Adoption & Rollout Guide

This guide is for teams adopting **MessageFoundry** — an open-source, Python healthcare integration engine — and taking it from a first install to full production use. It is an **orchestration** document: it ties the reference docs together and adds the install-to-production **rollout plan** that nothing else covers. Where a topic has a dedicated reference, this guide links to it rather than repeating it.

> **Read this section first.** MessageFoundry does a focused set of things well, and a staged rollout is the safest way to put any new integration engine into production. The point of the staged rollout in section 11 is to find issues where they are cheap (a lab, a shadow feed) instead of where they are expensive (a production cut-over). If you adopt MessageFoundry, adopt the rollout discipline with it.

Table of contents

1. What MessageFoundry is, and who should adopt it

MessageFoundry routes, transforms, and validates messages between **Connections** — HL7 v2.x by default, with payload-agnostic support for a wide range of other formats including X12 and FHIR — and its routing and handling are written in **Python**. The runtime model is a graph wired by name:

- **Connection** — an endpoint that receives (inbound) or sends (outbound) messages: MLLP, TCP, File, REST/SOAP, X12, FHIR, and Database destinations, plus a Database poll source. See CONNECTIONS.md.
- **Router** — a Python function bound to an inbound connection that decides which Handler(s) see each message.
- **Handler** — a Python function that filters, transforms a message, and emits `Send`s to outbound connections.

The engine is a **headless asyncio service** (FastAPI/uvicorn) that owns a durable message store and supervises one worker set per connection. A separate **desktop console** and a **VS Code extension** operate it over an HTTP API. See ARCHITECTURE.md for the full model.

Who should adopt it. Teams who want a Python-native, open-source healthcare integration engine and who prefer to validate any new tool against their own traffic before trusting it. A single engine node on a trusted network is the simplest deployment; **native TLS** (API + MLLP) and an opt-in **active-passive failover** cluster on a shared server database (PostgreSQL or SQL Server) are both available when you need them (see sections 2, 6, and 14). Horizontal *active-active* scale-out is not the model — active-passive HA is the supported HA pattern, and throughput scales intra-node (section 14).

---

2. Capabilities at a glance

MessageFoundry is built for single-node production and also supports **opt-in active-passive failover** (a leader/standby cluster on a shared server database — PostgreSQL or SQL Server; section 14). The authoritative reference is ARCHITECTURE.md; use the table below alongside it when planning.

Capability	Notes
Code-first Connection/Router/Handler graph	The core runtime model.
SQLite (WAL) store backend	Zero-config default; single-node and development.
PostgreSQL store backend	Full staged pipeline, at-rest encryption, retention; production-grade.
Microsoft SQL Server store backend	Full staged pipeline + response capture, at-rest encryption; needs the `sqlserver` extra + OS-level ODBC Driver 18. Retention is managed at the DBA tier (TDE + SQL Agent).
Transactional staged queue (ingress→routed→outbound), at-least-once, dead-letter, replay	See ADR 0001.
Auth + RBAC + hash-chained audit log	See SECURITY.md.
Multi-factor authentication	**Required** for local accounts; enterprise/AD users get MFA through their own identity provider via SSO federation.
At-rest body encryption (AES-256-GCM, opt-in) + key rotation	See PHI.md.
Connectors: MLLP / TCP / File; REST / SOAP; **X12**; **FHIR**; Database destinations; Database poll source	See CONNECTIONS.md.
Interface authentication: **mTLS**, **OAuth 2.0 client-credentials**, **SMART-on-FHIR Backend Services**	For securing inbound/outbound interfaces.
Native transport TLS (API + MLLP)	In-process API TLS (HTTPS/WSS) + per-connection MLLP-over-TLS, TLS 1.2 or higher, opt-in mTLS, and a fail-closed off-loopback bind guard. See DEPLOYMENT.md.
Validation & load tooling (`generate` , `check` , `dryrun` , the test harness, the load harness)	See sections 8/9 and LOAD-TESTING.md.
Operational metrics endpoint (`/metrics`)	Exposes engine metrics for scraping (section 12).
Console **Alerts** and **Dead-Letters** pages	Triage dead-letters and manage alerts from the desktop console (section 12).
Windows-service deployment	See SERVICE.md.
Active-passive HA / failover	Opt-in leader/standby cluster on a shared server database (PostgreSQL or SQL Server): only the leader runs the graph, with a self-fencing leadership lease and immediate on-promotion recovery. Single-node stays the default. See CLUSTERING.md and section 14.

Deployment guidance and honest caveats

- **Raw TCP and X12 transports are plaintext on the wire.** Keep them on a trusted network segment or front them with a TLS-terminating proxy. (API and MLLP do have native TLS — see section 6 and DEPLOYMENT.md.)
- **ACK semantics are ACK-on-receipt.** The sender is acknowledged once the message is durably committed; any later routing/transform/delivery failure surfaces as a message disposition + alert, not a NAK. Operators monitor disposition and alerts for post-ingress failures, not the ACK (section 7).
- **Server-database deployments are greenfield.** There is no automatic in-place migration of SQLite history into a server database — drain and cut over deliberately (section 13).
- **Throughput is hardware-dependent.** A baseline and tuning method is published (TUNING-BASELINE.md) as a two-tier reference: host-independent conformance invariants plus performance numbers measured on the reference configuration. Those throughput figures are measurements on that reference config, not a promise for your hardware — **measure on your own hardware** (section 9).

