MessageFoundry User Guide

MessageFoundry is an open-source, Python integration engine for healthcare — a modern alternative to Mirth Connect and Corepoint. It **receives, routes, transforms, and validates** messages between systems (HL7 v2.x by default, payload-agnostic for other formats), with routing and handling expressed as **ordinary Python you own and version-control**. This guide is **task-oriented** ("how do I…"): it walks you from a clean machine to a running engine, then through authoring Connections/Routers/Handlers and operating and monitoring the system. It links to the reference docs rather than restating them.

Contents

What MessageFoundry is, and how to use this guide

MessageFoundry receives, routes, transforms, and validates messages between systems: **HL7 v2.x by default**, and payload-agnostic for other formats (JSON, XML/SOAP, X12 EDI, database records). Unlike a legacy engine's embedded scripting language or a locked-in low/no-code GUI, the routing and handling logic is **ordinary Python you own and version-control** — and connection setup in particular can be pure data (a TOML file, edited by hand or in a VS Code GUI). The engine runs headless as an asyncio service; you operate it from a separate PySide6 console and a VS Code extension, both talking to it over a localhost HTTP/WebSocket API.

The mental model in one read

The configuration is a **graph wired by name** — there is **no "channel" object** that bundles a source, filters, transforms, and destinations the way a legacy engine does. Instead you wire four building blocks, using the project's exact vocabulary:

- **Connection** — an endpoint that **receives** (`inbound()`) or **sends** (`outbound()`) messages: MLLP, TCP, File, REST, SOAP, Database, SFTP/FTP. Every message in or out is counted and logged.

- **Router** (`@router`) — a pure Python function bound to **one** inbound Connection. It sees every received message and returns the **name(s)** of the Handler(s) to forward to (or none, to filter).
- **Handler** (`@handler`) — a pure Python function that takes a message, **filters** → **transforms**, then returns `Send(...)` s naming one or more outbound Connections.
- **Message store** — durable persistence + the staged queue. SQLite (WAL) by default; PostgreSQL or SQL Server for production.

The edges between these are just names resolved at config load: an inbound names its Router, a Router returns Handler names, a Handler `Send` s to a named outbound. To understand a feed, follow the names; to change it, change a name. (See samples/config/IB_ACME_ADT.py and samples/config/adt.py for a complete, runnable example of this wiring.)

Under the hood, a received message flows through a **staged pipeline** of three persisted stages — **ingress** (the raw message, committed *before* the ACK) → **routed** (one row per Handler the Router selected) → **outbound** (one row per destination) — each drained by its own asyncio worker. Because nothing is ever silently dropped, every message carries a **disposition** that the store records as it flows: `RECEIVED` → `ROUTED` / `UNROUTED` → `PROCESSED` / `FILTERED` / `ERROR` . This is the count-and-log promise; operators read the disposition (and alerts), not the ACK, to confirm a message reached its destination. The full reliability model (at-least-once delivery, why routers/handlers must be pure) is in ADR 0001.

Who this guide is for

This guide serves two audiences: **operators** who install, run, and monitor the engine, and **config authors** (integration developers and analysts) who wire Connections, Routers, and Handlers. It links out to the reference docs rather than restating them. If you want a lighter, narrative onboarding first, read EARLY-ADOPTER-GUIDE.md (install-to-production rollout) or MENTAL-MODEL.md (the concepts above, in depth).

Where the depth lives — the reference map:

When you want…	Read
The full architecture, modularity standard, and dependency rules	ARCHITECTURE.md (diagrams: architecture-diagram.md)
Connection types, settings, and the graph (incl. `connections.toml`)	CONNECTIONS.md, ADR 0007
Service settings, environments, and `env()` values	CONFIGURATION.md
Running as a Windows service (NSSM)	SERVICE.md
Auth, RBAC, audit, and TLS	SECURITY.md, DEPLOYMENT.md
PHI handling and encryption-at-rest	PHI.md
The staged pipeline / reliability; payload-agnostic ingress; the read-only `db_lookup` ; X12	ADR 0001, ADR 0004, ADR 0010, ADR 0012
What's built vs. planned	FEATURE-MAP.md, README.md

A note on PHI before you run anything: this engine carries PHI, and a few commands (notably `dryrun` and `generate`) print **full message bodies** to stdout. Use **synthetic HL7 only** in examples, and never redirect that output to a committed file, ticket, or CI log. See PHI.md for the hard rules.

How the rest of this guide is laid out

The remaining sections are a **path**, in the order you'll actually work through them:

1. **Install** the engine and scaffold your config repo.
2. Send your **first message** end to end (e.g. samples/messages/adt_a01.hl7 via samples/send_mllp.py) against the bundled sample config.
3. **Author Connections** (in Python or `connections.toml`).
4. **Author Routers and Handlers** to route and transform.
5. **Operate** via the PySide6 console and the VS Code extension (messagefoundry/console/, ide/).
6. **Monitor and troubleshoot** with dispositions, alerts, dead-letter triage, and replay.

Work through them in order the first time; afterward, jump to the task you need.

Installing and running the engine

This section takes you from a clean machine to a running engine and an attached console. It covers the **developer / checkout** path (running the bundled `samples/config`) and points at the **pinned-wheel** consumer path where they differ.

