Migrating from a legacy engine to MessageFoundry

Moving live clinical interfaces off an established engine is a careful, reversible exercise — not a big-bang rewrite. This guide is written for the interface analyst and integration developer who already runs feeds on Corepoint, Mirth, Cloverleaf, Rhapsody, or Ensemble and wants a clear, low-risk path to MessageFoundry.

It walks through the four phases of a real migration: **inventory** what you have, **map** the incumbent's concepts onto MessageFoundry's building blocks, **test and validate** in parallel against real-shaped traffic, and **cut over a feed at a time with a one-step rollback**.

> Corepoint, Mirth Connect, Cloverleaf, Rhapsody, and Ensemble are trademarks of their respective owners. MessageFoundry is an independent project and is not affiliated with, sponsored by, or endorsed by any of those vendors. Comparisons here are intended to be factual and fair; the goal is to help you translate concepts you already know, not to disparage the tool you are leaving.

Why teams migrate

Most teams we talk to are not unhappy with what their incumbent engine *does* — they are unhappy with how it is *owned*. The integration logic lives inside the engine's database or its embedded scripting language, which makes it hard to diff, review, or reproduce across environments.

MessageFoundry takes a different posture: **your configuration is code you own and version-control.** Connections, routing, and transforms live in an org-owned git repository, not in the engine's database. Every change is a reviewable diff with an author and full history; rolling back is a `git` operation; and the same modules run unchanged in dev, staging, and production, with only per-environment hosts differing. The engine itself is installed as a pinned, signed dependency and left alone — your team works only in the config repo.

You can author that configuration two ways: visually, with guided wizards and a connection editor that validate as you save, **or** in plain Python when you want full control. Python is the power tool, not the price of entry.

Before you start: the mental-model shift

The single most important thing to internalize is this: **MessageFoundry has no "channel" object.**

Where an incumbent engine gives you one *channel* that bundles a source, a filter, transformers, and destinations into a single unit, MessageFoundry models the same flow as a **graph wired by name** — a set of small, independent nodes joined by named edges, like boxes connected by arrows in a flowchart. There are four building blocks:

Building block	What it is
Connection	An endpoint that *receives* (inbound) or *sends* (outbound) messages — MLLP, file, TCP, X12, REST, SOAP, database, FHIR. Every message in or out is counted and logged.
Router	A small pure function bound to one inbound. It sees every received message and returns the name(s) of the Handler(s) to forward to. It may filter by returning nothing.
Handler	A small pure function that takes a message from a Router, filters and transforms it, and returns one or more `Send`s to outbound Connections.
Message store	The durable queue and persistence layer — SQLite by default, or PostgreSQL / SQL Server for production.

The edges are just strings resolved when the configuration loads: an inbound names its Router, a Router returns its Handlers' names, and a Handler `Send`s to a named outbound. Nothing holds a direct reference to anything else.

This is why a migration is mostly a **decomposition exercise**. A single incumbent channel usually becomes one inbound Connection, one Router, one or more Handlers, and one or more outbound Connections — and the shared pieces (a common destination, a reused transform) get defined **once** and referenced by name, instead of being copied into every channel that needs them.

Phase 1 — Inventory your existing interfaces

Before touching MessageFoundry, build a complete picture of what you run today. For each interface, capture:

- **Source** — the transport and direction (MLLP/TCP listener, file drop, database poll, HTTP/SOAP endpoint), the listen port or path, and the upstream partner/system.
- **Destinations** — every downstream the channel delivers to, with transport, host/port, and any per-destination filtering.
- **Message types** — the HL7 v2 trigger events (ADT, ORM, ORU, SIU, DFT, MDM, VXU, …), or X12 / FHIR / other payloads.
- **Routing logic** — the rules that decide which messages go where (often a filter on `MSH-9`, a facility, or a sending application).
- **Transforms** — field mappings, value-set translations (code sets), segment add/remove, and any enrichment from a lookup.
- **Acknowledgement behavior** — original vs. enhanced ACK, and whether the partner expects application-level NAK semantics.
- **Security** — TLS/mTLS on the wire, credentials, and any IP allowlisting.
- **Volume and timing** — peak throughput, whether the feed is steady or intermittent, and any ordering requirements.