---

3. Prerequisites & environment checklist

Consolidate these before you install anything:

- [ ] **Python 3.11+** on the engine host.
- [ ] **OS:** Windows is the primary supported/serviced platform; the engine itself is cross-platform Python.
- [ ] **Administrator/elevation** on the host if you will install the Windows service.
- [ ] **Outbound network access** for the service installer to download its SHA-256-pinned dependency (or pre-stage it on the host / on `PATH`).
- [ ] **Firewall plan:** open your inbound MLLP listener port(s) (the samples use e.g. `2575` / `2600`) to senders, and decide who may reach the **API on** `127.0.0.1:8765` (default loopback — keep it that way unless you configure TLS; see section 6).
- [ ] **A writable data directory** for the store + logs (service default: `C:\ProgramData\MessageFoundry`).
- [ ] **Backend decision (made here, not later): SQLite** (default, zero extra deps) for a single-node deployment, or a server database — **PostgreSQL** (`messagefoundry[postgres]`, pure-Python) or **SQL Server** (`messagefoundry[sqlserver]` + OS-level ODBC Driver 18) — if you want a server store or a path toward DB-tier HA.
- [ ] If you will run the **cluster** path (active-passive failover): **NTP time sync** across nodes is a hard prerequisite, every node needs the **same config dir**, and `[store].backend = "postgres"` or `"sqlserver"`. (Single-node deployments skip this entirely; see section 14.)
- [ ] A **PHI encryption key** plan (section 6) and a **backup target + key-escrow** plan (section 10) decided before any real data flows.

---

4. Installation

> **New here? Start with the Installation Guide** — the focused walkthrough of installing the engine and standing up your own private **config repo**, including running multiple instances from one repo. This section is the rollout-oriented summary of the same material.

Full reference: the Installation Guide and SERVICE.md. The essentials:

4.1 Install the engine

MessageFoundry is a **version-pinned dependency**: install a published wheel and **pin the exact version**, the same way you pin any other production dependency. Create a venv and install:

```
python -m venv .venv
.\.venv\Scripts\Activate.ps1
pip install "messagefoundry==0.2.0rc1"        # pin the exact engine version (core runtime only)
```

The pinned install pulls only the **core runtime** — what a headless engine needs. Add extras (section 4.2) for the desktop console, a server-database backend, or SFTP. `pip install messagefoundry` from public PyPI is the standard install path; you can equally install from the engine's **GitHub Release assets**.

Verify the release before you install. MessageFoundry ships one signed wheel to many PHI-bearing instances, so verify the artifact's provenance *before* installing — pinning a version (or a hash) proves you got a *fixed* file, not that it is the one MessageFoundry built. Every release carries **SLSA build provenance** and a **Sigstore signature** via token-less Trusted Publishing; check both with the **GitHub CLI** (`gh` ≥ 2.49), and optionally `sigstore` (`pip install sigstore`). Install **only** the file that passes:

```
$V = "0.2.0rc1"   # the exact version you intend to install

# Download the wheel + its Sigstore bundle from that release's assets
gh release download "v$V" --repo MEFORORG/MessageFoundry `
  --pattern "messagefoundry-$V-*.whl" --pattern "messagefoundry-$V-*.whl.sigstore*"

# Verify SLSA build provenance:  artifact -> source commit -> builder workflow
gh attestation verify "messagefoundry-$V-py3-none-any.whl" --repo MEFORORG/MessageFoundry

# (defense in depth) Verify the Sigstore signature pins the release workflow identity
python -m sigstore verify identity "messagefoundry-$V-py3-none-any.whl" `
  --cert-identity "https://github.com/MEFORORG/MessageFoundry/.github/workflows/release.yml@refs/ta
  --cert-oidc-issuer "https://token.actions.githubusercontent.com"

# Only if BOTH pass, install the exact file you verified
pip install ".\messagefoundry-$V-py3-none-any.whl"
```

The same attestation also covers the **public PyPI** copy (it is byte-identical), so you can `pip download "messagefoundry==$V" --no-deps -d .\verify`, `gh attestation verify` the downloaded wheel, then `pip install --no-index --find-links .\verify "messagefoundry==$V"`. A registry/mirror substitution or a relabelled file **fails** the check. (The `--cert-identity` ref must match the tag you install.)

For a **reproducible pinned** deploy, generate a hash-locked requirements file scoped to the extras you actually run and install it with `--require-hashes`. The scaffolded config repo (`messagefoundry init`, below) already pins the engine in its `requirements.txt` — extend that into a full hash-lock for your host.

4.2 Optional extras

Extra	Pulls in	When
`postgres`	`asyncpg` (pure-Python, no OS dep)	Using the PostgreSQL backend (recommended production path).
`console`	desktop UI + keyring	Running the desktop admin console.
`sftp`	paramiko	SFTP connectors.
`sqlserver`	`aioodbc` **+ OS-level Microsoft ODBC Driver 18**	The SQL Server *store* backend (`backend=sqlserver`, production) and the Database connector family.
`dev`	pytest/ruff/mypy/httpx	Development & CI.

> ⚠️ If you set `backend=postgres` but forget `pip install 'messagefoundry[postgres]'`, you get a raw `ImportError` at startup. Install the extra together with the backend.

Start your own config repo (`messagefoundry init`)

Section 4.1 installed the **engine**. Now scaffold the other half a deploying organization owns — your **own** separately-versioned **config repo** — which holds your Connections/Routers/Handlers and drives one or more engine instances. **This is the recommended way to run MessageFoundry:** the `samples/` directory used in quickstarts ships only in a source checkout, **not in the installed wheel**, so a wheel install runs against *your* `config/`, not `samples/`.

```
messagefoundry init ./my-config-repo
```

It writes a runnable starter feed (`config/`), `environments/<env>.toml` value stubs, a synthetic fixture, an instance `messagefoundry.toml` (active environment + posture), a `requirements.txt` pinning this engine version, a CI `check` workflow, and `.vscode` settings — so `messagefoundry check --config config --messages messages/sets` is green from the first commit. See the generated `README.md` for the day-to-day workflow.