> Two install styles exist. To **try MessageFoundry against the sample config** (this guide's running example), install from a checkout (`pip install -e .`). To **deploy your own interfaces**, install the pinned wheel and scaffold a config repo with `messagefoundry init` — the full consumer model is INSTALL-GUIDE.md.

1. Check prerequisites

- **Python 3.11+** (64-bit; 3.11, 3.12, 3.13, 3.14 supported). Everything else (the SQLite store, NSSM for the service) is auto-provisioned or bundled.
- **git** if you are cloning the checkout or standing up a config repo.
- **Administrator/elevation** only for the Windows service step (6).

Full hardware, OS, database, and port details: SYSTEM-REQUIREMENTS.md.

2. Install the engine

Create a virtual environment and install. For the **checkout / contributor** path (gives you `samples/config`, `samples/messages/`, and `samples/send_mllp.py`):

```
python -m venv .venv
.\.venv\Scripts\Activate.ps1                # Linux/macOS: . .venv/bin/activate
pip install -e .                            # core runtime + SQLite store
```

For a **deployment**, pin the published wheel instead (and verify its provenance — see INSTALL-GUIDE.md):

```
pip install "messagefoundry==0.2.0rc1"
```

Add only the extras a host actually needs (each is opt-in and lazy-imported):

```
pip install -e ".[console]"      # PySide6 admin console (step 5)
pip install -e ".[postgres]"     # PostgreSQL store backend
pip install -e ".[sqlserver]"    # SQL Server store backend — also needs the OS-level Microsoft ODE
pip install -e ".[sftp]"         # SFTP transport for the REMOTEFILE connector
```

(For a deployment wheel, the same extras apply: `pip install "messagefoundry[console]==0.2.0rc1"`, etc.) SQLite is the zero-dependency default — you need no extra to run the sample config.

3. Run the engine headless (dev)

From the repo root, against the bundled sample config:

```
python -m messagefoundry serve --config samples/config --db ./messagefoundry.db --env dev
```

- `--config` points at the directory of Connection/Router/Handler modules (here `samples/config/`, which includes IB_ACME_ADT.py and its transform adt.py).
- `--db` is the SQLite message store path (created on first run).
- **`--env` is required** — `serve` refuses to start without it. The active environment is a free-form **name** (`dev` / `staging` / `prod`, or a custom name) that does two things: it selects the value file `environments/<env>.toml` that `env("…")` lookups resolve against, and it sets the instance's **PHI posture** (`data_class` / `production`). Built-in names carry a default posture; a custom name must declare it. See CONFIGURATION.md.

When the engine runs from somewhere other than the repo root (e.g. under the service), anchor the value files with `--project-root <repo-root>` so `env()` values don't silently resolve empty — see INSTALL-GUIDE.md.

Network / auth posture. The API binds `127.0.0.1:8765` and **requires authentication** by default. A non-loopback bind without TLS is refused at startup; configure native TLS (or an upstream terminator) to expose it. Details: SECURITY.md and DEPLOYMENT.md.

Store encryption (PHI instances). On a PHI-carrying environment (`data_class = phi`), `serve` warns — and on a *production* PHI instance **refuses to start** — if no store encryption key is configured. Mint one with `messagefoundry gen-key` (set it as `MEFOR_STORE_ENCRYPTION_KEY`), or on Windows DPAPI-protect it to a file

with `messagefoundry protect-key --generate --out <file>` and point `[store].encryption_key_file` at it. The full key story is in PHI.md.

Confirm it's up:

```
curl http://127.0.0.1:8765/health
```

4. Scaffold your own config repo

When you're ready to author real interfaces, scaffold a standalone, `check`-green config repo instead of editing the samples:

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

It writes a runnable starter feed under `config/`, `environments/dev.toml` + `prod.toml`, a synthetic fixture under `messages/sets/`, a `messagefoundry.toml`, a `requirements.txt` pinning the engine, and a CI `check.yml`. Validate and run it:

```
pip install -r requirements.txt
messagefoundry check --config config --messages messages/sets
messagefoundry serve --config config --env dev
```

Making it a private git repo (and why secrets/PHI never land in it) is covered end-to-end in INSTALL-GUIDE.md.

5. Launch the admin console

The console is a **separate PySide6 process** that talks to the engine over the localhost API (it needs the `console` extra). With the engine running:

```
python -m messagefoundry.console --url http://127.0.0.1:8765
```

`--url` defaults to `http://127.0.0.1:8765`, so you can omit it when the engine is local on the default port. The console prompts for sign-in (authentication is on by default). Source: messagefoundry/console/.

