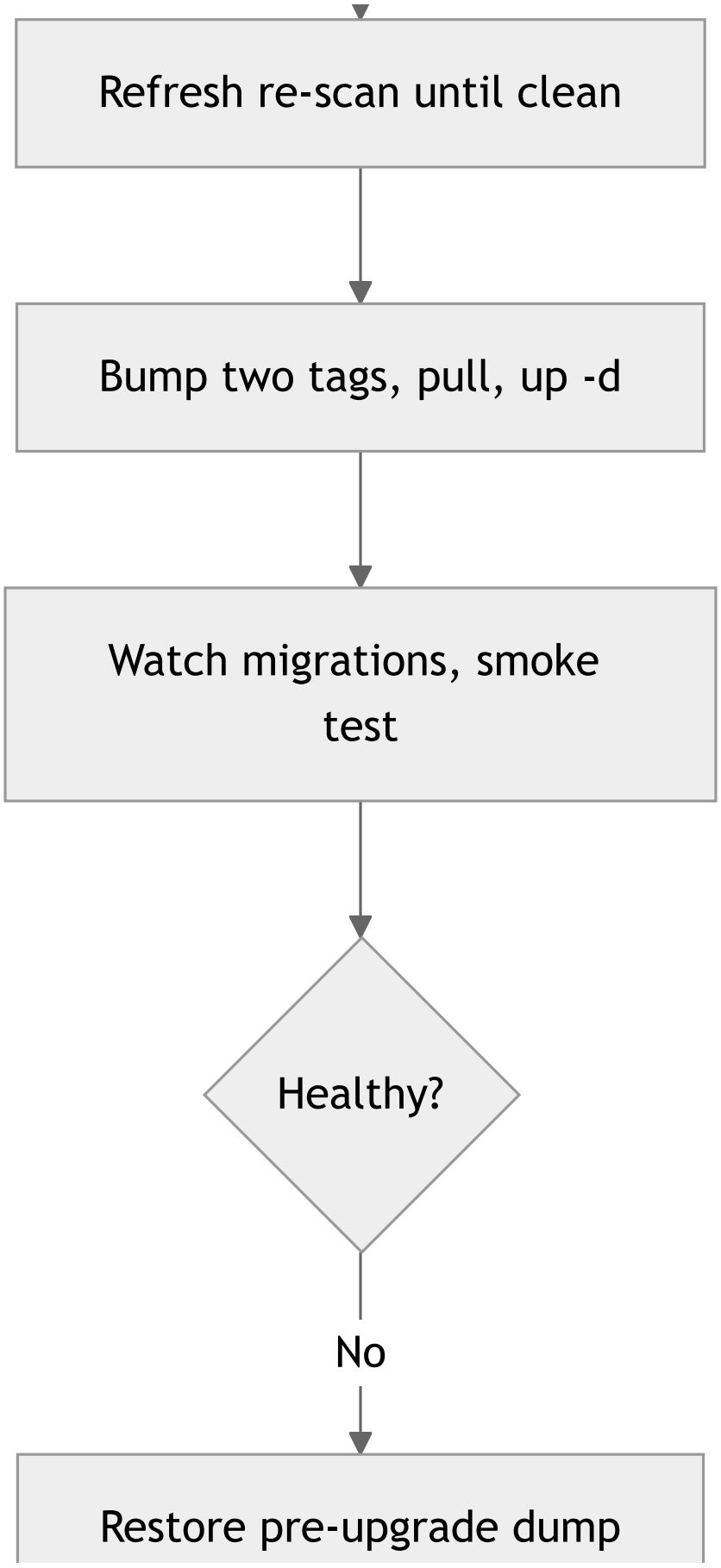
2.0 is a cleanup release, not a feature release. Almost everything in it is one of three things:

1. **Security defaults that flipped.** Opt-in hardening on 1.x is on by default in 2.0. Workflows relying on the loose default stop working until you fix the workflow or explicitly opt back out.
2. **Legacy code paths removed.** MySQL, the SQLite legacy driver, in-memory binary data, the Start node, `n8n --tunnel`, Pyodide Python, four nodes for services that no longer exist.
3. **Renames and replacements.** Publish/Unpublish instead of the active toggle, `publish:workflow` instead of `update:workflow`, `stable` / `beta` instead of `latest` / `next`.

So this is mostly a *configuration* migration. Most of the work is in your `.env` and in a handful of workflows, and n8n ships a scanner that tells you which ones.

One thing 2.0 does not change: there is still no basic auth. `N8N_BASIC_AUTH_USER` and `N8N_BASIC_AUTH_PASSWORD` were removed in 1.0, and 2.0 does not bring them back. If those variables are still sitting in your `.env`, delete them.


Pin current 1.x version
Backup incl. encryption key
Restore onto a throwaway copy
Settings - Migration Report
Critical, then Medium, then Low


Refresh re-scan until clean
Bump two tags, pull, up -d
Watch migrations, smoke test
Healthy?
No
Restore pre-upgrade dump

Diagram

Prerequisites

- A running self-hosted n8n on any 1.x release, with shell access to the host.
- Docker Compose v2 (`docker compose` , with a space).
- PostgreSQL as the n8n database. On MySQL or MariaDB, see section 7 — you must move off it **before** upgrading.
- A working backup and a tested restore, per the Backup, Restore, and Disaster Recovery chapter.
- Your `N8N_ENCRYPTION_KEY` , stored somewhere other than the server you are about to change.
- Global admin or owner access to the editor, so you can open the Migration Report.

---

1. Why 2.0 Matters

Your current version stops getting fixes once 2.x is the supported line, and n8n 3.0 lands in October 2026 with a further round of removals that assumes you already made this jump. Going 1.x straight to 3.x later means absorbing two sets of breaking changes at once, without the 2.0 migration scanner. Take the smaller step now.

Three changes drive most of the rest:

- **Every Code node now runs on a task runner.** Optional on 1.x, mandatory on 2.0. That one change causes three separate breaking changes: `$evaluateExpression` stops working, the runner binary leaves the main image, and Pyodide Python is replaced by native Python.
- **Sub-workflows that wait now return their real output.** A long-standing wrong answer was fixed, so workflows written around the wrong answer break.
- **Active/inactive becomes Publish/Unpublish, with versioning.**

---

2. Pre-Flight Checklist

Do all four, in order.

Pin your current version. If your compose file has a floating tag or none at all, you cannot roll back to what you were running, because you do not know what you were running:

```
docker compose exec n8n n8n --version
docker compose images n8n
```

Put that exact version in `compose.yml` and run `docker compose up -d` to confirm nothing changes. You now have a known starting point.

Take the the Backup, Restore, and Disaster Recovery chapter backup, including the encryption key. A database dump alone is not a backup of n8n. Credentials are encrypted with `N8N_ENCRYPTION_KEY`, and a dump restored without it gives you rows you cannot decrypt.

```
# Database
docker compose exec -T postgres pg_dump -U n8n -F c n8n > n8n-pre-2.0-$(date +%F).dump

# n8n data volume: settings file, filesystem binary data
docker run --rm \
  -v n8n_data:/data -v "$(pwd)":/backup \
  alpine tar czf /backup/n8n-data-pre-2.0-$(date +%F).tar.gz -C /data .

# Configuration, including the key
cp .env "env-pre-2.0-$(date +%F).bak"
```

Store the `.env` copy encrypted. It holds the key and the database password.

Test on a copy. Bring up a second stack under a different project name and restore the dump into it:

```
docker compose -p n8n-upgrade-test up -d postgres
docker compose -p n8n-upgrade-test exec -T postgres \
  pg_restore -U n8n -d n8n --clean --if-exists < n8n-pre-2.0-2026-09-05.dump
```

Give the copy the same `N8N_ENCRYPTION_KEY` but a different `N8N_HOST` and no public webhook URL, so nothing in it fires against real systems. Leave every workflow unpublished.

Read the Migration Report on the copy. That is the next section.

---

3. Using the Built-In Migration Report

n8n 1.x ships a scanner for exactly this upgrade. Go to **Settings** → **Migration Report**. It is visible to global admins only; a member account will not see the entry.

The top of the page reads *"X out of Y workflows are compatible with n8n 2.0."* Your job is to make X equal Y and to clear the instance issues as well.

Two tabs:

- **Workflow Issues** — breaking changes affecting specific workflows. Each row gives an issue title, a severity badge, a description, a documentation link, and a count of affected workflows. Click the *"N workflows affected"* count for the detail page: workflow name (click to open the editor), published state, the nodes affected (click one to open the editor on that node), total executions, last executed, last updated. The execution counts matter — a Critical issue on a workflow with 40,000 executions is your first job; one with two executions from last March may be easier to delete than to fix.
- **Instance Issues** — configuration changes that apply instance-wide. Same fields, minus the workflow count. These are the environment variables in sections 6, 7 and 8.