4.3 Run it (foreground, to learn the ropes)

From your config repo (the one `messagefoundry init` created), run the engine against its `config/` directory:

```
cd ./my-config-repo
python -m messagefoundry serve --config config --db ./messagefoundry.db --env dev
```

`serve` flags and their precedence (**CLI > `MEFOR_<SECTION>_<KEY>` env > `messagefoundry.toml` > built-in default**): `--config` (your graph directory), `--service-config` (default `./messagefoundry.toml` if present), `--db`, `--host`, `--port`, `--log-level`, `--env` (a **free-form** environment name), `--allow-insecure-bind`.

> ⚠️ **The active environment is required.** `serve` refuses to start (exit 2) without `--env <name>` (or `[ai].environment`) — there is no silent `prod` default, so a missing env can never resolve another environment's values/secrets. Built-in names `dev` / `staging` / `prod` carry a default posture; a custom name (e.g. `test`, `poc`) also needs `[ai].data_class` + `[ai].production`. The active environment is logged at startup.

4.4 Run it as a Windows service (the supported production run-mode)

Use the elevated installer; install under a **least-privilege virtual account** rather than the default LocalSystem:

```
# from an elevated shell
scripts\service\install-service.ps1 -ServiceAccount "NT SERVICE\MessageFoundry"
```

The installer is idempotent, auto-downloads a SHA-256-pinned service shim, bakes absolute `serve` paths into the service, and (with `-ServiceAccount`) auto-grants config-read + data-dir-read/write to the account. Service defaults: name `MessageFoundry`, data dir `C:\ProgramData\MessageFoundry`, store `<DataDir>\messagefoundry.db`, logs `<DataDir>\logs`, bind `127.0.0.1:8765`.

> ⚠️ **Pinned-wheel operational model.** With a pinned-version install (section 4.1), the running service loads the **installed wheel** — a known, pinned version. Picking up a new engine version is a deliberate `pip install "messagefoundry==<new>"` + service restart (section 13), so every upgrade is an explicit, reviewable act.

4.5 First-run admin bootstrap

Auth is **enabled by default**. On the first start against an empty store, MessageFoundry creates a one-time bootstrap admin (`admin`) and writes its password to an **owner-only** `bootstrap-admin.txt` next to the store (only the file *location* is logged — never the password). Then:

1. Log in as `admin`; you are **forced to change the password** on first use, and to enrol the required MFA factor.
2. **Create a second real administrator** promptly.
3. **Delete** `bootstrap-admin.txt`.

The bootstrap account auto-retires once a second admin exists, or — while still unclaimed — 72h after creation.

4.6 Verify it runs

```
curl http://127.0.0.1:8765/health           # -> {"status":"ok"}
# tail <DataDir>\logs\service.out.log for the "wiring started" banner
```

Then send a synthetic message to confirm the end-to-end path. The scaffolded starter feed listens on MLLP `2575` and ships a PHI-free fixture at `messages/sets/example_adt.hl7` — send it with any MLLP client. The convenience senders ship with the **engine source checkout**, not the installed wheel.

If start fails, check `service.err.log` first — the common causes are relative paths resolving to the system dir under a service account, a busy MLLP/API port, or a data dir the account can't write.

---

5. Minimum viable configuration

Full reference: CONFIGURATION.md (service settings) and CONNECTIONS.md (the graph).

There are two distinct configuration surfaces:

1. **The message graph (Python modules)** in your `--config` directory. The minimum first flow is one module: an `inbound()` with a transport spec and a `router=` binding, a `@router` that returns handler name(s), and a `@handler` that returns `Send(...)` to a declared `outbound()`. The scaffolded repo (`messagefoundry init`, section 4) gives you a working `config/IB_EXAMPLE_ADT.py` to start from. The loader globs `*.py` (non-recursive; skips `_*`-prefixed helper files), then merges an optional `connections.toml`.
2. **Service/operational settings** in `messagefoundry.toml` (+ `MEFOR_*` env + CLI). Keep **all secrets out of this file and out of source control** — supply them via `MEFOR_<SECTION>_<KEY>` env vars (the loader *warns* if it sees a known secret in the file).

Guidance for a clean first flow:

- **Use the MLLP/File pair** for an initial end-to-end test — both need no extras. The Database connector family adds the `sqlserver` extra + ODBC Driver 18; MLLP/File keep the first hop dependency-free.
- **Never set a host on an inbound MLLP/TCP connection** (it is a config error). Set the listen interface once, service-side, via `[inbound].bind_host` (loopback for dev; a specific NIC behind a firewall for prod). Outbound MLLP/TCP *do* take the downstream host.
- **Author anything environment-specific as `env("key")`**, put non-secret values in `environments/dev.toml` / `environments/prod.toml` with identical keys, and inject secrets only via `MEFOR_VALUE_<KEY>`. A referenced-but-undefined key fails loud at load. Use `current_environment()` (not `env()`) inside a handler to branch on the deployment.
- **`connections.toml` (data) is optional:** move *transport config* there if you want hand/GUI editing; keep *logic* (routers/handlers) in `.py`. A name declared in both a module and `connections.toml` is a hard error (no silent shadowing).

- **The `--config` directory is a trust boundary.** `serve` and `POST /config/reload` **execute** the Python in it, in-process, as the service account. On POSIX the loader refuses a group/world-writable config dir; **on Windows this is your responsibility** — lock the directory's ACL to admins + the service account.