6. Run as a Windows service (NSSM)

For production on Windows, run the engine as a background service via **NSSM** — it starts on boot, restarts on crash, captures stdout/stderr to rotating logs, and stops with Ctrl+C so connections drain cleanly. From an **elevated** PowerShell:

```
.\scripts\service\install-service.ps1 -Environment prod
```

`-Environment` is **required** (it becomes `serve --env`, just like step 3). The script is idempotent (re-run to reconfigure), auto-downloads a SHA-256-pinned NSSM if one isn't on `PATH`, and defaults to service name

`MessageFoundry`, config `<repo>\samples\config`, store + logs under `C:\ProgramData\MessageFoundry`, bind `127.0.0.1:8765`. Override paths/port/account with flags, e.g.:

```
.\scripts\service\install-service.ps1 -Environment prod -Port 9000 `
    -Config D:\hl7\config -DataDir D:\MessageFoundry `
    -ServiceAccount "NT SERVICE\MessageFoundry"     # least-privilege; auto-grants the needed ACLs
```

Manage and remove it:

```
nssm start  MessageFoundry
nssm status MessageFoundry
nssm stop   MessageFoundry
.\scripts\service\uninstall-service.ps1            # elevated; leaves logs + store in place
```

The complete procedure — least-privilege accounts, locking down the config/log directories, DPAPI key protection, update-vs-reinstall, and troubleshooting — is in SERVICE.md.

A note on PHI-emitting commands

`messagefoundry dryrun` and `messagefoundry generate` print **full message bodies** to stdout/stderr (`dryrun` only with `--show-phi`; redacted otherwise). Run them against **synthetic HL7 only** — never real PHI — and never redirect their output into a committed file, ticket, or CI log. See PHI.md.

Quickstart: send your first message

This walkthrough uses only the shipped samples in samples/config/ — no editing required. You'll start the engine, push a synthetic HL7 ADT message over MLLP, and watch it get received, routed, and archived to a file.

The sample inbound that does the work is `IB_Test_ADT` in samples/config/adt.py: an MLLP listener on **port 2575** whose Router forwards `ADT` messages to the `archive` Handler, which writes them to `./out/adt/{MSH-10}.hl7` via the `FILE-OUT_Test_ADT` outbound connection. (The config dir also wires other sample feeds — ACME ADT on 2600, X12, immunizations — but this Quickstart only exercises `IB_Test_ADT`.)

1. Start the engine

In one terminal, from the repo root, run the engine against the sample config. The active `--env` is **required** — use `dev` for local work:

```
python -m messagefoundry serve --config samples/config --db ./messagefoundry.db --env dev
```

This loads the config modules, opens (or creates) the SQLite store at `./messagefoundry.db`, serves the localhost API, and starts every sample listener. The startup log announces the active environment and

posture. Leave it running. The API binds `127.0.0.1` by default; first-run auth/console details are covered in the install guide — see INSTALL-GUIDE.md.

2. Send the sample message

In a **second** terminal (also from the repo root), send the shipped synthetic ADT^A01 over MLLP with the helper in samples/send_mllp.py:

```
python samples/send_mllp.py samples/messages/adt_a01.hl7
```

The script defaults to `--host 127.0.0.1 --port 2575`, which matches `IB_Test_ADT`, so no flags are needed. To be explicit (or to target a different listener), pass them:

```
python samples/send_mllp.py samples/messages/adt_a01.hl7 --host 127.0.0.1 --port 2575
```

The file it sends, samples/messages/adt_a01.hl7, is synthetic — never substitute real PHI here.

3. What to expect

`send_mllp.py` reuses the engine's MLLP framing, waits for the framed ACK, and prints it:

```
--- ACK ---
MSH|^~\&|...|...|...||ACK^A01|...|P|2.5.1
MSA|AA|MSG00001
...
```

An `MSA|AA` is a positive acknowledgement (`AA` = Application Accept). Under the staged pipeline the ACK means **received and durably persisted** (status `RECEIVED` at the ingress stage), *not* final delivery — see the ACK-on-receipt model in ARCHITECTURE.md. A moment later the message is routed and delivered: because `adt_a01.hl7` is an `ADT^A01`, the Router forwards it to the `archive` Handler, the Handler stamps the facility mnemonic and sends it on, and the file outbound writes it under `./out/adt/`. Its disposition advances `RECEIVED` → `ROUTED` → `PROCESSED`. (A non-`ADT` message would be logged `UNROUTED`; an `ADT` event the Handler drops would be `FILTERED` — nothing is silently discarded.)

Confirm the delivered file landed (its name is the message's MSH-10 control ID, `MSG00001`):

```
ls ./out/adt/
```

4. Where to see it land

Two complementary ways to inspect what happened:

- **The console Log Search page.** Launch the PySide6 admin console in a third terminal (`python -m messagefoundry.console`), then open the **Log Search** page to find the message, its disposition, the original raw body, and the per-destination delivery. The console talks only to the engine's API — see messagefoundry/console/.

- **A dryrun preview (no engine needed).** To see exactly how the config *would* route and transform a message without sending it anywhere, run:

```
python -m messagefoundry dryrun --config samples/config --messages samples/messages/adt_a01.hl7
```

Dryrun prints the resolved inbound, disposition, selected handlers, and would-send payloads. **Its output is PHI-bearing** — message bodies are redacted by default and only included with `--show-phi`. The samples here are synthetic, but never pipe `dryrun` (or `generate`) output to a committed file or a CI log.

Now change what it does

Once the round trip works, make it yours: edit the Router and Handler in samples/config/adt.py to change routing and transforms (see Authoring Routers and Handlers), add or retarget connections in samples/config/connections.toml — by hand or via the VS Code editor (ADR 0007) — and follow the connection naming convention in CONNECTIONS.md. Run `python -m messagefoundry check --config samples/config --messages samples/messages` before committing config changes.