Severities mean what they say:

Severity	Meaning	When to fix
Critical	The workflow or instance will fail after upgrading	Before you upgrade. No exceptions.
Medium	May cause unexpected behavior, or needs attention soon	Before you upgrade, unless you have a reason.
Low	Deprecations and minor changes that will not break anything	After the upgrade is fine.

Follow the recommended order of work literally:

1. **Initial assessment.** Read the summary and skim both tabs before changing anything.
2. **Sort by severity.** Critical, then Medium, then Low.
3. **Fix workflow issues.** Open each issue, read its linked documentation, edit each affected workflow, test on the copy.
4. **Address instance issues.** Edits to `.env` or the compose environment block. Make them on the copy first.
5. **Verify.** Click **Refresh** to re-scan; if your build has no Refresh button, reloading the page re-scans too. Confirm the fixed issues are gone and the compatibility count matches your workflow count.
6. **Proceed.** Two empty tabs means the instance is ready.

A clean report is not a guarantee. The scanner sees workflow JSON and instance configuration; it cannot see what your Code nodes do at runtime, and it cannot see a `.env` quoting problem. It narrows the search — it does not replace the smoke test in section 12.

---

4. The Breaking-Change Checklist

Sections 5 through 12 in one table, grouped the way the n8n docs group them.

Group	Change	Affected if	Action
Behavior	Waiting sub-workflow return values	Parent waits for a child that hits a Wait node over 65s, webhook, form, or HITL node	Rework the parent to cons output
Behavior	Start node removed	Any workflow still has a Start node	Manual Trigger, or Execute sub-workflows; delete if dis
Behavior	Publish/Unpublish replaces active toggle	Everyone	No action; learn the new U
Behavior	Spontit, crowd.dev, Kitemaker, Automizy removed	Any workflow uses one	Remove or replace the noc
Security	`N8N_BLOCK_ENV_ACCESS_IN_NODE=true`	Code node reads `process.env`	Move the value into a cred
Security	Settings file must be `0600`	Config file permissions are looser	`chmod 600`, or `N8N_ENFORCE_SETTINGS_F`
Security	Task runners on by default	Any Code node	Configure external runners is deprecated
Security	`$evaluateExpression` dead in Code node	A Code node calls it	Rewrite in JS, or pre-evalu
Security	Runner removed from `n8nio/n8n` image	You ran external runners from the main image	Use `n8nio/runners` at th
Security	Pyodide Python removed	Any Python Code node or tool	External runners plus `N8N_NATIVE_PYTHON_RUNN` code
Security	ExecuteCommand and LocalFileTrigger excluded	A workflow uses them	Adjust `NODES_EXCLUDE`, o
Security	`N8N_SKIP_AUTH_ON_OAUTH_CALLBACK=false`	You reconnect OAuth credentials	Test each OAuth credentia
Security	`N8N_RESTRICT_FILE_ACCESS_TO=~/.n8n-files`	File nodes touch other paths	Move the files, or widen th
Security	`N8N_GIT_NODE_DISABLE_BARE_REPOS=true`	Git node uses a bare repo	Set to `false` only if you n
Data	MySQL/MariaDB dropped	`DB_TYPE=mysqldb` or `mariadb`	Move to Postgres **while st**
Data	SQLite legacy driver removed	`DB_TYPE=sqlite`	Pooled driver is now defau `DB_SQLITE_POOL_SIZE`
Data	In-memory binary mode removed	`N8N_DEFAULT_BINARY_DATA_MODE=default`	Pick `filesystem`, `databa` `N8N_AVAILABLE_BINARY_D`
Config	dotenv upgraded	`.env` uses backticks, `#`, or odd quoting	Re-quote values; verify pa
Config	`n8n --tunnel` removed	Dev setups only	Use ngrok, localtunnel, or
Config	`QUEUE_WORKER_MAX_STALLED_COUNT` removed	Queue mode	Delete the variable
Config	`N8N_CONFIG_FILES` removed	You load JSON config files	Move to env vars, `.env`, o
CLI	`update:workflow` replaced	Scripts or CI call it	`publish:workflow` / `unp`
Hooks	`workflow.activeChange` deprecated	Custom external hooks	Use `workflow.published`
Channels	`latest` / `next` → `stable` / `beta`	You track a floating tag	Pin an exact version instea

5. Behavior Changes

Sub-workflow return values when the child waits. On 1.x, if a parent called a sub-workflow that entered the waiting state — a Wait node with a timeout over 65 seconds, a webhook call, a form submission, or a human-in-the-loop node such as Slack approval — and the parent was set to wait for completion, the parent received the child's *input* echoed back as its output. That was wrong. On 2.0 the parent receives the child's real output.

How to tell: the Migration Report flags it. Manually, look for Execute Workflow nodes set to wait, and check whether the child contains a Wait, webhook, form, or HITL node.

What to do: rework the parent to consume the child's actual output. This is what unlocks the useful pattern — a sub-workflow that asks a human to approve or decline, and a parent that branches on the answer.

Start node removed. Replace it based on how the workflow runs: a **Manual Trigger** for manual executions; an **Execute Workflow Trigger** if another workflow calls this one as a sub-workflow (then publish it); if the Start node is disabled, delete it.

Publish/Unpublish replaces the active toggle. Activate/Deactivate is now Publish/Unpublish, backed by versioning, so you edit freely and publish deliberately. Nothing to fix — but see section 14 about workflows appearing unpublished after the upgrade.

Removed nodes. Spontit, crowd.dev, Kitemaker and Automizy are gone because the services behind them are gone. Any workflow containing one errors. Remove or replace.

---

6. Security

Every item here is a default that flipped. Each has an escape hatch, and each escape hatch gives up the protection — say what you are giving up before you use one.

`N8N_BLOCK_ENV_ACCESS_IN_NODE=true`. Code nodes can no longer read the container's environment variables. Setting it back to `false` means anyone who can edit a workflow can read every secret in your environment block, including `N8N_ENCRYPTION_KEY` and the database password. Move the values into credentials instead.

Settings file permissions. The config file in the n8n data directory must be `0600` — owner read/write only, the way SSH treats private keys — and n8n refuses to start otherwise. Rehearse on 1.x by setting `N8N_ENFORCE_SETTINGS_FILE_PERMISSIONS=true`. On a filesystem that cannot express Unix permissions (a Windows bind mount, some network volumes) set it to `false`.

Task runners on by default. Every Code node execution runs on a task runner. `N8N_RUNNERS_ENABLED` is deprecated in 2.0 — on 1.x it was how you turned runners on, and setting it to `true` there is how you rehearse. On 2.0 what matters is `N8N_RUNNERS_MODE`: `internal` spawns a child process and is not for production, `external` runs the sidecar. The reference stack uses `external`.

`$evaluateExpression` in Code nodes. Runners run in secure mode, which disables evaluating strings as code — the mechanism expressions rely on. `$evaluateExpression()` inside a Code node now returns `null` or errors. Expressions in ordinary node fields, such as Edit Fields (Set), are unaffected. Fix it in this order: write the logic in JavaScript directly; evaluate the expression in a Set node before the Code node and read the result from the incoming item; or, last resort, set `N8N_RUNNERS_INSECURE_MODE=true`, which turns the runner's security measures off instance-wide. Treat that as temporary — the method may be removed outright in a future version.

The runner moved out of the main image. `n8nio/n8n` no longer contains the task runner binary for external mode. Use the separate `n8nio/runners` image, pinned to the same version as n8n.

Pyodide Python removed. The browser-style Python runtime is replaced by native Python, which runs only on external task runners; set `N8N_NATIVE_PYTHON_RUNNER=true` on the n8n service. Native Python drops the Pyodide-era conveniences — no `_input` built-in, no dot-access notation — so existing Python Code nodes need rewriting, not just re-pointing. Native Python *tools* get `_query`, the input string the AI Agent passed in.

ExecuteCommand and LocalFileTrigger disabled. Both are excluded by default through `NODES_EXCLUDE`, because both hand workflow editors arbitrary command execution and filesystem access on the host. If you need one, remove just that entry. `NODES_EXCLUDE="[]"` re-enables everything and is the blunt option.

OAuth callbacks require authentication. `N8N_SKIP_AUTH_ON_OAUTH_CALLBACK` now defaults to `false`. Set it to `false` on 1.x and reconnect one OAuth credential, to confirm your proxy passes session cookies through the callback path.

`N8N_RESTRICT_FILE_ACCESS_TO` has a default. It is now `~/.n8n-files`, constraining the Read/Write File and Read Binary Files nodes. The reference stack already mounts a named volume there. Widen it only to paths you intend workflows to reach.