A useful exercise while you do this: note, for each incumbent channel, how many *distinct* destinations and transforms it actually contains. Channels that fan out to several downstreams, or that share a transform with other channels, are the ones that benefit most from MessageFoundry's reuse-by-name model — you will collapse a lot of duplicated configuration.

Phase 2 — Map the concepts

The translation is mechanical once you know the vocabulary. This table maps common incumbent terms onto MessageFoundry building blocks. Incumbent terminology varies by product and version, so treat the left column as a guide, not a contract.

Incumbent concept	MessageFoundry equivalent
Channel (Mirth) / Interface	A *wired path*: one inbound Connection → Router → Handler(s) → outbound Connection(s). There is no single bundled object — you assemble it from the four blocks.
Source connector (Mirth)	An **inbound Connection** (`inbound(...)`).
Destination connector (Mirth)	An **outbound Connection** (`outbound(...)`).
Source filter / channel filter	A **Router** — returns the Handler name(s) to forward to, or nothing to drop the message (recorded as `UNROUTED`, never silently lost).
Transformer / destination transformer	A **Handler** — filters, transforms, and `Send`s.
Destination-set routing / "send to these destinations"	A Router that fans out (`return ["to_a", "to_b"]`) or a Handler that fans out (multiple `Send`s).
"E Process" (Corepoint)	A **Router** — typically grouped in a `routers_<area>.py` file, each listing its handler(s).
"E Child" (Corepoint)	A **Handler** — a shared transform defined once in `handlers_<partner>.py` and named by multiple routers.
Channels stored in the engine database	**Configuration in a git repo.** MessageFoundry deliberately keeps logic out of the data store — the database holds only messages and runtime state, never configuration.
Embedded JavaScript / proprietary scripting	Ordinary **Python** you own, plus a built-in structured HL7 transform model (read/set by field path, iterate repetitions, add/remove segments, MSH-aware re-encode).

A worked example

Suppose your incumbent has a channel "ACME ADT → Epic": it listens for ADT over MLLP, drops anything that is not an ADT, and forwards to your EHR. In MessageFoundry that is one small module:

```
from messagefoundry import MLLP, Send, env, handler, inbound, outbound, router

inbound("IB_ACME_ADT", MLLP(port=2600), router="acme_adt_router")
outbound("OB_EPIC_ADT", MLLP(host=env("epic_host"), port=env("epic_port", cast=int)))

@router("acme_adt_router")
def route(msg):
    return ["acme_adt_handler"] if msg["MSH-9.1"] == "ADT" else []   # non-ADT → UNROUTED

@handler("acme_adt_handler")
def handle(msg):
    # filter / transform here
    return Send("OB_EPIC_ADT", msg)
```

Note `env(...)`: the downstream host and port differ per environment, so they are resolved from `environments/<env>.toml` at load — the same module runs unchanged in dev, staging, and prod. The connection naming convention is `[TYPE]_[PARTNER]_[MESSAGE]` (for example `IB_ACME_ADT` inbound, `OB_EPIC_ADT` outbound), which keeps a large interface inventory legible.

Connections can be data, not just code

If you prefer to treat endpoints the way you treat channels today — as editable settings rather than code — a connection's transport configuration (type, settings, the inbound's router binding, delivery knobs) can live in a `connections.toml` file, edited by hand or through the VS Code connection editor. The *routing and transform logic* stays in Python. Either way, the loader produces the same registry the engine runs, with the same validation. Secrets are never written inline — they are referenced from environment variables.

Mapping the transport for each connector

MessageFoundry ships connectors that cover the common incumbent connector types:

Incumbent connector	MessageFoundry connector
MLLP / LLP Listener and Sender	`MLLP(...)` — inbound listener and outbound sender, with optional TLS and mutual-TLS.
TCP Listener / Sender (custom framing)	`Tcp(...)` — raw TCP with configurable delimiter framing, for X12 or other non-HL7 feeds carried opaquely.
X12 over TCP	`X12(...)` — frames by the interchange itself (ISA…IEA), with optional synchronous request/response (e.g. 270 → 271 real-time eligibility) and TA1 classification on a capturing outbound.
File Reader / Writer	`File(...)` — polls or writes a directory, with content-sniffed HL7 ingest, size caps, quarantine of malformed files, and atomic writes.
HTTP Sender	`Rest(...)` — outbound HTTP(S) client.
Web Service Sender	`Soap(...)` — outbound SOAP, including a WS-* mode with mutual TLS, WS-Security, and WS-Addressing.
Database Reader / Writer	`DatabasePoll(...)` (inbound poll) and `Database(...)` (outbound write) — parameterized SQL against SQL Server.
FHIR client	`FHIR(...)` — outbound FHIR REST client (create/update/transaction/batch) with conditional create/update for idempotency.