Authoring Connections

A **Connection** is an endpoint that either *receives* messages (an **inbound** source) or *sends* them (an **outbound** destination). MLLP and File are the two most common transports, both shipped today (plus raw TCP, X12, REST, SOAP, Database, and SFTP/FTP — see the full catalog and per-setting reference in CONNECTIONS.md). Connections carry only *transport* config; routing/transform *logic* lives in code-first Routers and Handlers (see Authoring Routers and Handlers).

You author a connection one of two ways — **as code** (a `.py` module) or **as data** (`connections.toml`). Both desugar into the same registry, so they coexist freely.

Name your connection

Use the convention `[TYPE]_[PARTNER]_[MESSAGE]`:

- **TYPE** — transport + direction code: `IB` / `OB` (inbound/outbound MLLP), `FILE-IN` / `FILE-OUT`, `TCP-IN` / `TCP-OUT`, `X12-IN` / `X12-OUT`, etc. (the full table is in CONNECTIONS.md).
- **PARTNER** — the system on the other end (`ACME`, `Epic`, `Test`).
- **MESSAGE** — the HL7 message code (`ADT`, `ORU`, `VXU`, ...) or `MIXED` / `ALL`.

Example: `IB_ACME_ADT` = inbound MLLP from ACME carrying ADT. Names are plain strings, so hyphens and mixed case (`FILE-OUT_Test_ADT`) are fine. Router/Handler names are *not* connections and don't follow this formula.

Author a code-first inbound and outbound

In a config module, call the `inbound()` / `outbound()` factories with a transport spec (`MLLP()`, `File()`, ...). Here is the shape, drawn from samples/config/IB_ACME_ADT.py:

```
from messagefoundry import MLLP, env, inbound, outbound

inbound("IB_ACME_ADT", MLLP(port=2600), router="acme_adt_router")
outbound("OB_ACME_ADT", MLLP(host=env("acme_adt_host"), port=env("acme_adt_port", cast=int)))
```

Key points the sample demonstrates:

- **Inbound MLLP takes only a `port`** — passing `host` is a wiring error. The listen interface is the service-level `[inbound].bind_host` (loopback in DEV, a specific NIC in PROD), an operator setting, not authored here.
- **Outbound MLLP needs a `host` and `port`.** Anything that differs by environment (a downstream peer, a credential) uses `env("key")`, resolved per instance from `environments/<env>.toml` (and `MEFOR_VALUE_<KEY>` for secrets) — so one module runs unchanged in every environment. A referenced-but-undefined value fails loud at load, never a silent blank host.
- For a **File** endpoint, use `File(directory="./out/adt")` (in) / `File(directory=..., filename="{MSH-10}.hl7")` (out). For non-HL7 bodies, declare `inbound(..., content_type="x12")` so the body routes as a `RawMessage` — see samples/config/IB_PARTNER_X12.py.

The complete per-connector settings (TLS, retry, DoS guards, ACK mode, `simulate`, etc.) are documented in CONNECTIONS.md; each factory in messagefoundry/config/wiring.py **is the schema** for its transport.

Bind an inbound to its Router

An inbound names its Router with the `router=` keyword (`router="acme_adt_router"` above). The string must match a `@router` declared in some `.py` module loaded from the config dir; names resolve **globally** across the directory, so the inbound and its router can live in separate files. An inbound with no matching router fails `messagefoundry check`. (The router and handler are authored in code — covered in Authoring Routers and Handlers.)

Validate the wiring before running it:

```
messagefoundry check --config samples/config --messages samples/messages/adt_a01.hl7
```

Author a connection as data: `connections.toml`

A connection's transport config may instead live as **data** in an optional `connections.toml` next to the `.py` modules (ADR 0007). The loader merges these into the **same** registry the factories produce, so runtime, validation, and egress gating are identical. **Routing/transform logic stays code-first** — a data-authored inbound still binds a `router` declared in a `.py` module. Here is the shape, from samples/config/connections.toml:

```
[[inbound]]
name      = "IB_ACME_ADT_TCP"      # a second ACME intake, authored as data
transport = "mllp"
router    = "acme_adt_router"      # binds a router registered in a .py module
  [inbound.settings]
  port = 2700                      # inbound MLLP takes only a port
```

- The `transport` maps to the same factory (`mllp` → `MLLP()`), so a TOML connection produces a byte-identical spec and inherits every factory default and guard.
- **Secrets and per-environment peers use `{ env = "key" }`**, never an inline value (e.g. `host = { env = "acme_adt_host" }`, `port = { env = "acme_adt_port", cast = "int" }`). The file is repo-versionable and diffable.
- A name declared in **both** a `.py` module and `connections.toml` is a hard error (no silent shadowing).

Edit the file two ways, same file — by hand, or via the CLI (which is what the VS Code connection editor shells; it does a comment/format-preserving, validate-before-persist write):

```
messagefoundry connection list   --config samples/config
messagefoundry connection upsert --config samples/config --data '{...}'
messagefoundry connection remove --config samples/config --name IB_ACME_ADT_TCP
```

`upsert` / `remove` validate the whole config dir (structure + connector build + the fail-closed `[egress]` allowlist) before persisting and roll back on failure.