`N8N_GIT_NODE_DISABLE_BARE_REPOS=true`. The Git node blocks bare repositories. Set it to `false` only if a workflow depends on them.

---

7. Data

MySQL and MariaDB are dropped. Deprecated since 1.0, now removed. This is the one change you cannot fix afterwards: a 2.0 container pointed at MySQL will not start, and there is no in-place conversion. Migrate to PostgreSQL **while still on 1.x** using n8n's database migration tooling, verify the instance runs correctly on Postgres, take a fresh backup, and only then upgrade. The MySQL *node* is unaffected — this is only n8n's own storage backend.

SQLite legacy driver removed. The pooled driver is now the default and only SQLite driver. It uses WAL mode with one write connection and a pool of read connections, and n8n's benchmarks put it up to ten times faster. `DB_SQLITE_POOL_SIZE` defaults to `2`; enable pooling on 1.x by setting it above `0`. For anything beyond a laptop, use Postgres.

In-memory binary data mode removed. `N8N_DEFAULT_BINARY_DATA_MODE=default`, which held execution binary data in RAM, is gone. The options are `filesystem` (default in regular mode), `database` (default in queue mode) and `s3`. `N8N_AVAILABLE_BINARY_DATA_MODES` is removed entirely, so the mode is set solely by `N8N_DEFAULT_BINARY_DATA_MODE`. Delete the old variable and make sure the host has disk for data that used to live in memory.

---

8. Configuration and Environment

dotenv upgraded. n8n moved from dotenv 8.6.0 to a current release, and parsing changed. Three things to check in `.env`:

- Values containing backticks must be wrapped in single or double quotes.
- `#` starts a comment. A password containing `#` needs quoting, or everything after it disappears.
- Multiline values are now supported, so a stray unbalanced quote can swallow the following lines instead of erroring on its own.

Read `.env` line by line before you upgrade. A silently truncated password is a miserable thing to debug at 2am.

`n8n --tunnel` removed. Use ngrok, localtunnel, or Cloudflare Tunnel, and set `WEBHOOK_URL` and `N8N_EDITOR_BASE_URL` to the tunnel hostname.

`QUEUE_WORKER_MAX_STALLED_COUNT` removed. The variable and the Bull stalled-job retry behind it are gone; they were confusing and unreliable. Delete it. n8n no longer retries stalled jobs automatically. Use `N8N_GRACEFUL_SHUTDOWN_TIMEOUT` to give workers time to finish on shutdown.

`N8N_CONFIG_FILES` removed. JSON config files are no longer loaded. Move that configuration into environment variables, `.env`, or `_FILE`-suffixed variables pointing at a file holding a single secret.

---

9. CLI and Workflow

`update:workflow` is replaced by two commands with clearer intent:

```
# Publish one workflow, optionally a specific version
docker compose exec n8n n8n publish:workflow --id=<workflow-id>
docker compose exec n8n n8n publish:workflow --id=<workflow-id> --versionId=<version-id>

# Unpublish one workflow, or all of them
docker compose exec n8n n8n unpublish:workflow --id=<workflow-id>
docker compose exec n8n n8n unpublish:workflow --all
```

The asymmetry is deliberate: `publish:workflow` has no `--all`, so you cannot mass-publish production by accident, while `unpublish:workflow` keeps it, because turning everything off in a hurry is a legitimate thing to want. Grep your CI pipelines, deploy scripts and cron jobs for `update:workflow` before you upgrade.

---

10. External Hooks

If you run custom frontend external hooks, `workflow.activeChange` and `workflow.activeChangeCurrent` are deprecated, replaced by a single `workflow.published` hook that fires whenever any version of a workflow is published. Update your hook code to the new name. If you do not use external hooks, skip this.

---

11. Release Channels

The channels are renamed: `latest` becomes `stable`, `next` becomes `beta`, on both Docker Hub and npm. `stable` is the newest stable release; `beta` the newest experimental one. The old tags still resolve for now and will be removed in a future major version.

Renaming them does not make floating tags safe. This book pins an exact version everywhere, so `docker compose pull` never surprises you and a rollback has something concrete to return to. If you must track a channel on a non-production instance, use `stable`.

---

12. Performing the Upgrade

You have a clean Migration Report on the copy, a fresh backup, and a `.env` you have re-read. On the reference stack the upgrade is two tag changes in `compose.yml`:

```
  n8n:
    image: n8nio/n8n:2.38.1

  runners:
    image: n8nio/runners:2.38.1
```

Both must be the same version — a runners sidecar on a different version than n8n is unsupported and fails in confusing ways. If your 1.x stack has no `runners` service at all, add one now, because on 2.0 every Code node needs it.

Pull first, so the download does not happen inside the restart window, then bring the stack up and watch the logs. n8n runs its database migrations on first start of the new version, and on a large executions table this takes minutes:

```
docker compose pull
docker compose up -d
docker compose logs -f n8n
```

You want migration lines in sequence, then a normal startup banner. Do not interrupt this. A half-applied migration is a restore, not a retry.

Verify with something that exercises the new machinery, not just the editor loading:

1. Log in with your existing owner account.
2. Open a credential and confirm the fields are populated — that proves the encryption key survived.
3. Build a throwaway workflow: **Manual Trigger** → **Code node** returning `[{ json: { ok: true } }]`, and execute it. A Code node that runs proves the task runner is connected; one that hangs or errors means the sidecar is not talking to the broker.
4. Check `docker compose logs runners`.
5. Publish one low-risk real workflow, watch it run once, then publish the rest.

---

13. Rollback Plan

Be clear about what rollback means. **n8n's database migrations are not reversible.** Once 2.0 has migrated the schema, pointing a 1.x container at that database does not work, and there is no downgrade path in the software. A rollback is a restore.

That is why section 2 insisted on the pre-upgrade dump.

```
# 1. Stop everything
docker compose down

# 2. Put the old tags back in compose.yml, then start only Postgres
docker compose up -d postgres

# 3. Restore the pre-upgrade dump over the migrated database
docker compose exec -T postgres \
  pg_restore -U n8n -d n8n --clean --if-exists < n8n-pre-2.0-2026-09-05.dump

# 4. Restore the data volume if anything in it changed
docker run --rm \
  -v n8n_data:/data -v "$(pwd)":/backup \
  alpine sh -c "rm -rf /data/* && tar xzf /backup/n8n-data-pre-2.0-2026-09-05.tar.gz -C /data"

# 5. Restore .env, including the original N8N_ENCRYPTION_KEY
cp env-pre-2.0-2026-09-05.bak .env

# 6. Start the old version
docker compose up -d
```

Two consequences follow. Everything between the backup and the rollback is lost — executions, edits, new credentials — so take the dump immediately before the upgrade, not the night before. And decide your rollback deadline in advance: "we roll back if it is not healthy in 30 minutes" is a decision to make while calm, because after 30 minutes of production traffic the restore costs you 30 minutes of data.

---

14. Troubleshooting Common Issues

n8n refuses to start, complaining about config file permissions. 2.0 enforces `0600` on the settings file. Fix it inside the volume:

```
docker compose run --rm --user root --entrypoint sh n8n \
  -c "chmod 600 /home/node/.n8n/config && chown node:node /home/node/.n8n/config"
```

If the volume is a bind mount on a filesystem that cannot express Unix permissions, set `N8N_ENFORCE_SETTINGS_FILE_PERMISSIONS=false` and understand you are keeping the config file readable by anything on that host.

Code node errors after the upgrade. In order: an error mentioning `process.env`, or `undefined` where a secret should be, is `N8N_BLOCK_ENV_ACCESS_IN_NODE=true` — move the value into a credential. An error mentioning `$evaluateExpression` is section 6. If the node hangs, times out, or reports no runner available, the sidecar is the problem: confirm the `runners` service is running, that both images are the same version, that `N8N_RUNNERS_AUTH_TOKEN` is identical on both, and that `N8N_RUNNERS_TASK_BROKER_URI` points at the right service name on port 5679. In queue mode, every worker needs its own runners sidecar. If it is a Python Code node, it is the Pyodide removal — you need external runners, `N8N_NATIVE_PYTHON_RUNNER=true`, and rewritten code.

Workflows that were active now show as unpublished. Check before you panic: the toggle became Publish/Unpublish, so the label changed for everything, and a workflow still live now reads “Published”. If one really is unpublished, the usual cause is that it contained something 2.0 rejects — most often a Start node. Open it, replace the Start node with a Manual Trigger or an Execute Workflow Trigger, save, publish. The Migration Report stays available after the upgrade; check it on the new instance too.