A note on transports that are still on the roadmap (for example some inbound HTTP/SOAP source facades and additional remote-file schemes): if a feed depends on one of these, plan to migrate it in a later wave, or front it with one of the shipped sources during the interim.

Two reliability facts to design around

Two engine behaviors shape how you write transforms and how downstreams should behave:

- **At-least-once delivery.** MessageFoundry never silently loses a message — every received message is persisted before it is acknowledged. The trade-off is that, in a narrow crash-after-send window, the engine may re-deliver a single in-flight message. In normal operation every message is delivered exactly once. Because transforms are pure, a re-delivered HL7 message keeps the **same `MSH-10` control ID**, so a downstream keyed on `MSH-10` sees a retry of a known message, not a new clinical event. Design outbound receivers to be idempotent (a natural upsert, an idempotency key, or a message-id de-dup). FHIR's conditional create/update knobs and X12's TA1 handling exist precisely for this.
- **Routers and transforms must be pure.** A Router or Handler is message-in, message-out, with no external side effects, so a safe re-run produces identical output. The one sanctioned exception is a read-only database lookup for enrichment or gating. If your incumbent transforms reach out to external systems mid-transform, plan to move that work into a destination or a read-only lookup.

Phase 3 — Test and validate in parallel

The strength of a code-first configuration is that you can validate it long before any feed is cut over.

Validate the configuration itself

Every change is gated by `messagefoundry check` — the same command your commit hooks and CI run. It validates the wiring (an unknown router, a dangling handler, a duplicate name or port is a loud error, not a silent surprise), dry-runs sample messages, and runs advisory lint. Because the wiring is just names, mistakes surface at load time rather than in production.

Dry-run transforms with before/after diffs

The VS Code extension includes a **Test Bench** that dry-runs `.hl7` files through your Routers and Handlers and shows before/after diffs, so an analyst can confirm a transform does exactly what the incumbent did, file by file, using synthetic messages. Pair this with synthetic message generation so you are never testing against real PHI.

Probe connectivity without sending a message

Before a feed goes live, `POST /connections/{name}/test` runs a **reachability probe**: it builds a fresh connector (never the live one), honors the egress allowlist, and confirms it can reach the peer — a socket connect for MLLP/TCP/X12, a `SELECT 1` for a database, an HTTP `HEAD` for REST/SOAP, a writability check for a file directory — **without sending any real message.** It is audited. This lets you confirm firewall rules, credentials, and TLS to every downstream ahead of cutover.

Run a true parallel comparison with the tee relay

For the highest-confidence validation, run MessageFoundry **alongside** your live incumbent on real-shaped traffic before cutting over anything. The **tee relay** is a small, standalone tool that sits in front of both engines:

- Repoint the upstream source (for example an EHR's outbound) at the relay. The relay **acknowledges on receipt**, then forwards the **unchanged** bytes to **both** your live incumbent (production, unchanged) and a shadow MessageFoundry instance.
- Optionally, add a duplicate outbound send in your incumbent's configuration that mirrors its outbound messages to a second relay listener, so the shadow MessageFoundry can also see the incumbent's *output* for side-by-side comparison.

Run the shadow MessageFoundry's outbound connections in **simulate (shadow) mode**: it exercises the full route-and-transform pipeline and records what it *would* have sent, but suppresses real egress — so it can process live-shaped traffic without double-delivering to downstreams. You then compare MessageFoundry's transformed output against your incumbent's for parity, and you do it before a single real feed has moved.

A few things to understand about the relay so you use it correctly:

- **It is a validation tool for test and synthetic data only.** It is not hardened to carry production PHI, prints a warning at every start, and should run only on a trusted test segment. Use it to gain confidence,

not as a permanent production component.