Try it end-to-end

Run the engine against the dev environment (the active `--env` is required), then send a synthetic message at the inbound's port:

```
python -m messagefoundry serve --config samples/config --db ./messagefoundry.db --env dev
python samples/send_mllp.py samples/messages/adt_a01.hl7
```

Use only synthetic HL7 (as in `samples/messages/`) — never real PHI on a test feed.

Authoring Routers and Handlers

A **Router** and a **Handler** are plain Python functions you write against the `messagefoundry` surface and register with the `@router` / `@handler` decorators. The Router sees *every* received message and decides which Handler(s) get it; each Handler filters, transforms the message, and returns `Send`s to outbound connections. Both are wired by name to a Connection — there is no enclosing "channel" object. The end-to-end template is samples/results_relay/results_relay.py; the simplest pair is samples/config/adt.py.

1. Write a Router (`@router`)

A Router takes the message and returns the **handler name(s)** to forward to — return `[]` to route nowhere (the message is still counted and logged `UNROUTED`, never dropped). It is the place to do fast, tolerant field peeks for routing decisions.

From samples/config/adt.py:

```
@router("adt_router")
def route(msg):
    if msg["MSH-9.1"] != "ADT":
        return []  # not ADT — routed nowhere (logged UNROUTED)
    return ["archive"]
```

The Router name (`"adt_router"`) is what an inbound Connection binds to: `inbound("IB_Test_ADT", MLLP(port=2575), router="adt_router")`. For the wiring conventions and per-connector settings, see CONNECTIONS.md.

2. Write a Handler (`@handler`)

A Handler receives the message from a Router, then **filters → transforms → returns `Send` (s)**. Return `None` to filter the message out (logged `FILTERED`); return one `Send` or a list to fan out to multiple outbound connections.

From samples/config/adt.py:

```
@handler("archive")
def archive(msg):
    if msg["MSH-9.2"] not in EVENT_LABELS:
        return None  # only admit/register/update events are archived (others FILTERED)
    mnemonic = FACILITY_MNEMONICS.get(msg["MSH-4"])
    if mnemonic:
        msg["MSH-4"] = mnemonic
    return Send("FILE-OUT_Test_ADT", msg)
```

The `Send` target (`"FILE-OUT_Test_ADT"`) names an `outbound(...)` Connection declared in the same config. To fan out, return a list — e.g. samples/results_relay/results_relay.py ends with `return [Send(OB_EHR, msg), Send(FILE_ARCHIVE, msg)]`.

3. The `Message` operations you'll use

Routers and Handlers work against the mutable HL7 `Message` in messagefoundry/parsing/message.py — never string-slice raw HL7; read/mutate through `Message` and re-encode. The methods you'll reach for (see the docstrings in that file for full signatures):

- **Peek a field** — `msg["PID-3"]` / `msg.field("OBX-3.1", occurrence=i)` ; convenience properties `msg.message_code` (MSH-9.1), `msg.trigger_event` (MSH-9.2), `msg.control_id` (MSH-10).

- **Iterate repetitions / segments** — `msg.repetitions("PID-3")` walks a `~`-list; `msg.count_segments("OBX")` plus `field(occurrence=…)` walks repeating segments.
- **Mutate** — `msg["MSH-4"] = value` / `msg.set(path, value, occurrence=…, repetition=…)`; rebuild a repeating block with `msg.delete_segments("OBX")` + `msg.add_segment(line, index=…)` + `msg.add_repetition(...)`.
- **Read MSH separators** — never hardcode `|^~\&`. The repeating-segment rebuild in samples/results_relay/results_relay.py reads them from MSH-1/MSH-2 (its `_separators` helper) before joining components.
- **Re-encode** — `msg.encode()` (or just pass `msg` to a `Send`, which encodes for you).

A non-HL7 inbound (`content_type` other than `hl7v2`) delivers a `RawMessage` instead — read `.raw` / `.text` / `.json()` and `Send` a built string. For cross-field business-rule checks beyond what schema validation catches, compose the primitives in `parsing/consistency.py`, as samples/consistency/validated_adt.py does (raise `ConsistencyError` → dead-letter, or `return None` → filter). The three validation tiers are laid out in HL7-VALIDATION.md.

4. Purity rule (don't break this)

Routers and Handlers **must be pure**: message in → message(s) out, no external side effects. At-least-once delivery re-runs a transform after a crash and relies on the re-run producing identical output. Side effects (network, file, DB writes) belong in outbound Connections, not in your functions.

The **one sanctioned exception** is a Handler making a **live, read-only** `db_lookup(connection, statement, params)` for enrichment/gating — its result may differ on a re-run, and that is accepted by design. It is read-only, gated by `[egress].allowed_db`, runs off the event loop, and is **unavailable on a Router or in dry-run** (it raises). See ADR 0010.