---

6. Security & PHI hardening before real data

Full references: SECURITY.md, PHI.md, and DEPLOYMENT.md (network exposure). MessageFoundry ships real auth with required MFA for local accounts, RBAC, audit, opt-in at-rest encryption, **native TLS** (API + MLLP, with a fail-closed off-loopback bind guard), and interface authentication via mTLS, OAuth 2.0 client-credentials, and SMART-on-FHIR Backend Services. The features here support a **HIPAA-compliant deployment**; compliance is a property of how you operate the system, so complete this checklist **before any real PHI flows**:

- [ ] **API off-loopback requires native TLS.** The API binds `127.0.0.1` by default. To reach it from another host, configure **in-process TLS** (`[api].tls_cert_file` + `[api].tls_key_file` , `tls_min_version` ≥ 1.2, opt-in mTLS via `tls_client_ca_file`) **or** front it with a TLS terminator (`[api].tls_terminated_upstream = true` + `[api].trusted_proxies`). A non-loopback bind **without** TLS (or a trusted terminator) is **refused at startup**. **Never use `--allow-insecure-bind` for real PHI** — it is a loud dev-only escape that puts bearer tokens and PHI on the wire in cleartext.
- [ ] **MLLP off-loopback requires native TLS too.** MLLP-over-TLS is built: set `tls = true` + `tls_cert_file` / `tls_key_file` per connection (opt-in mTLS via `tls_ca_file` ; ≥ TLS 1.2). MLLP is **plaintext by default**, and a non-loopback plaintext MLLP bind is refused. **Raw TCP and X12 have no transport TLS** — keep them on a trusted segment or proxy-terminate. Full matrix: DEPLOYMENT.md.
- [ ] **Turn on at-rest encryption and make it mandatory:** mint a key with `messagefoundry gen-key` (or a Windows DPAPI-protected key file via `messagefoundry protect-key`), set `MEFOR_STORE_ENCRYPTION_KEY` , **and** set `[store].require_encryption = true` so the engine refuses to start unencrypted.
- [ ] **Enable volume encryption (BitLocker/LUKS).** App-level encryption protects message *bodies*; the `summary` / `control_id` / `message_type` columns and the `-wal` / `-shm` /temp files are **not** app-encrypted and rely on volume encryption.
- [ ] **Run under a least-privilege account** (the virtual account from section 4.4) and lock down the store directory and any File-connector spill directories. **Treat backups as PHI.**
- [ ] **Finish the bootstrap-admin handoff** (section 4.5): change the password, enrol MFA, create a second admin, delete `bootstrap-admin.txt` .
- [ ] **For Active Directory:** use **LDAPS** with a trusted CA, never set `MEFOR_ALLOW_INSECURE_TLS` in production, and configure the directory's lockout/complexity policy. AD users authenticate (and get MFA) through your own identity provider via SSO federation; the engine's built-in MFA and account lockout cover local accounts.
- [ ] **Populate the fail-closed `[egress]` allowlist** (it defaults to unrestricted) for REST/Database destinations.

- [ ] **Keep logging at `INFO` or above** and `expose_docs` off in production. Full payloads are never logged at INFO+ by design — **do not raise the service to DEBUG with real PHI**.
- [ ] **Author routers/handlers so they never interpolate raw HL7 into an exception message** (it can surface in `last_error` / `detail`).
- [ ] Run `messagefoundry audit-verify` periodically (the audit log is tamper-*evident*, not tamper-*proof*), and set `[retention]` windows — they are **off by default (kept forever)**.

> **Security posture.** MessageFoundry has completed an internal **OWASP ASVS 5.0 Level 3 self-assessment** (212 requirements met, 0 failed, 0 partial, 133 not applicable, of 345). This is a self-assessment, **not** an external audit. ASVS Level 3 recommends — but does not require — an independent code review and penetration test; both are planned. See SECURITY.md.

---

7. Reliability configuration — how nothing gets lost

This is the heart of operating the engine safely. The durability model is a **transactional staged queue** (no external broker): each message flows ingress → routed → outbound, with every handoff a single committed transaction, giving **at-least-once** delivery with crash-safe re-runs. Details in ADR 0001.

Key semantics to internalize:

- **ACK-on-receipt.** The sender is acknowledged (`AA`) as soon as the raw message is durably committed (after synchronous decode/parse/optional strict-validate, which still NAK). **Any routing/transform/delivery failure happens *after* the ACK** and surfaces as an internal **disposition + alert**, never a NAK. Operators monitor disposition + alerts, **not** the ACK, for post-ingress failures.
- **Disposition lifecycle:** `RECEIVED` → `ROUTED` / `UNROUTED` → `PROCESSED` / `FILTERED` / `ERROR`. The store finalizer is the **sole authority** and never finalizes while any stage row is still in flight. Note: a single dead row at *any* stage flips the whole message to `ERROR` **even if a sibling handler delivered** — so read the **per-message event trail**, not just the headline status.
- **Failure classification & policy (per outbound):**
- Permanent partner reject (`AR` / `CR`) → **dead-letter immediately** (still replayable).
- Transient (`AE` / `CE`) or transport error → **retry per `RetryPolicy`**.
- Internal/code error → either **STOP** the lane and raise a `connection_stopped` alert, or **CONTINUE** (auto-dead-letter the bad message and keep flowing).
- `RetryPolicy.max_attempts` **unset = retry forever** (nothing silently lost) with exponential backoff. Under the default **FIFO** ordering, a permanently-failing head **blocks its lane** until it succeeds or is purged.

Mandatory before go-live:

- [ ] **Wire real alerts.** Configure the `[alerts]` **webhook and/or email** notifier — do **not** rely on the default logging-only sink. The conservative defaults (FIFO head-of-line blocking, retry-forever, STOP-on-internal-error) are only safe if a human gets paged when a lane stalls.

- [ ] **Set `[delivery]` buildup thresholds** (`max_oldest_seconds` defaults to 300s; set a `max_depth` sized to each connection's throughput) so `queue_buildup` fires before a stuck lane silently backs up. Buildup detection covers the ingress and routed stages too, not just outbound.
- [ ] **Choose `RetryPolicy` per outbound deliberately:** retry-forever for partners that must never lose a message (accept head-of-line blocking + rely on buildup alerts), or a finite `max_attempts` where stale data is worse than a replayable dead-letter.
- [ ] **Choose `InternalErrorPolicy` intentionally:** `CONTINUE` (default) for high-volume feeds where uptime matters most; `STOP` for low-volume feeds where ordering/no-loss matters more than uptime.
- [ ] **Code routers/handlers as pure and idempotent.** At-least-once means a message can re-run after a crash or a replay. No side-effecting writes mid-transform; the **one** allowed exception is a **live, read-only DB lookup**. Downstream connectors/partners must **dedupe** (e.g. on MSH control id).