Credentials are undecryptable after a restore. Credentials exist and open but show empty or garbled fields, and nodes fail to authenticate. The cause is almost always that `N8N_ENCRYPTION_KEY` is not the value it was when those credentials were saved — typically because the key was never set explicitly, so n8n generated one into the data volume, and the volume was replaced while the database was restored. Restore the original `.env` and the original data volume together, and confirm:

```
docker compose exec n8n printenv N8N_ENCRYPTION_KEY
```

If the key is genuinely lost, the credentials are not recoverable. Delete and re-enter them, then set the key explicitly in `.env` so this cannot happen twice.

Further reading

- https://docs.n8n.io/changelog/v20-breaking-changes
- https://docs.n8n.io/changelog/v20-migration-tool
- https://docs.n8n.io/changelog/v30-breaking-changes
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/set-up-task-runners
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/scaling/handle-binary-data
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration
- https://docs.n8n.io/build/understand-workflows/save-and-publish-workflows
- https://docs.n8n.io/integrations/builtin/core-nodes/n8n-nodes-base.code
- https://github.com/motdotla/dotenv/blob/master/CHANGELOG.md

---

Chapter 34: Preparing for n8n 3.0

Overview

n8n 3.0 is scheduled for October 2026. n8n's own breaking-changes notice describes it as a security- and cleanup-focused major release: tighter defaults, a Docker-only deployment model, and the removal of nodes and helpers that newer patterns replaced years ago. For most recently built workflows, none of this touches you. For anyone still running n8n with `npm` or `npx n8n`, or with workflows built on the old Function node, Item Lists node, or version 1 of the AI Agent node, it does.

This chapter tells you what n8n has confirmed will change, what to do about it now, and how to find out today whether your workflows are affected. It does not guess at anything n8n hasn't published. n8n's breaking-changes page says outright: "n8n will update this page with full details, migration guides, and links as n8n 3.0 approaches its release." The exact shape of the tighter security defaults, and the full list of "non-functional nodes" being retired, are still vague in the source document as of this writing. Where that's true, this chapter says so instead of inventing detail.

You will:

1. Understand the headline change: Docker-based self-hosting only.
2. Get a migration path off `npm`, `npx`, or PM2, if you're still on one of them.
3. See every removed node and helper mapped to its replacement.
4. Understand what changes for the Execute Workflow node and the AI Agent node.
5. Know what's tightening in security defaults, including the Compression node.
6. Learn what's being retired outright.
7. Run a one-line scan across your workflow exports to find out if this affects you today.
8. Get a 90-day timeline to work from before release.

Prerequisites

- An n8n instance on the 2.x line (the reference stack in this book runs `n8nio/n8n:2.38.1`).
- Shell access to your workflow exports, and `jq` installed for the scanning script in Section 9.
- If you're on `npm` or `npx n8n`: read n8n on Your Laptop with Docker Compose and Legacy Installs with Node.js, npm, and PM2 (until n8n 3.0) before starting the migration in Section 2.
- Fifteen minutes to run the scan in Section 9, even if you don't plan to act on anything else in this chapter yet.

1. The headline change: Docker-based self-hosting only

n8n 3.0 drops npm and npx installs; any OCI container runtime or orchestrator (Docker, Podman, Kubernetes, Nomad) that runs the official image is fine. This isn't a recommendation shift — it's the removal of an install method. An instance running `npm` or `npx n8n` will not be able to upgrade in place to 3.0.

n8n's guidance here is short: if you run n8n with `npm` or `npx n8n`, move to a Docker-based deployment before you upgrade, and for local installs Docker Compose is expected to be the easiest path. n8n has not yet published step-by-step migration tooling for this — the breaking-changes page notes that guidance is still coming. Section 2 gives you a path that works today, using commands already documented in the n8n CLI. This also covers anyone running n8n under PM2 on top of an `npm` install — the install method is what's removed, regardless of what supervises the process.

2. Migrating off npm, npx, and PM2

If your instance runs on `npm`, `npx n8n`, or `npm` under PM2, here's the migration path. It uses n8n's built-in export and import CLI commands, which already work in 2.x, so you can do this well before 3.0 ships and cut over on your own schedule.

Step 1: Export workflows and credentials from the existing instance.

```
# Run on the existing npm/npx/PM2 host
n8n export:workflow --all --output=export/workflows.json --pretty
n8n export:credentials --all --output=export/credentials.json --pretty
```

Credentials export in their encrypted form by default. You'll need the source instance's `N8N_ENCRYPTION_KEY` on the destination for the import to decrypt them — that's the whole reason the key has to match, not a separate step you can skip.

Step 2: Find the existing encryption key.

```
# npm install: check the config file or your process environment
cat ~/.n8n/config | grep -i encryptionKey
# or, if you set it as an environment variable:
echo "$N8N_ENCRYPTION_KEY"
```

Write this value down. Losing it between export and import makes the credentials export unreadable.

Step 3: Stand up the Docker stack.

Use the reference stack from Ubuntu VPS with Docker Compose (the Reference Stack). Before starting it, set `N8N_ENCRYPTION_KEY` in `.env` to the **exact same value** you found in Step 2:

```
N8N_ENCRYPTION_KEY=the-same-key-from-the-npm-install
```

Bring the stack up, but don't point any webhooks or DNS at it yet:

```
docker compose up -d
```

Step 4: Import workflows and credentials into the new container.

```
docker compose cp export/workflows.json n8n:/tmp/workflows.json
docker compose cp export/credentials.json n8n:/tmp/credentials.json
docker compose exec n8n n8n import:workflow --input=/tmp/workflows.json
docker compose exec n8n n8n import:credentials --input=/tmp/credentials.json
```

Step 5: Verify, then cut over.

Open the new instance's editor, confirm your workflows and credentials look right, and run each active workflow manually once if you can. Then update `WEBHOOK_URL`, `N8N_HOST`, and your DNS or reverse proxy to point at the Docker instance, and decommission the `npm`/PM2 host. Keep the old host stopped, not deleted, for a week.

3. Removed nodes and their replacements

n8n 3.0 removes three legacy nodes outright. Current replacements have existed for a long time; this just removes the fallback.

Removed	Replace with
Function node	Code node, **Run Once for All Items** mode
Function Item node	Code node, **Run Once for Each Item** mode
Item Lists node	Split Out, Aggregate, Sort, Limit, Remove Duplicates, or Summarize — pick the node matching the operation you were using

The Item Lists node bundled several operations into one node, so check which operation each instance performs before choosing its successor.

4. Execute Workflow node: old behavior removed

n8n 3.0 removes the Execute Workflow node's older behavior. n8n's breaking-changes page states this as a fact without describing the mechanics of what the old behavior was or what replaces it — that detail hasn't been published yet. If you have Execute Workflow nodes that predate n8n 2.0 and haven't been touched since, flag them for a manual re-check once n8n publishes the migration guidance this page promises, rather than assuming they'll behave identically after the upgrade.

5. AI Agent node: version 1 and its modes removed

Version 1 of the AI Agent node supported several agent type modes: **SQL Agent**, **Conversational Agent**, **OpenAI Functions Agent**, **Plan and Execute Agent**, and **ReAct Agent**. n8n 3.0 removes version 1 of the node along with all five modes.

If your workflows already use the current AI Agent node in **Tools Agent** mode, nothing changes — n8n's guidance is explicit that Tools Agent workflows continue to behave the same after the upgrade. The work is only for workflows still pinned to version 1.

For SQL Agent specifically, there's a direct replacement: pair a Postgres or MySQL tool sub-node with a current AI Agent node instead. For the other four modes, n8n's guidance is simply to update the workflow to the current AI Agent node version — it doesn't map each old mode to a new configuration, so budget time to rebuild and test the agent's prompt and tool set rather than expecting a drop-in swap.

6. `$getPairedItem` removed

n8n 3.0 removes the deprecated `$getPairedItem` expression helper. Use n8n's standard item-linking mechanism instead: the `pairedItem` property, or `$("<node-name>").item` to reach back into a specific node's output. If you have expressions calling `$getPairedItem(...)`, they'll need to be rewritten before the upgrade — the helper won't fail gracefully, it will simply not exist.

7. Tighter security defaults

n8n 3.0 tightens several security defaults. n8n's own summary is short: tighter handling of risky resource names, more secure credential behavior, and key rotation enabled by default. None of the three has published mechanics yet — there's no specific list of which resource names become "risky" or what exactly changes in credential handling. Treat this as a pending hardening move, not a feature change, and re-test credential-heavy workflows after you upgrade rather than before.

The one item in this section with concrete numbers is the Compression node. Its decompression limits drop from the current defaults:

- `N8N_COMPRESSION_NODE_MAX_DECOMPRESSED_SIZE_BYTES` drops from 2 GiB to 256 MiB.
- `N8N_COMPRESSION_NODE_MAX_ZIP_ENTRIES` drops from 5,000 to 1,000.