5. The authoring dev loop

Run these from your config-repo root as you write — they touch no network and start no server.

```
# Static check: structural problems in the config dir
python -m messagefoundry validate --config samples/config

# See the wired Connection → Router → Handler → Connection graph
python -m messagefoundry graph --config samples/config

# Run real messages through Routers/Handlers WITHOUT sending — shows the disposition,
# selected handlers, and would-send payloads. Bodies are redacted unless you pass --show-phi.
python -m messagefoundry dryrun --config samples/config --messages samples/messages/adt_a01.hl7

# The commit/CI gate: validate + dryrun (+ advisory ruff/mypy). Exit 0 only if every required check
python -m messagefoundry check --config samples/config --messages samples/messages
```

`dryrun --show-phi` prints **full message bodies** (raw + would-send payloads) to stdout — that is PHI. Run it on synthetic HL7 only, and never redirect its output to a committed file, a ticket, or a CI log (PHI is redacted by default for exactly this reason). Wire `check` into your pre-commit hook / CI so a broken Router or Handler can't merge. For depth on connection settings see CONNECTIONS.md; for the validation tiers see HL7-VALIDATION.md.

Operating with the console and the VS Code extension

MessageFoundry has two operator-facing UIs, and they do different jobs. The **Console** (PySide6) monitors and operates a *running* engine over the localhost API — start/stop connections, search the message log, watch health, manage users. The **VS Code extension** is for the *config author* — building and testing Connections/Routers/Handlers and promoting them to an engine. Neither touches the database directly; both go through the engine API.

Launching and signing in to the console

The console is a separate process that attaches to a running engine. Start the engine first (note the active `--env` is required):

```
python -m messagefoundry serve --config samples/config --db ./messagefoundry.db --env dev
```

Then launch the console (defaults to `http://127.0.0.1:8765`):

```
python -m messagefoundry.console
# point at a different engine: python -m messagefoundry.console --url http://127.0.0.1:8765
```

When the engine requires authentication (the default), a **Sign in** dialog appears first:

1. Enter your **username** and **password**, and pick a **Provider** — *Local* always; *Active Directory* appears only if the engine advertises AD.
2. If your account uses **two-factor (TOTP)**, you are prompted for the 6-digit code from your authenticator app (or a single-use recovery code) before the window opens.
3. On a forced password change, the console chains a change-password step (local accounts only — AD passwords are changed in Active Directory).

On success the session token is stored in your OS keyring (Windows Credential Manager), so later launches skip the prompt until it expires or is revoked. The account "⋯" menu in the header offers **Change password…**, **Two-factor authentication…** (enroll a TOTP authenticator — it shows a setup key + `otpauth://` URL for manual entry, then your one-time recovery codes — or turn it off), **Active sessions…** (inventory/revoke your own sessions), and **Sign out**. Which pages and actions you can use is governed by RBAC — see SECURITY.md for roles and per-route permissions.

A tour of the console pages

The left nav holds **Connections, Alerts, Dead Letters, Log Search, Engine Status**, and **Users** (the last only if you hold `users:manage`). A heart glyph in the lower-left reflects overall health (green healthy, orange low disk, red engine/DB stopped), and an auto-refresh interval (top-right) drives the active page.

- **Connections** — one row per endpoint (each inbound + each outbound). Select inbound (source) rows and use **Start / Stop / Restart**; select outbound (destination) rows and use the **Actions** ▾ menu to **Purge Top Message** or **Purge All Queued Messages** (a confirmed, irreversible drain that cancels queued deliveries — they will not retry). Each row's **Logs** link jumps to Log Search filtered to that connection. Columns show live counts (# Read / # Written / # Errored), queue depth, idle time, and delivery backlog.
- **Log Search** — browse the message store and inspect one message at a time. The detail panel has four tabs: **Parse tree** (the structured HL7 view, rendered client-side), **Raw** (exactly what arrived — preserved alongside the transformed form), **Deliveries** (per-destination status, attempts, next-attempt time, last error), and **Audit / events**. This is also where you **Replay** a message: select it and click **Replay** to re-run delivery for a failed/dead-lettered message. Viewing raw bodies is PHI access and is audited server-side with your username.
- **Engine Status** — read-only health: engine (version, uptime, PID, channels running/total, queue by status), database (path, size, free disk, journal mode, row counts), and, when clustered, the active-passive **Cluster** roster (mode, this node's role, leader, lease owner, per-node last-seen). **Run integrity check** runs `PRAGMA quick_check` on demand. If a Windows service is installed, a Service panel offers Start/Stop/Restart and (when absent) **Install service…** via NSSM/UAC — see SERVICE.md.
- **Users** (gated by `users:manage`) — RBAC administration: **Add user…**, **Set roles…**, **Set scope…** (per-channel scope), **Delete**, and **Revoke sessions**. Every operation is audited server-side. Role definitions live in SECURITY.md.
- **Alerts** — a read-only view of the loaded alert configuration: whether the webhook / email transports are wired, the global re-alert interval, and the ordered list of operator-authored alert rules (ADR 0014). Alert *delivery* still flows through the engine's AlertSink to those transports (see Monitoring dispositions and troubleshooting).
- **Dead Letters** — a dedicated page listing dead-lettered deliveries, each with its last error, plus **Replay selected…** and **Replay all…** actions (re-queuing is gated on the `messages:replay` permission and step-up re-auth, and audited server-side). A failed delivery also surfaces in **Log Search** — the **Deliveries** tab shows the per-destination error and a **Replay** button. For diagnosing and clearing stuck deliveries, see the troubleshooting guidance in EARLY-ADOPTER-GUIDE.md.