Recovery tools you should know cold: `/dead-letters` (triage) + `/dead-letters/replay` (bulk *outbound* recovery), and per-message `/messages/{id}/replay` (for dead ingress/routed rows — router/transform errors, undecryptable raw, a removed handler). Startup automatically returns stale in-flight rows to pending (crash recovery) and dead-letters rows whose destination/handler left the config. The console **Dead-Letters** page surfaces this triage in the UI (section 12).

---

8. Pre-traffic validation

Prove correctness **before** any network traffic. None of this should ever run against real PHI — `generate` / `dryrun` can emit full message bodies; never redirect their output to a committed file or CI log.

1. **Build a synthetic corpus:** `messagefoundry generate --type ADT --count 50 --out <fixtures>` (conformant HL7 v2.5.1, validated against hl7apy; 13 message types, 57 ADT triggers; PHI-free).
2. **Gate the config in CI / a pre-commit hook:** `messagefoundry check --config <dir> --messages <fixtures>` . - `validate` (every module loads; every inbound→router reference resolves; no port collisions) is **required and blocking**. - `dryrun` is **required only when you supply a fixtures dir containing `*.hl7` — without fixtures the dryrun is silently skipped** and the gate passes on `validate` alone, so a transform that errors at runtime is *not* caught. **Build and maintain the fixtures.** - `ruff` / `mypy` are advisory (never block).
3. **Inspect the wiring:** `messagefoundry validate --json` (all problems at once) and `messagefoundry graph --config <dir>` (confirm the wired graph matches intent).
4. **Confirm dispositions:** `messagefoundry dryrun` runs the same core the live engine runs (no I/O), so dry-run and live route identically. Then exercise the **test harness**: its 5 headless `--scenario` runs (`processed` / `filtered` / `unrouted` / `error` / `dead_letter`) assert dispositions over the API for CI, and its GUI can inject delivery faults (delay-then-AA, close, fail-N-then-AA) to prove your **retry / dead-letter / replay** behavior before you trust it.

Note: `validate` only catches **literal** port collisions; `env()` -resolved ports are checked at bind time. A `prod` -only missing `env()` value may not surface during a `dev` -context check — validate against the target

environment before promoting.

---

9. Capacity & load testing on *your* hardware

Full references: LOAD-TESTING.md and the published throughput baseline & tuning reference — a **two-tier reference**: host-independent **conformance** invariants (zero loss, bounded drain, low error rate) plus **performance** numbers *as measured on the reference config*. Because the durable-write path is hardware-dependent, those throughput figures are **not** a promise for your hardware — establish your own baseline.

The headless load harness drives an already-running engine over real MLLP and the HTTP API (it never touches the store), so it is **store-agnostic** — swap the engine's `--db` to compare SQLite vs Postgres ceilings on identical traffic.

Recommended ramp:

1. `smoke` — tiny zero-loss wiring check (no performance claim).
2. `fanout-baseline` — warmup → ramp → sustained → spike → recovery; SLOs are evaluated only on the measured sustained phases. Reference targets in the profile: ≥200 msg/s sustained, ACK p99 ≤50ms, e2e p99 ≤5s, error ≤0.001, drain ≤60s, zero-loss. (These are reference-config measurements, not a promise for your box.)
3. `soak` — ~1-hour steady state watching DB/WAL growth + dead-letter accumulation.

Treat the **zero-loss reconciliation** (`sent == engine_read` , `sink_received == engine_written` , backlog drained to zero) as the **headline gate** — throughput numbers are meaningless if messages were lost. Use a **closed-loop** phase (fixed concurrency) to find your true max sustainable throughput, and the `slow` transform mode to find your per-core transform ceiling. Save the JSON/CSV reports and use `--baseline` + `--tolerance` to catch regressions over time. Size `correlator_capacity` above your peak in-flight (watch for correlation-miss notes), and remember a single Python sender process is the offer ceiling — shard it across processes if it can't saturate your engine.

Sizing reality: the staged pipeline has ~**3× write-amplification** on SQLite (3 commits for a common single-handler message; 2 + H for an H-way fan-out) — see the write-amplification benchmark. Plan disk headroom for `.db` + `-wal` , plan retention/VACUUM (section 10), and move to **Postgres** if a single-writer SQLite ceiling becomes the bottleneck.

---

10. Backup, restore & disaster recovery

> Backup and DR are part of *your* operational responsibility. Rehearse a full restore before you carry real data.

Back up the store.

- **SQLite:** the WAL backend means three files must be captured **consistently** — `.db`, `.db-wal`, `.db-shm`. Use `sqlite3 <db> ".backup '<dest>'"` against the live DB, or take a **quiesced cold copy** (graceful stop → copy → restart). A naive copy of just the `.db` while the service runs can be inconsistent.
- **PostgreSQL:** use your standard DB backups — `pg_dump` for logical backups and/or WAL archiving / PITR for point-in-time recovery. Server-database deployments are greenfield, so the DB tier owns store-level DR here.

Escrow the encryption key SEPARATELY. If you enabled at-rest encryption (section 6), a restored store is **unreadable without the same `MEFOR_STORE_ENCRYPTION_KEY` / DPAPI key file**. Back the key up in a different location/system from the data, with its own access control.

Restore-and-verify drill (do this in the lab, section 11 Stage 0): restore the store + key into a clean host, start the engine, confirm `/health`, run `/status/integrity-check` (SQLite `PRAGMA quick_check`), and spot-check `/messages` and dispositions.