If any workflow decompresses archives larger than 256 MiB or with more than 1,000 entries, set both variables explicitly to their previous values before upgrading:

```
N8N_COMPRESSION_NODE_MAX_DECOMPRESSED_SIZE_BYTES=2147483648
N8N_COMPRESSION_NODE_MAX_ZIP_ENTRIES=5000
```

Setting these preserves current behavior; leaving them unset means large or entry-heavy archives will start failing where they didn't before.

8. Retired capabilities

Three things go away outright in 3.0:

- **Chat Hub** is retired.
- **Workflow import from URL** in the editor is removed. Other import paths keep working: copy-paste, **Import from File** in the editor's UI menu, the CLI (`n8n import:workflow`), and the n8n API. Switch any automation that pushes a URL into the editor's import field to the API or CLI instead.
- **Non-functional nodes** are removed. n8n hasn't published which nodes this covers — the phrase implies nodes that no longer work against their target service or API, not a named list yet.

9. Finding affected workflows today

Before you write any shell script, check **Settings** → **Migration Report** (the same admin tool described in “Upgrading to n8n 2.0”). It already scans the instance for Workflow and Instance Issues at Critical, Medium, and Low severity, with a Refresh button to re-scan, and it needs no shell access — it’s the fastest first pass. It predates 3.0, though, so treat the `jq` scan below as the complement: it targets the specific 3.0-era node types (the removed nodes, AI Agent v1) that the report may not flag yet.

You don’t need to wait for 3.0 to find out if you’re affected. Export your workflows and scan the JSON for the node types this chapter covers: `n8n-nodes-base.function`, `n8n-nodes-base.functionItem`, `n8n-nodes-base.itemLists`, and version 1 of the AI Agent node (`n8n-nodes-langchain.agent` with `typeVersion` 1).

```
n8n export:workflow --all --separate --output=export/
```

Then run this over the export directory:

```
jq -r '.nodes[] | select(.type=="n8n-nodes-base.function" or .type=="n8n-nodes-base.functionItem" or
        .type=="n8n-nodes-base.itemLists" or (.type=="n8n-nodes-langchain.agent" and .typeVersion==1))
        | .type' export/*.json | sort | uniq -c
```

The output is a count of each affected node type across every exported workflow. Zero output means none of your workflows use these nodes; anything else is your inventory.

10. A 90-day timeline

Work backward from October 2026 using n8n's confirmed changes, not the parts still pending.

- **Now: inventory.** Run the scan in Section 9 against every environment you run — production, staging, anything with active credentials. Note which workflows use the Function, Function Item, or Item Lists nodes, and which use AI Agent version 1 in a non-Tools-Agent mode.
- **30 days: move installs to Docker.** If any instance still runs on `npm`, `npx n8n`, or PM2, follow Section 2 and cut it over. Do this well before the node-level work in the next step — you want to be testing node replacements on the deployment model you'll actually run in production.
- **60 days: replace nodes.** Work through the table in Section 3 and the AI Agent guidance in Section 5, workflow by workflow. Rewrite any `$getPairedItem` expressions found during the scan. Set the Compression node environment variables from Section 7 if you rely on the current limits.
- **Before release: pin your version and wait.** Pin production to your current 2.x tag explicitly (this book's reference stack pins `n8nio/n8n:2.38.1`) rather than tracking `stable`. When 3.0.0 ships, let it sit for one or two patch releases before touching production — a first major release is exactly when this chapter's vaguer sections (Section 4, most of Section 7, and the non-functional-nodes list in Section 8) will surface as real bug reports from other operators. Read those patch changelogs before you upgrade.

11. Troubleshooting Common Issues

Credential import fails or credentials show as unreadable. The destination `N8N_ENCRYPTION_KEY` doesn't match the source instance's key. Re-check the value from Section 2, Step 2, and re-run `n8n import:credentials` after correcting `.env`. There is no recovery path other than re-entering credentials by hand.

`docker compose exec n8n n8n import:workflow` returns a permissions or path error. The file didn't make it into the container, or landed somewhere the `node` user can't read. Confirm the `docker compose cp` step completed and re-check the path passed to `--input`.

A workflow that used the Function node behaves differently after switching to the Code node. The Function node ran once for all items by default; if you pick **Run Once for Each Item** without adjusting the script, logic that assumed access to the full input array will break. Match the mode to what the original node actually did, not just its name.

An AI Agent workflow stops producing tool calls after upgrading from version 1. This usually means the workflow used **OpenAI Functions Agent**, **ReAct Agent**, **Plan and Execute Agent**, or **Conversational Agent** mode, none of which carry forward. Rebuild it on the current AI Agent node in **Tools Agent** mode, re-attach the same tool sub-nodes, and re-test the prompt — treat it as a rebuild, not an upgrade.

The scan in Section 9 finds nothing, but you know a workflow uses an old node. Check that the export actually captured it — `n8n export:workflow --all` should include inactive and archived workflows, but a filtered export won't. Re-run without ID filters and confirm the file count matches your workflow count in the editor.

Compression node starts failing on archives that worked before. You upgraded without setting `N8N_COMPRESSION_NODE_MAX_DECOMPRESSED_SIZE_BYTES` and `N8N_COMPRESSION_NODE_MAX_ZIP_ENTRIES`. Set both to their prior defaults (`2147483648` and `5000`) and restart the container.

Further reading

- https://docs.n8n.io/changelog/v30-breaking-changes.md
- https://docs.n8n.io/deploy/host-n8n/install-options/install-with-npm.md
- https://docs.n8n.io/deploy/host-n8n/install-options/install-using-docker-compose.md
- https://docs.n8n.io/build/manage-workflows/export-and-import.md
- https://docs.n8n.io/changelog/v20-migration-tool.md

Appendix A: Legacy Installs with Node.js, npm, and PM2 (until n8n 3.0)

> **Deprecated path.** Installing n8n with `npm install -g n8n` and running it directly on a host's Node.js runtime works on n8n 1.x and 2.x, but it is already deprecated. n8n 3.0, scheduled for **October 2026**, supports **Docker-based self-hosting only** — the npm package stops being a supported way to run n8n. If you are setting up a new instance, skip this appendix and use Docker Compose instead: see "n8n on Your Laptop with Docker Compose" for a laptop, or "Ubuntu VPS with Docker Compose (the Reference Stack)" for a server. This appendix exists for people who already have an npm install running and need to maintain it, or migrate it, before 3.0 ships.

Who this is for

You have an existing n8n instance installed via npm, possibly kept alive with PM2 or a plain systemd unit, and you need one of three things: to keep it patched and running a little longer, to understand what actually matters in its configuration, or to move it onto the Docker stack the rest of this book uses. This appendix covers all three and drops everything else the original chapters covered — installer GUIs, Yarn/pnpm variants, offline installers, and corporate-proxy edge cases are not maintenance concerns and are not repeated here.

Node.js version and nvm

n8n's npm install requires **Node.js 20.19 through 24.x**. Anything outside that range will fail to start or throw obscure errors during `npm install`. Use nvm to pin a working version without touching your system Node.js:

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.3/install.sh | bash
source ~/.nvm/nvm.sh
nvm install 22
nvm use 22
nvm alias default 22
node -v
```

Node.js 22 sits comfortably inside the supported range and is the safest default. If PM2 or systemd runs n8n as a different user (a dedicated service account, for example), that user needs its own nvm install and `default` alias — nvm is per-user, not system-wide.

Installing and pinning n8n

Always install a specific version, never a bare `npm install -g n8n`, which pulls whatever is newest at that moment:

```
npm install -g n8n@2.38.1
n8n --version
```

Pinning matters more here than with the Docker image tags used elsewhere in this book: npm has no equivalent of `docker compose down && up -d` to roll back cleanly, and a `n8n@next` or unpinned install can jump you into a version with schema or breaking changes you haven't read about yet. Before bumping the pinned version, check the release notes for the target version and run `n8n db:revert` is available as an escape hatch for a bad database migration, but treat it as a last resort, not a plan.

Environment variables that matter

Most of what the original npm chapters covered — `N8N_PORT`, proxy flags, CLI switches — is cosmetic. Three things are not:

- `N8N_ENCRYPTION_KEY` — encrypts every stored credential. Generate it once and never regenerate it on an existing instance:

  ```
  openssl rand -hex 32
  ```

  Losing this key makes every saved credential permanently unreadable. Back it up with the same care as the database.

- `N8N_HOST` / `WEBHOOK_URL` — if anything besides `localhost` reaches this instance (a reverse proxy, a tunnel, a real domain), set both, plus `N8N_PROTOCOL=https` and `N8N_EDITOR_BASE_URL`. Webhooks silently register the wrong URL if you skip this.

- **Database** — `DB_TYPE=sqlite` is the default and fine for a single low-traffic instance; for anything you care about, point at PostgreSQL with `DB_TYPE=postgresdb`, `DB_POSTGRESDB_HOST`, `DB_POSTGRESDB_PORT`, `DB_POSTGRESDB_DATABASE`, `DB_POSTGRESDB_USER`, `DB_POSTGRESDB_PASSWORD`.

Set these in a `.env` file next to wherever you launch n8n from, loaded with `dotenv`:

```
N8N_ENCRYPTION_KEY=replace-with-64-hex-characters
N8N_HOST=n8n.example.com
N8N_PROTOCOL=https
WEBHOOK_URL=https://n8n.example.com/
N8N_EDITOR_BASE_URL=https://n8n.example.com/
DB_TYPE=postgresdb
DB_POSTGRESDB_HOST=localhost
DB_POSTGRESDB_PORT=5432
DB_POSTGRESDB_DATABASE=n8n
DB_POSTGRESDB_USER=n8n
DB_POSTGRESDB_PASSWORD=replace-me
```

```
npx dotenv n8n
```

or export the same variables from a shell profile / systemd `EnvironmentFile` if you're not using PM2's own env handling (below).

A note on basic auth: older tutorials and some search results still reference `N8N_BASIC_AUTH_USER` and `N8N_BASIC_AUTH_PASSWORD`. These variables do nothing — they were removed in n8n 1.0. On a fresh instance, the first visit to the editor creates the owner account; there is no separate basic-auth layer to configure.

Running as a service with PM2

PM2 restarts n8n on crash and on boot. An ecosystem file is easier to audit and version-control than inline environment variables on the command line:

```
// ecosystem.config.js
module.exports = {
  apps: [{
    name: 'n8n',
    script: 'n8n',
    args: 'start',
    env: {
      NODE_ENV: 'production',
      N8N_ENCRYPTION_KEY: process.env.N8N_ENCRYPTION_KEY,
      N8N_HOST: process.env.N8N_HOST,
      N8N_PROTOCOL: 'https',
      WEBHOOK_URL: process.env.WEBHOOK_URL
    },
    autorestart: true,
    max_restarts: 5,
    restart_delay: 5000,
    out_file: '/var/log/n8n/out.log',
    error_file: '/var/log/n8n/error.log',
    log_date_format: 'YYYY-MM-DD HH:mm Z'
```

```
  }]
};
```

Referencing `process.env.*` instead of hardcoding secrets in the file means the values still have to come from somewhere — source a `.env` file (`set -a; source .env; set +a`) before running `pm2 start`, or use `pm2 start ecosystem.config.js --env-from-file .env` if your PM2 version supports it.

```
npm install -g pm2
pm2 start ecosystem.config.js
pm2 list
pm2 logs n8n
```

To survive a reboot:

```
pm2 startup systemd -u $USER --hp $HOME
# run the sudo command PM2 prints, then:
pm2 save
```

`pm2 save` snapshots the current process list; without it, a reboot brings up nothing. If the service user relies on nvm, confirm the generated systemd unit's `PATH` includes the nvm-selected Node.js binary — PM2's `startup` command sometimes captures the invoking shell's PATH at the moment you ran it, not a login shell's.

The leaner alternative: a plain systemd unit

PM2 is a dependency you have to install, update, and reason about on its own. If you don't need PM2's cluster mode or dashboard, a systemd unit does the same job — restart on failure, start on boot — with one fewer moving part:

```
# /etc/systemd/system/n8n.service
[Unit]
Description=n8n
After=network.target

[Service]
Type=simple
User=n8n
EnvironmentFile=/etc/n8n/n8n.env
ExecStart=/home/n8n/.nvm/versions/node/v22.14.0/bin/n8n start
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
```

```
sudo systemctl daemon-reload
sudo systemctl enable --now n8n
systemctl status n8n
```

`ExecStart` needs an absolute path to the `n8n` binary that nvm installed for the service user — systemd doesn't source shell profiles, so a bare `n8n` on `$PATH` won't resolve. Find it with `nvm which 22` (or the version you pinned) once, and hardcode the result.

Logs

- **PM2**: `pm2 logs n8n` for a live tail, or read the `out_file` / `error_file` paths set in the ecosystem file directly. Install `pm2 install pm2-logrotate` so those files don't grow unbounded.
- **systemd**: `journalctl -u n8n -f` for a live tail, `journalctl -u n8n --since "1 hour ago"` for a window.

Updating

```
npm view n8n versions --json   # see what's available
npm install -g n8n@2.38.1      # install a specific, tested version
```