The VS Code extension (for config authors)

The extension is a thin TypeScript UI that shells out to the `messagefoundry` CLI; it authors and tests interfaces but does **not** run or monitor the engine (that's the Console). What it gives a config author:

- **Setup / scaffolding** — a **Home** launchpad with a **New Route Wizard** (Inbound → Router → Handler → Outbound generated as one module), **New Connection** (form → generates a

`[TYPE]_[PARTNER]_[MESSAGE]` module like `IB_ACME_ADT`), New Router/Handler, **Generate Samples** (writes a synthetic, conformant corpus via `messagefoundry generate` — no PHI), and **Set Up Version Control & Checks** (puts the project under git with a `messagefoundry check` pre-commit hook).

- **Validate + graph** — *Validate on save* surfaces problems in the Problems panel; the **Connections** sidebar renders the wired graph (`messagefoundry graph`) by convention name, with **Filter** and **Group** controls and a row ⚙ **gear** to open a connection's `MLLP()` / `File()` settings in code.
- **Test Bench** — load `.hl7` files (each may hold many messages, split on `MSH`), **dry-run** them through the config without sending, and see each message's disposition, with a **Before/After** diff and a **Debug** step-through under `debugpy`. The load dialog opens to `messagefoundry.messageSetsDir` (default `samples/messages`).
- **Stage → Promote** — apply local config to a *running* engine: validate, pick a target environment, pre-flight a dry-run `POST /config/reload {dry_run:true}` against that target's `env()` values, confirm, then atomically swap the live graph. The engine requires auth, so the extension signs you in (token cached in VS Code SecretStorage).
- **AI assist** — an `@messagefoundry` chat participant (`/explain`, `/transform`, `/router`, `/review`, `/migrate`, `/test`) that is provider-agnostic and **only ever sends code + the config graph, never message bodies / PHI**.

Full feature and settings reference: ide/README.md.

> PHI note: `messagefoundry generate` and `dryrun` (which the Test Bench uses) print full message bodies to stdout/stderr — run them only against synthetic HL7 (e.g. samples/messages/adt_a01.hl7), and never redirect their output to a committed file, ticket, or CI log.

Monitoring dispositions and troubleshooting

MessageFoundry never accepts-and-drops a message: **every message a connection receives is persisted and counted before it is ACKed**, and its *disposition* is recorded as it flows through the pipeline. This section is how you watch those dispositions, recover failed deliveries, and get paged when a lane stalls. For the underlying guarantees see ARCHITECTURE.md and the staged-pipeline rationale in ADR 0001.

What each disposition means

A message's status moves through the staged pipeline (`ingress -> routed -> outbound`). The store's finalizer is the **single authority** that sets the final disposition — it only finalizes once every handler's work resolves, so one delivered handler can't mark a message done while a sibling is still in flight:

Status	What it means to you
`RECEIVED`	Persisted at ingress and ACKed. The count is booked; routing hasn't run yet.
`ROUTED`	The Router selected at least one Handler; the message is awaiting transform/delivery.
`UNROUTED`	The Router ran but chose **no** Handler. Not an error — logged, kept, not delivered. Check the Router logic if you expected a destination.
`FILTERED`	Every Handler ran but delivered nothing (filtered out). Expected for drop-by-design rules; surprising drops mean a Handler filter is too aggressive.
`PROCESSED`	Every selected Handler transformed and **all** destinations delivered. The happy path.
`ERROR` / dead-letter	A stage failed. A decode/parse/strict-validate failure is recorded *before* ingress (and NAK'd — see below); a routing/transform/delivery failure is recorded *after* the ACK as `ERROR` plus an AlertSink event. A delivery that exhausts retries becomes a **dead-letter** you can inspect and replay.

The key operator shift under the staged pipeline: **an `AA` ACK means "received and persisted," not "delivered."** A post-ingress failure is a disposition + alert, not a NAK.

Where to watch dispositions

- **Console -> Log Search.** The message browser (console/search.py) has a **status** filter — type a disposition (e.g. `unrouted`, `error`) to narrow the list, then open a message to see its raw body, parse tree, deliveries, and audit trail. The Connections page has a per-connection **Logs** link that opens Log Search pre-filtered to that channel.
- **Console -> Connections.** The dashboard (console/connections.py) shows each connection's live status plus per-connection counts, including an **errored** column, so a climbing error count on one feed is visible at a glance.
- **API.** `GET /messages?status=error` (and `&channel_id=`, `&message_type=`) is the filter the console uses; `GET /stats` returns outbox-by-status + in-pipeline depth; `GET /status` returns engine uptime, running/stopped channel counts, and DB size/free-disk. All require the `monitoring:read` (stats/status) or `messages:read` (Log Search) permission — see SECURITY.md.

The ERROR / dead-letter path: inspect and replay

A delivery dead-letters when its retries are exhausted. Retry behavior is per-outbound (defaults in `[delivery]` — see CONFIGURATION.md):

- `retry_max_attempts` **unset = retry forever** (the conservative default; under FIFO the failing head blocks its lane until it succeeds or is purged). Set a finite value to opt into retry-then-dead-letter.
- A partner `AR` **reject fails fast** (no retry); an `AE` **NAK / transient transport failure is retried** with backoff.