Keep the store bounded. `[retention]` is **off by default (kept forever)**. Set `max_db_mb` (drives a `storage_threshold` alert), `messages_days` / `dead_letter_days` (body purge), and the daily VACUUM so the store doesn't grow unbounded and a full disk doesn't take you down mid-operation.

---

11. Staged rollout plan with go/no-go gates

This is the recommended path from first install to full production. **Do not skip stages** — each one exists to catch a different class of problem cheaply. Advance only when the stage's **exit criteria** are met.

Stage 0 — Lab / standalone

Goal: prove wiring, dispositions, and recovery on a throwaway box, with **synthetic data only**.

Setup: SQLite, loopback, auth on, a synthetic corpus from `messagefoundry generate`.

Exit criteria (→ Stage 1): - [ ] `messagefoundry check --config <dir> --messages <fixtures>` exits 0 (validate **and** dryrun green). - [ ] All 5 disposition `--scenario` runs pass (`processed` / `filtered` / `unrouted` / `error` / `dead_letter`). - [ ] You have driven a retry → dead-letter → **replay** cycle via the harness fault injection and understand the recovery tools (section 7). - [ ] A **backup + restore** has been rehearsed once (section 10).

Stage 1 — Shadow / parallel run

Goal: run MessageFoundry alongside your **incumbent** engine on **real production traffic** without affecting any downstream system.

How: tee/duplicate the production **inbound** feed to a MessageFoundry instance whose outbounds point at a **throwaway/null sink** (the harness correlation sink works well), or use a router that `Send`s only to a dedicated "shadow" outbound. Compare MessageFoundry's dispositions and transformed output against the incumbent's outcomes for the same messages.

> ⚠️ **Do not dual-*write* to real partners in shadow.** At-least-once + non-idempotent downstreams make a true dual-write dangerous. Keep shadow outbounds pointed at a sink unless the partner dedupes.

Exit criteria (→ Stage 2): - [ ] **Zero-loss reconciliation holds** over a sustained window (e.g. 1–2 weeks) at production volume. - [ ] MessageFoundry dispositions/output **match the incumbent's** for the same messages (differences explained). - [ ] **No unexplained dead-letters**; every `ERROR` understood. - [ ] A load test on **production-like hardware** meets your own SLO targets (section 9).

Stage 2 — Limited production

Goal: MessageFoundry becomes the system of record for a **small, low-risk subset** of real feeds (one partner / one low-volume interface).

Prereqs: switch to **Postgres** if you need a server DB; **encryption on** (`require_encryption=true`); **alerts wired** and **monitoring in place** (section 12); **backups automated**; **upgrade + rollback runbook validated on staging** (section 13).

Exit criteria (→ Stage 3): - [ ] e2e p99 within your SLO; **zero unexplained dead-letters** over the observation window. - [ ] **Alert wiring proven by a deliberate fault-injection drill** — you triggered `queue_buildup` / `connection_stopped` and the on-call was actually paged. - [ ] **Backup + restore rehearsed against the production store** (not just the lab copy). - [ ] Rollback runbook exercised at least once.

Stage 3 — Full production

Goal: migrate the remaining feeds **in waves** (never big-bang). Keep decommissioning the incumbent as a **separate, later** step so you retain a fallback.

Steady-state expectations: - [ ] Sustained-load SLO met on production hardware. - [ ] DR (backup/restore) rehearsed and scheduled. - [ ] On-call + the failure-drill runbook (section 12) in place. - [ ] If you require HA: stand up the **active-passive** cluster (leader/standby on a shared server-database store — PostgreSQL or SQL Server) via the **section 14 HA rollout runbook** — VIP standup + both failover drills passed — and keep HA at the DB tier too. Decided and rehearsed before you depend on it.

---

12. Day-2 operations & monitoring

Verify-it-runs (after every start/restart): `GET /health` → `{"status":"ok"}` , send a synthetic message, and confirm the **"wiring started"** banner in `service.out.log` .

Monitoring surfaces: - `/metrics` — operational metrics for scraping. - `/stats` — outbox counts by status. - `/status` — DB size vs disk free, journal mode, counts. **Scrape db-size-vs-disk-free.** - `/status/integrity-check` — on-demand store integrity. - `/ws/stats` — ~1 Hz queue-depth WebSocket. - `/messages` + `/messages/{id}` — per-message detail and the **event trail** (read this, not just the status — see section 7). - `/dead-letters` — **page on dead-letter accumulation.** - **AlertSink events** to alert on:

`connection_stopped`, `queue_buildup`, `storage_threshold` (wire these to webhook/email — section 7). - `service.err.log` — watch it.

The desktop console includes **Alerts** and **Dead-Letters** pages for in-UI alert management and dead-letter triage; the CLI/API remain available for scripted operations.

Log management: logs land under `<DataDir>\logs`. Configure rotation, keep the level at `INFO` or above (DEBUG can leak PHI — section 6), treat `service.out/err.log` as **potential-PHI artifacts** (ACL them), and include them in your retention policy.

Graceful drain for maintenance: stopping the service (Ctrl+C / service stop) triggers the ASGI lifespan to call `engine.stop()` for a clean drain. Always **drain → stop → back up → change → restart → verify**.

Failure-drill runbook — rehearse these in the lab/staging before prod:

Symptom	First moves
Stuck FIFO lane (retry-forever head)	`queue_buildup` alert → inspect `/messages` for the head → fix-and-`replay`, or `dead_letter_now` to unblock the lane (the dead row stays replayable).
Poison message	It dead-letters under `CONTINUE` (or stops the lane under `STOP`) → triage via `/dead-letters` → fix the transform → replay.
Full disk / `storage_threshold`	Free space / tighten `[retention]` / VACUUM → confirm `/status` disk-free recovers.
Crash / unexpected restart	Startup auto-recovers in-flight rows → verify `/health`, the "wiring started" banner, and that backlog drains.
Planned maintenance	Drain-and-stop (graceful), perform the change, restart, verify.