```
pm2 restart n8n               # or: sudo systemctl restart n8n
```

Read the release notes for every version you cross, not just the one you land on — n8n ships breaking changes between minor versions, and skipping several at once means finding out about all of them at 2 a.m. Restart the process manager, not just the shell, after every update: PM2 and systemd both cache the resolved binary path at start.

Moving to Docker

This is the maintenance path with an end date. Do it before October 2026:

1. **Stop the service** but don't delete anything yet: `pm2 stop n8n` or `sudo systemctl stop n8n`.
2. **Keep the `~/.n8n` folder.** It holds the SQLite database (if you're using SQLite), the encryption-protected credential store, and binary data. Back it up regardless of database choice.
3. **Point the Docker stack at the same data.** If you're on PostgreSQL, run the `compose.yml` from this book's reference stack with `DB_POSTGRESDB_HOST` pointing at the same Postgres instance (or a restored copy of it) — no data migration needed, n8n's schema doesn't change based on how it's hosted. If you're on SQLite, copy the database file into the named volume the container mounts:

   ```
   docker volume create n8n_data
   docker run --rm -v n8n_data:/data -v ~/.n8n:/source alpine \
     cp /source/database.sqlite /data/database.sqlite
   ```

4. **Set the same `N8N_ENCRYPTION_KEY`** in the container's environment. This is the step people skip and then panic over — a different key means every credential decrypts to garbage.
5. **Start the container and verify credentials decrypt.** Log in, open a workflow that uses a saved credential, and run it (or at least open the credential and confirm it loads without an error). Don't move on until this works.
6. **Remove the npm install** — `npm uninstall -g n8n`, disable and remove the PM2 process (`pm2 delete n8n`, `pm2 save`) or the systemd unit (`systemctl disable --now n8n`) — so nothing tries to bind the same port or touch the same files as the container.

10. Troubleshooting Common Issues

Problem	Cause	Fix
`n8n: command not found`	Global npm bin isn't on `PATH`, or wrong nvm version active	`npm root -g` to find the bin dir and add it to `PATH`; confirm `nvm current` matches what you installed n8n under
`npm install -g n8n` fails to build	Node.js version outside 20.19–24, or missing build tools	`node -v`; install `build-essential` (Debian/Ubuntu) or `gcc-c++ make` (RHEL)
PM2 process vanishes after reboot	`pm2 save` was never run, or `pm2 startup` PATH doesn't include nvm's Node.js	Rerun `pm2 startup`, confirm the printed sudo command, then `pm2 save`
systemd unit fails immediately	`ExecStart` points at a Node.js binary that doesn't exist for that user	Run `sudo -u n8n nvm which 22` (or your version) and fix the path in the unit file
Credentials fail to decrypt after a restore	`N8N_ENCRYPTION_KEY` differs from the one the data was encrypted with	Recover the original key from backup; there is no recovery without it
Port already in use	Another process, or the old npm install still running alongside the new Docker container	`ss -tulpn | grep 5678`; stop and remove the npm-based service fully before starting the container
Environment variables silently ignored	Set in a shell profile that non-interactive services don't source	Move them into the PM2 ecosystem file's `env` block or a systemd `EnvironmentFile`

Windows and macOS notes

Both platforms follow the same shape — nvm (or Volta), a pinned `npm install -g n8n@2.38.1`, and a process manager — with different syntax. On Windows, set variables in PowerShell with `$env:N8N_ENCRYPTION_KEY = "..."` and use `pm2 startup windows` (or a tool like NSSM) instead of systemd; there is no native systemd equivalent. On macOS, `pm2 startup launchd` registers a launch agent instead of a systemd unit. Neither platform is a sensible target for a production n8n instance going forward — treat Windows and macOS npm installs as development-only, and move production workloads to Docker on Linux regardless of what you're maintaining today.

Further reading

- https://docs.n8n.io/deploy/host-n8n/install-options/install-with-npm
- https://docs.n8n.io/deploy/host-n8n/install-options/install-using-docker-compose
- https://docs.n8n.io/changelog/v30-breaking-changes
- https://docs.n8n.io/changelog/v20-breaking-changes
- https://docs.n8n.io/deploy/host-n8n/configure-n8n/basic-configuration/use-environment-variables/deployment

---

Appendix B: Environment Variable Reference for n8n 2.x

Environment variables the book's chapters rely on, grouped by function, with type, 2.x default, meaning, and what changed in 2.0. Defaults reflect n8n 2.38.1. Most variables accept a `_FILE` suffix (e.g. `N8N_ENCRYPTION_KEY_FILE`) to read the value from a file — see Encryption and secrets. Task runner container variables don't support `_FILE`.

Core and URLs

Variable	Type	2.x default	Meaning	Changed in 2.0
`N8N_HOST`	String	`localhost`	Hostname n8n runs on and advertises.	
`N8N_PORT`	Number	`5678`	HTTP port n8n listens on.	
`N8N_PROTOCOL`	Enum: `http`, `https`	`http`	Protocol used to reach n8n; use `https` behind TLS.	
`WEBHOOK_URL`	String	-	Base URL for webhooks behind a reverse proxy. Deprecated from 2.35.0, alias of `N8N_WEBHOOK_URL` — still works but logs a warning; prefer `N8N_WEBHOOK_URL` in new deployments.	
`N8N_EDITOR_BASE_URL`	String	-	Public editor URL. Also used in n8n's emails and the SAML redirect URL.	
`N8N_PROXY_HOPS`	Number	`0`	Number of reverse proxies in front of n8n. Set `1` behind a single proxy so n8n trusts the forwarded IP/protocol.	
`N8N_SECURE_COOKIE`	Boolean	`true`	Restricts cookies to HTTPS. Set `false` only for plain HTTP on a non-localhost address — this exposes the session cookie to interception.	
`GENERIC_TIMEZONE`	String	`America/New_York`	Default timezone for the Schedule Trigger and other date logic.	

Variable	Type	2.x default	Meaning	Changed in 2.0
`TZ`	String	see docs	System timezone for the Node.js process. Match it to `GENERIC_TIMEZONE` so logs and scheduling agree.	
`N8N_DIAGNOSTICS_ENABLED`	Boolean	`true`	Whether n8n sends anonymous telemetry. `false` also disables Ask AI in the Code node.	

Encryption and secrets

Variable	Type	2.x default	Meaning	Changed in 2.0
`N8N_ENCRYPTION_KEY`	String	Random key generated by n8n	Encrypts credentials in the database. Set explicitly and back it up; without it, a restored database yields unreadable credentials. Must match on every container touching the database.	
`<VAR>_FILE`	String (path)	-	Suffix on most variables: `MY_VAR_FILE=/path/to/secret` loads the value from a file, e.g. `N8N_ENCRYPTION_KEY_FILE`, `DB_POSTGRESDB_PASSWORD_FILE`. Useful with Docker/Kubernetes secrets.	
`N8N_ENFORCE_SETTINGS_FILE_PERMISSIONS`	Boolean	`true`	Requires the settings file to have `0600` permissions (owner-only), like SSH private keys. Set `false` on filesystems without Unix permissions (Windows).	`false` → `true`.

Database

Variable	Type	2.x default	Meaning	Changed in 2.0
`DB_TYPE`	Enum: `sqlite`, `postgresdb`	`sqlite`	Database backend. MySQL/MariaDB dropped in 2.0; migrate to PostgreSQL first with the built-in tool.	MySQL/MariaDB removed.
`DB_POSTGRESDB_DATABASE`	String	`n8n`	PostgreSQL database name.	
`DB_POSTGRESDB_HOST`	String	`localhost`	PostgreSQL host.	
`DB_POSTGRESDB_PORT`	Number	`5432`	PostgreSQL port.	
`DB_POSTGRESDB_USER`	String	`postgres`	PostgreSQL user.	
`DB_POSTGRESDB_PASSWORD`	String	-	PostgreSQL password.	
`DB_POSTGRESDB_POOL_SIZE`	Number	`2`	Number of parallel open PostgreSQL connections.	
`DB_POSTGRESDB_SCHEMA`	String	`public`	PostgreSQL schema.	

Variable	Type	2.x default	Meaning	Changed in 2.0
`DB_POSTGRESDB_SSL_ENABLED`	Boolean	`false`	Enables SSL to PostgreSQL. Auto-enabled if `_SSL_CA`, `_SSL_CERT`, or `_SSL_KEY` is set, or `_SSL_REJECT_UNAUTHORIZED` is `false`.	
`DB_POSTGRESDB_SSL_CA` / `_SSL_CERT` / `_SSL_KEY`	String	-	PostgreSQL SSL certificate authority, client certificate, and client key, respectively.	
`DB_POSTGRESDB_SSL_REJECT_UNAUTHORIZED`	Boolean	`true`	Whether to reject unauthorized (self-signed or mismatched) certificates.	
`DB_SQLITE_POOL_SIZE`	Number	`0`	`0` uses SQLite's rollback journal mode; above `0` switches to WAL mode with that many parallel read connections. WAL is faster and more reliable — use it beyond a laptop test. This book sets `DB_SQLITE_POOL_SIZE=2` explicitly to get the pooled WAL driver.	
`DB_SQLITE_VACUUM_ON_STARTUP`	Boolean	`false`	Runs `VACUUM` on startup to rebuild and shrink the file. Long-running and blocks startup; leave off unless reclaiming space.	

Executions

Variable	Type	2.x default	Meaning	Changed in 2.0
`EXECUTIONS_MODE`	Enum: `regular`, `queue`	`regular`	Whether executions run in-process (`regular`) or dispatch to workers via a queue.	
`EXECUTIONS_DATA_PRUNE`	Boolean	`true`	Whether to delete old execution data on a rolling basis.	
`EXECUTIONS_DATA_MAX_AGE`	Number	`336`	Execution age in hours before pruning (14 days).	
`EXECUTIONS_DATA_PRUNE_MAX_COUNT`	Number	`10000`	Maximum executions kept in the database; `0` disables the limit.	
`EXECUTIONS_DATA_SAVE_ON_ERROR` / `_ON_SUCCESS`	Enum: `all`, `none`	`all`	Whether to save execution data on error / on success.	
`EXECUTIONS_DATA_SAVE_ON_PROGRESS`	Boolean	`false`	Whether to save data after each node executes, not just at the end.	

Variable	Type	2.x default	Meaning	Changed in 2.0
`EXECUTIONS_DATA_SAVE_MANUAL_EXECUTIONS`	Boolean	`true`	Whether to save data for manually triggered executions.	