To recover:

1. **Find the dead-letters.** Console: open the message in Log Search and read its delivery row's **Last error**. API: `GET /dead-letters` (optionally `?channel_id=&destination_name=`) lists dead deliveries newest-first;

each row carries `last_error`.

2. **Fix the cause** (the downstream endpoint, the transform, the config).
3. **Replay.** `POST /dead-letters/replay` re-queues the dead deliveries (optionally scoped by `channel_id` / `destination_name`); each affected message reverts from `error` to `received` and re-drains. Already-delivered rows are left alone. Replay requires the `messages:replay` permission and is **step-up (re-auth) gated**, and may be held for a second approver when `[approvals]` is configured. (The console's **Dead Letters** page drives this same replay from the UI — **Replay selected…** / **Replay all…**.)

> Replaying re-transmits real message bodies — it is audited per acting user. Treat it like any PHI action (PHI.md).

Alerting: fire-and-forward notifications

The engine raises operational alert events — **`connection_stopped`** (a lane halted by the `stop` internal-error policy), **`queue_buildup`** (a backlog past its depth/age threshold), **`storage_threshold`** (the store grew past `[retention].max_db_mb`), and **`cert_expiry`** (a monitored TLS certificate nearing expiry) — through an AlertSink. With no `[alerts]` transport configured these are just logged at `WARNING`; configure a transport and they fan out to it (alert_sinks.py).

Configure alerts in the `[alerts]` section (CONFIGURATION.md):

```
[alerts]
webhook_url = "https://hooks.example.com/mf"   # POSTs each event as JSON (Slack/Teams/PagerDuty)
email_smtp_host = "smtp.example.com"
email_from = "messagefoundry@example.com"
email_to = ["oncall@example.com"]
# Optional per-event routing/severity/suppression rules (first match wins) — ADR 0014:
[[alerts.rules]]
event_type = "connection_stopped"
severity = "critical"
transports = ["webhook"]
```

The SMTP password is a secret — supply it via `MEFOR_ALERTS_EMAIL_PASSWORD`, never the file. Per-event severity, transport routing, thresholds, suppression, and cooldown are tuned with ordered `[[alerts.rules]]` tables (ADR 0014); an event matching no rule notifies every configured transport at `warning`.

Important: fired alerts are **fire-and-forward notifications**, not a queryable event history — once configured, you rely on your webhook/email target to see them as they happen. (The console's **Alerts** page and `GET /alerts/rules` show the *configured* transports and rules, read-only — not a log of past alerts.) Payloads carry only the connection name and queue shape, never message content (no PHI).

Common problems

- **Sender got a NAK (AE/AR).** A decode/parse/strict-validate failure rejects *synchronously* at the listener and records `ERROR` **before** any ingress row — the message never entered the pipeline. Fix the inbound HL7 (or relax `validation.strict` on that connection). Treat the message body as untrusted data, not a malformed instruction.
- **Sender got AA but nothing was delivered.** Expected under ACK-on-receipt: routing/transform/delivery failures happen *after* the ACK. Look at the message's disposition (`UNROUTED` / `FILTERED` / `ERROR`) and the AlertSink — **not** the ACK — for the outcome.
- **A lane stopped processing.** A `connection_stopped` alert means an outbound's worker halted on an internal/code error (`internal_error = stop`). The messages are preserved for replay; fix the cause, then reload/restart the connection.
- **Backlog growing.** A `queue_buildup` alert usually means a retry-forever head is blocking its FIFO lane, or the downstream is down. Check the destination, then inspect/purge or replay the blocking row.
- **Console can't reach the engine.** The API binds `127.0.0.1:8765` by default and requires auth; confirm the engine is serving (`python -m messagefoundry serve --config samples/config --db ./messagefoundry.db --env dev`) and that the console points at the right host/port.
- **Low disk / store growing.** `GET /status` reports DB size and free disk; a `storage_threshold` alert fires past `[retention].max_db_mb`. Tune retention in `[retention]` (CONFIGURATION.md) — purges null PHI bodies while keeping the metadata/disposition rows, so counts and audit stay intact.

Where to go next

- **Concepts in depth** — MENTAL-MODEL.md, ARCHITECTURE.md (diagrams: architecture-diagram.md)
- **Narrative onboarding, install to production** — EARLY-ADOPTER-GUIDE.md, INSTALL-GUIDE.md, SYSTEM-REQUIREMENTS.md
- **Connections reference** — CONNECTIONS.md, ADR 0007 (`connections.toml`)
- **Service settings & environments** — CONFIGURATION.md
- **Validation tiers** — HL7-VALIDATION.md
- **Run as a service** — SERVICE.md
- **Security, RBAC, TLS** — SECURITY.md, DEPLOYMENT.md
- **PHI handling & encryption-at-rest** — PHI.md
- **VS Code extension** — ide/README.md
- **What's built vs. planned** — FEATURE-MAP.md, README.md
- **Design records** — ADR 0001 (staged pipeline), ADR 0004 (payload-agnostic ingress), ADR 0010 (`db_lookup`), ADR 0012 (X12), ADR 0014 (alerting rules)