---

13. Upgrade & rollback

Safe upgrade runbook (pinned-wheel model): 1. **Drain** inbound (quiesce senders or stop accepting new work) and confirm queues are draining. 2. **Stop** the service (graceful). 3. **Back up** the store **and** the encryption key (section 10). 4. **Bump the pinned engine version:** update the pin in your config repo's `requirements.txt` (`messagefoundry==<new>`) and `pip install "messagefoundry==<new>"` into the deployment venv. 5. **Re-validate:** run `messagefoundry check` against your config (and `ruff` / `mypy` / `pytest` too if you develop the engine). 6. **Restart** and **verify** (`/health`, "wiring started" banner, `/status`).

Rollback: - **Config rollback** is the cheapest lever: the audited `POST /config/reload` does a quiesce-and-swap to a known-good `--config` directory (confined to the allow-listed reload roots). Keep your last known-good config dir available. - **Engine rollback:** re-pin the prior version (`pip install "messagefoundry==<prev>"`) → restart (same runbook above). - ⚠️ **Schema/store-level changes are not trivially reversible** against a populated store given the greenfield-only posture (no in-place migration). Plan code/config rollback as your primary path; use **dead-letter replay** to recover messages that a bad transform stranded before the rollback.

Release cadence: pin a released version (`messagefoundry==X.Y.Z`); the **latest release** is the supported target. Reproduce a problem against the latest release before filing an issue, and keep upgrades **small and frequent** rather than large and rare.

---

14. High availability — rollout & VIP standup runbook

Single-node is the default and is genuinely reliable — the durable staged queue (section 7), not clustering, is what guarantees no message is lost on one node. Reach for HA only when you need **failover** (an unattended node loss must not stop intake), **not** for throughput (for that, scale intra-node — see the end of this section). When you do need it, MessageFoundry ships an **opt-in active-passive cluster** — the supported HA model. This section is the operational runbook; the authoritative topology, lease semantics, and full settings catalog are in CLUSTERING.md.

The model in one paragraph. Run N identical engine processes against **one shared server database** (PostgreSQL or SQL Server) with `[cluster].enabled = true`. Exactly one node — the **leader/primary** — runs the whole message graph (every listener *and* the router/transform/delivery workers); the rest are **warm standbys** that only contend for leadership (heartbeat + cache convergence), binding no listeners and running no workers until they win it. A **self-fencing leadership lease** in the shared DB elects the one primary and guarantees a partitioned old primary stops before a standby takes over. There is no separate broker and no node-to-node socket — coordination rides the shared `[store]` connection.

14.1 Stand up the cluster

1. **Pick a server-DB backend and make the DB tier itself HA.** `[store].backend = "postgres"` or `"sqlserver"` (SQLite is rejected for clustering). MessageFoundry coordinates the *processing* leader; it does **not** replicate your store — delegate store HA to the database (**PostgreSQL streaming replication / managed-Postgres HA**, or **SQL Server Always On**) and rehearse a DB failover separately.
2. **Provision every node identically.** Same installed engine version, the **same config dir**, and the **same `[store]` target** (server/database/schema) on each host. Set `[store].pool_size ≥ 2` (**≥ 3 recommended** — the leader drives the maintenance loop + reclaim sweep + workers concurrently).
3. **Enable the cluster** in the shared `messagefoundry.toml` : `toml [cluster] enabled = true heartbeat_seconds = 10.0 # keep the invariant: heartbeat < fence < ttl leader_fence_timeout_seconds = 20.0 # a leader that can't renew self-fences here (split-brain guard) leader_lease_ttl_seconds = 30.0 # a standby may acquire only once the lease has expired` The defaults trade a ~30 s crash-failover for margin; lower all three **proportionally** for faster failover at the cost of tolerance for a slow DB / GC pause.
4. **Sync clocks (NTP).** Row leases use node wall-clock — keep nodes synced so a lease expiry isn't mistimed. (Leadership is evaluated on the *DB* clock, so it's skew-immune; the row leases are not.)
5. **Start the same `serve` command on every node** (run each as its own Windows service — see SERVICE.md). Confirm membership: `GET /cluster/nodes` lists every node and names exactly one `leader_node_id` ; each node's `GET /cluster/status` reports `role: "primary"` on one and `"standby"` on the rest.

14.2 Stand up the VIP / load balancer (required)

Like other clustered integration engines, senders must reach "the engine" through a **floating VIP / L4 load balancer**, never a fixed node — because **only the primary binds the inbound listener ports**, the VIP is what makes a failover transparent (modulo a reconnect). **MessageFoundry ships the bind behavior and the `/cluster/*` endpoints, but does not ship a load balancer** — you stand one up (keepalived + IPVS, HAProxy in `tcp` mode, an F5, a cloud **L4 / network** LB, …).

Data plane (MLLP / raw-TCP / X12 inbound) — one VIP per listener port:

- [ ] **Mode = L4 / TCP pass-through.** These are byte streams — do **not** front them with an HTTP/L7 LB.
- [ ] **Health check = a plain TCP connect to that exact listener port** on each backend node. Because only the primary binds the port, exactly one backend is ever healthy, so the VIP routes there automatically; on failover the old primary's port closes (goes unhealthy) and the new primary's opens (goes healthy) and the VIP follows. An application-level (MLLP-message) probe is unnecessary — a TCP connect is the correct, sufficient signal.
- [ ] **Tune the probe to your failover budget.** The VIP repoints in roughly `check_interval × unhealthy_threshold`; size it well under your acceptable outage and above your DB/GC jitter (a 2–5 s interval is typical).
- [ ] **If MLLP-over-TLS is on, keep TLS end-to-end** (TCP pass-through) so the engine's own cert is presented; don't terminate TLS at the VIP unless you re-encrypt to the backends.
- [ ] **Don't let the LB reap long-lived MLLP sockets** — set TCP idle timeouts generous enough for persistent connections.
- [ ] **Verify every partner reconnects-on-drop.** On failover, connections to the old primary drop and senders must redial the VIP. This is standard MLLP client behavior — confirm it per partner.