`EXECUTIONS_TIMEOUT`	Number	`-1`	Default per-workflow timeout in seconds; `-1` disables it. Workflows can raise this up to `EXECUTIONS_TIMEOUT_MAX`.	
`N8N_GRACEFUL_SHUTDOWN_TIMEOUT`	Number	`30`	Seconds n8n waits for components (including in-flight worker executions) to finish before exit.	Replaces `QUEUE_WORKER_TIMEOUT`.

Queue mode

Variable	Type	2.x default	Meaning	Changed in 2.0
`QUEUE_BULL_REDIS_HOST` / `_PORT`	String / Number	`localhost` / `6379`	Redis host and port for the job queue.	
`QUEUE_BULL_REDIS_PASSWORD`	String	-	Redis password, if required.	
`QUEUE_BULL_REDIS_TLS`	Boolean	`false`	Enables TLS on the Redis connection.	
`QUEUE_BULL_REDIS_CLUSTER_NODES`	String	-	Comma-separated `host:port` list for a Redis Cluster. When set, n8n uses a cluster client and ignores `_HOST` / `_PORT`.	
`OFFLOAD_MANUAL_EXECUTIONS_TO_WORKERS`	Boolean	`false`	Runs manual ("Test workflow") executions on a worker instead of main. Recommended `true` in production.	
`N8N_WEBHOOK_RESPONSE_RELAY_SIZE_MAX`	Number	`64`	Max size in MiB of a webhook response a worker relays to main in a queue message. Redis holds ~1.5x this per	

Variable	Type	2.x default	Meaning	Changed in 2.0
			response in flight.	
`N8N_WEBHOOK_RESPONSE_RELAY_OFFLOAD_ENABLED`	Boolean	`false`	Above the size max, stores the response in binary storage instead of failing the node. Needs a storing binary mode reachable by every instance. From 2.34.0.	

Binary data

Variable	Type	2.x default	Meaning	Changed in 2.0
`N8N_DEFAULT_BINARY_DATA_MODE`	Enum: `filesystem`, `database`, `s3`, `azure`	`filesystem` in regular mode, `database` in queue mode	Where n8n stores binary data. The in-memory `default` mode was removed in 2.0; filesystem mode isn't supported in queue mode (use `database` or `s3`).	`default` (in-memory) mode removed.
`N8N_BINARY_DATA_STORAGE_PATH`	String	`N8N_USER_FOLDER/binaryData`	Filesystem path for binary data when the mode is `filesystem`.	
`N8N_EXTERNAL_STORAGE_S3_HOST`	String	-	S3-compatible host, e.g. `s3.us-east-1.amazonaws.com`.	
`N8N_EXTERNAL_STORAGE_S3_BUCKET_NAME` / `_BUCKET_REGION`	String	-	Bucket name and region (e.g. `us-east-1`) for S3-compatible storage.	
`N8N_EXTERNAL_STORAGE_S3_ACCESS_KEY` / `_ACCESS_SECRET`	String	-	Access key and secret for S3-compatible storage.	

Task runners

Variable	Type	2.x default	Meaning	Changed in 2.0
`N8N_RUNNERS_MODE`	Enum: `internal`, `external`	`internal`	`internal` launches a runner as a child process (not for production);	

Variable	Type	2.x default	Meaning	Changed in 2.0
			`external` expects an orchestrator sidecar (`n8nio/runners`) to launch it.	
`N8N_RUNNERS_AUTH_TOKEN`	String	Random string	Shared secret the runner uses to authenticate to n8n. Required in `external` mode; must match both sides.	
`N8N_RUNNERS_BROKER_LISTEN_ADDRESS`	String	`127.0.0.1`	Address the task broker listens on. Set `0.0.0.0` so a sidecar container can reach it.	
`N8N_RUNNERS_TASK_BROKER_URI`	String	`http://127.0.0.1:5679`	URI the runner uses to reach the task broker, e.g. `http://n8n:5679`.	
`N8N_RUNNERS_AUTO_SHUTDOWN_TIMEOUT`	Number	`15`	Seconds an idle runner waits before shutting down.	
`N8N_RUNNERS_INSECURE_MODE`	Boolean	`false`	Disables runner security measures for modules that need them. Discouraged for production; the escape hatch after `$evaluateExpression()` stopped working in the Code node.	
`N8N_NATIVE_PYTHON_RUNNER`	Boolean	see docs	Enables the native Python runner (replacing removed Pyodide Python). No default is published — treat as opt-in.	Pyodide Python removed.
`N8N_RUNNERS_ENABLED`	Boolean	`false` (deprecated)	Legacy 1.x switch for task runners. **Deprecated from 2.0** — runners handle every Code node by default, so this has no effect.	Deprecated; runners always on.

Security defaults changed in 2.0

Variable	Type	2.x default	Meaning	Change
`N8N_BLOCK_ENV_ACCESS_IN_NODE`	Boolean	`true`	Blocks expressions and the Code node from reading process environment variables. Set `false` only if workflows genuinely need it; prefer credentials.	`false` -
`NODES_EXCLUDE`	Array of strings (JSON)	`["n8n-nodes-base.executeCommand",`	Nodes n8n refuses to load. Both default-excluded nodes allow arbitrary command execution or	`Execute` now exc

Variable	Type	2.x default	Meaning	Change
		`"n8n-nodes-base.localFileTrigger"]`	unrestricted file access. Set `"[]"` to re-enable all, or list only what you need.	
`N8N_RESTRICT_FILE_ACCESS_TO`	String (semicolon-separated paths)	`~/.n8n-files`	Directories the `ReadWriteFile` / `ReadBinaryFiles` nodes may access. Previously unrestricted.	unset →
`N8N_SKIP_AUTH_ON_OAUTH_CALLBACK`	Boolean	`false`	Whether OAuth callback endpoints skip authentication. 2.0 requires it by default.	`true` —
`N8N_GIT_NODE_DISABLE_BARE_REPOS`	Boolean	`true`	Blocks the Git node from working with bare repositories.	`false` -

Observability

Variable	Type	2.x default	Meaning	Changed in 2.0
`N8N_LOG_LEVEL`	Enum: `info`, `warn`, `error`, `debug`	`info`	Minimum log level n8n emits.	
`N8N_LOG_OUTPUT`	Enum: `console`, `file`	`console`	Where logs go; comma-separate both to log to console and file at once.	
`N8N_METRICS`	Boolean	`false`	Enables the `/metrics` endpoint for Prometheus. Main and worker instances can both expose it; keep it internal-only.	
`N8N_METRICS_PREFIX`	String	`n8n_`	Prefix applied to n8n-specific metric names.	
`N8N_OTEL_ENABLED`	Boolean	`false`	Enables OpenTelemetry tracing; `false` means n8n doesn't load the SDK at all. From 2.19.0.	
`N8N_OTEL_EXPORTER_OTLP_ENDPOINT` / `_TRACING_PATH`	String	`http://localhost:4318` / `/v1/traces`	Base URL of the OTLP HTTP collector and the path appended for trace export. From 2.19.0.	
`N8N_OTEL_EXPORTER_SERVICE_NAME`	String	`n8n`	`service.name` resource attribute on exported spans. From 2.19.0.	
`N8N_OTEL_TRACES_SAMPLE_RATE`	Number	`1.0`	Fraction of traces exported (`0`-`1`) via a trace-ID ratio sampler — a whole trace is kept or dropped together. From 2.19.0.	

Variable	Type	2.x default	Meaning	Changed in 2.0
`N8N_OTEL_TRACES_INCLUDE_NODE_SPANS`	Boolean	`true`	Emits a `node.execute` span per node alongside `workflow.execute` spans. From 2.19.0.	
`N8N_OTEL_TRACES_INJECT_OUTBOUND`	Boolean	`true`	Injects W3C `traceparent` / `tracestate` headers into outbound requests made via n8n's HTTP helpers. From 2.19.0.	

Removed variables

Delete any of these carried over from the first edition or older community guides — they no longer exist or no longer do anything in n8n 2.x.

- `N8N_BASIC_AUTH_USER`, `N8N_BASIC_AUTH_PASSWORD` — removed in n8n 1.0. There is no basic auth; the first editor visit creates the owner account, and n8n has its own user management, roles, and 2FA.
- `N8N_CONFIG_FILES` — removed in n8n 2.0. Use a single configuration file or environment variables directly.
- `QUEUE_WORKER_MAX_STALLED_COUNT` — removed in n8n 2.0; setting it has no effect.
- `QUEUE_WORKER_TIMEOUT` — deprecated from n8n 1.22.0; use `N8N_GRACEFUL_SHUTDOWN_TIMEOUT` instead.
- `N8N_AVAILABLE_BINARY_DATA_MODES` — removed in n8n 2.0 with the in-memory `default` mode. Mode is now determined solely by `N8N_DEFAULT_BINARY_DATA_MODE`.
- `QUEUE_MODE`, `REDIS_HOST`, `REDIS_PORT` — never existed in n8n. Delete them if copied from the first edition; queue mode is actually controlled by `EXECUTIONS_MODE=queue`, `QUEUE_BULL_REDIS_HOST`, and `QUEUE_BULL_REDIS_PORT`.

Conclusion: Your Journey with n8n

You started with one Compose file on a laptop. By the end of Part I that file had a task runner beside it and you knew why. In Part II the same file moved to a VPS, gained a reverse proxy and a public hostname, and then went to four clouds, a tunnel, and a platform-as-a-service without changing shape. Part III split it into main, workers, webhook processors, and runners, and then expressed that split in Swarm, Kubernetes, Helm, and the managed container services. Part IV was about keeping it alive: high availability, multi-region, the backup that includes the encryption key, pipelines that publish workflows instead of toggling them, observability, and the security defaults that n8n 2.0 turned on for you. Part V looked at the two version boundaries that matter this year.

Three habits carry all 34 chapters:

1. **Pin versions, and change two tags at a time.** The `n8nio/n8n` and `n8nio/runners` images move together. Read the release notes for migrations before you pull.
2. **Guard the encryption key.** It is the one thing a database backup cannot replace. Store it in a secrets manager, set it on every container that touches the database, and never rotate it outside n8n's documented procedure.
3. **Let n8n tell you its public URL.** Behind any proxy, tunnel, or load balancer, `N8N_HOST`, `N8N_PROTOCOL`, `WEBHOOK_URL`, `N8N_EDITOR_BASE_URL`, and `N8N_PROXY_HOPS` decide whether webhooks and OAuth callbacks work. Most "it works locally but not in production" reports are one of these five.

n8n 3.0 arrives in October 2026. Everything in this edition is built to survive it: Docker-based installs, current node types, the current AI Agent node, and configuration that exists in the 2.x documentation. When 3.0 ships, re-read the chapter on preparing for it, pin the first patch release rather than 3.0.0, and upgrade a copy first.

The workflows, compose files, and manifests in this book live and get updated in the House of Loops community on Skool, which is free. If a step did not work as written, or a platform changed under it, that is where to say so, and it is where the next revision starts.

Second edition, revised September 2026 for n8n 2.38 and the 3.0 transition. First edition April 2025.