Management plane (console / IDE → engine API) — optional VIP:

- [ ] The API is a control/read plane over the shared DB and is **up on every node**, so an API VIP can health-check the unauthenticated `GET /health` (liveness); you can point the console at any node.
- [ ] To pin operators to the active leader, read `GET /cluster/status` → `role` (`primary` / `standby`) or `GET /cluster/nodes` → `leader_node_id` / `lease_owner` (the console already surfaces the live primary). Both endpoints need only `MONITORING_READ` (VIEWER and up; no PHI).

14.3 Rehearse & validate failover (before real traffic)

Failover is **not instantaneous** — quantify *your* window from these drills; don't assume zero-downtime.

- [ ] **Clean switchover** (planned): gracefully stop the primary's service. It **expires its lease**, so a standby promotes on its next heartbeat (**≈ one `heartbeat_seconds`**). Watch `leader_node_id` move on `/cluster/nodes`, watch the VIP repoint, and keep synthetic traffic flowing throughout.
- [ ] **Crash** (unplanned): hard-kill / power off the primary. Its lease **ages out**, so a standby promotes after **up to `leader_lease_ttl_seconds`** (~30 s default); a partitioned old primary **self-fences within `leader_fence_timeout_seconds`** so it can't double-process. Confirm the new primary's **owner-scoped**

on-promotion recovery re-pends the dead primary's in-flight rows immediately (delivery resumes without waiting out the per-row lease TTL).

- [ ] **Zero-loss across the failover.** At-least-once means a row interrupted mid-delivery is **re-delivered** after its lease expires — so confirm your downstream connectors **dedupe / are idempotent**, then reconcile sent-vs-delivered across the drill.
- [ ] **Record the measured window** and, if needed, tune `heartbeat < fence < ttl` down proportionally and re-drill.

14.4 HA go-live checklist

- [ ] Server-DB backend; **DB-tier HA configured and its own failover rehearsed**.
- [ ] Identical engine version + config dir + `[store]` target on every node; `pool_size ≥ 3`; **clocks NTP-synced**.
- [ ] `/cluster/nodes` shows all nodes and exactly one `leader_node_id`.
- [ ] **VIP per inbound port** with a **TCP-connect health check**; partner **reconnect-on-drop** verified.
- [ ] **Clean-switchover drill** passed (≈ one heartbeat, VIP follows, zero loss).
- [ ] **Crash drill** passed (failover within the TTL, no double-processing, in-flight rows recovered, zero loss).
- [ ] Measured failover window documented; lease timings tuned to your network.
- [ ] **Config changes applied as a coordinated (non-rolling) restart** — all nodes restart together so they never run divergent graphs across the change window.
- [ ] `/cluster/status` + `/cluster/nodes` monitored (alert on **no live leader** beyond a failover window) alongside the section 12 surfaces.

For throughput (not failover), scale intra-node: one independent delivery worker per outbound connection (a slow/failing lane never blocks siblings), and keep retry policies finite where head-of-line blocking on a shared FIFO lane would otherwise stall throughput. Adding cluster nodes does **not** raise throughput — only the leader processes.

---

15. Getting help & reporting bugs

- **Bugs & feature requests:** open a GitHub issue using the repository's issue templates (`bug_report.md` / `feature_request.md`).
- **Security vulnerabilities:** use the repository's **private security advisory** process per `.github/SECURITY.md` — do **not** open a public issue for a vulnerability.
- **Before filing:** verify against the latest release, and include the engine version, config shape, and relevant **non-PHI** log excerpts.
- 🔒 **Never attach real PHI** to an issue, log excerpt, or reproduction. Reproduce with a synthetic corpus from `messagefoundry generate`.

---

16. Decommissioning an environment

Tearing down an environment is a **PHI-disposal** event. `uninstall-service.ps1` removes the service but **deliberately leaves the store and logs on disk**. To tear down cleanly:

1. **Graceful drain + stop**, confirm no in-flight work remains.
2. **Uninstall the service** (`scripts\service\uninstall-service.ps1`).
3. **Securely dispose of all PHI-bearing artifacts:** the store (`.db` + `-wal` + `-shm`), any PostgreSQL database/backups, **File-connector spill directories**, the `logs` directory, every **backup copy**, and the **encryption key / DPAPI key file**.
4. **Revoke credentials** (service account, AD bind account, any API tokens).

Treat backups and the encryption key with the same disposal rigor as the live store — a forgotten encrypted backup plus its escrowed key is still recoverable PHI.

Appendix — where each topic is documented

Topic	Reference
Install + your config repo	INSTALL-GUIDE.md
System requirements / sizing by volume	SYSTEM-REQUIREMENTS.md
Install + Windows service	SERVICE.md
Network exposure / TLS	DEPLOYMENT.md
High availability / clustering	CLUSTERING.md
Throughput baseline / tuning	TUNING-BASELINE.md
Service settings / environments	CONFIGURATION.md
Connections / the graph / `connections.toml`	CONNECTIONS.md
Reliability / staged pipeline	ADR 0001
Security / auth / RBAC / audit	SECURITY.md
PHI handling / encryption	PHI.md
HL7 validation	HL7-VALIDATION.md
Load testing	LOAD-TESTING.md
Write-amplification / sizing	step-b-write-amplification.md
Architecture (authoritative)	ARCHITECTURE.md

MessageFoundry is independent and unaffiliated with Mirth, Corepoint, Cloverleaf, Rhapsody, or Ensemble; those names are trademarks of their respective owners.