- It is a **fail-closed relay, not durable store-and-forward.** If the live (production) path becomes unreachable, the relay stops accepting new connections so the upstream sees the outage and queues on its side; you restart it once the production path is healthy.
- The shadow leg is decoupled, so a slow or down shadow MessageFoundry never back-pressures the production path.
- It records one metadata row per forwarding leg (outcome, ACK code, `MSH-10`, message type, size, reason) and can export that log as JSON metadata — no message bodies — for review.

Phase 4 — Stage the cutover, keep rollback one step away

Migrate **one feed at a time**, not the whole engine at once. A typical per-feed cutover:

1. **Author and review** the feed's module(s) in your config repo as an ordinary pull request, gated by `messagefoundry check`.
2. **Validate** it with the Test Bench (transform parity) and the connectivity probe (every downstream reachable).
3. **Shadow it** through the tee relay in simulate mode and compare output to the incumbent until you are satisfied.
4. **Cut over** by repointing the upstream source at the MessageFoundry inbound and enabling real (non-simulated) egress for that feed's outbounds. Keep the incumbent's channel configured but idle.
5. **Watch** the feed in the monitoring console — per-message disposition, the delivery and audit trail, alerts, and the dead-letter queue with replay — for a soak period.
6. **Roll back** if anything looks wrong: repoint the source back at the incumbent (or, during the shadow phase, simply stop the relay). Because the incumbent channel was left in place and your MessageFoundry change is a reviewed diff, rollback is fast and clean.

Because configuration lives in git and the database holds only state, a feed's logic and its in-flight messages have independent blast radius: promoting new config never touches stored messages, and restoring the store never changes routing. Disaster recovery is two independent operations — redeploy config from git, restore data from a database backup.

Sequencing the waves

A pragmatic order:

1. **Start with low-risk, well-understood feeds** — a one-to-one ADT pass-through, or a file-based feed — to exercise your whole pipeline (author → check → test → shadow → cut over → monitor) end to end.
2. **Then consolidate the duplicated ones.** Feeds that share a transform or a destination across several incumbent channels are where the reuse-by-name model pays off — define the shared Handler or outbound once and reference it.
3. **Save the complex and the roadmap-dependent ones for last** — synchronous request/response feeds, WS-* mutual-TLS submissions, and anything that depends on a connector still on the roadmap.

Deployment guidance to plan for

A few operational points worth deciding before go-live, rather than discovering at cutover:

- **Bind interfaces deliberately.** Inbound MLLP/TCP listeners bind to a configured interface (loopback by default). Exposing a listener off loopback is a per-environment operator decision, and a non-loopback listener carrying PHI should use TLS; a per-connection source IP allowlist can restrict which peers may connect. Treat this as deployment configuration, not a limitation.
- **Egress is allowlisted, fail-closed.** Outbound hosts are gated by an allowlist — populate it for every downstream a migrated feed delivers to, or delivery is refused.
- **Pick your store for the target scale.** SQLite (zero-setup, single file) is the bundled default and suits pilots and single-node deployments; PostgreSQL or SQL Server is the production choice and is required for active-passive high availability. Measure your own feeds with the load harness before committing to a tier; published throughput figures are engineering estimates, not guarantees.
- **High availability is active-passive.** Run identical engine processes against one shared server database; exactly one leader runs the graph, with warm standbys that take over on failure. Front it with a floating VIP whose health check is a TCP connect to the listener port. Failover is minutes-class, not zero-downtime — another reason downstreams should be idempotent.

Security posture during and after migration

MessageFoundry carries PHI, so security is built in rather than bolted on, and a migration is a good moment to tighten it relative to the incumbent:

- **Authentication and RBAC** at a single API choke point, with deny-by-default per-route permissions and full audit of every PHI access (raw view, replay) tied to the acting user.
- **Multi-factor authentication** is built in and required for local accounts; enterprise/AD users get MFA through their own identity provider via SSO federation.
- **Interface authentication** options include mutual TLS, OAuth 2.0 client-credentials, and SMART-on-FHIR Backend Services for FHIR endpoints.
- **Encryption at rest** — message bodies are encrypted in the store.
- **On-premises by default** — the engine API binds to loopback and requires authentication; no PHI leaves the local environment without explicit, reviewed configuration.

MessageFoundry's security work includes an internal self-assessment against OWASP ASVS 5.0 Level 3 (212 requirements met, 0 failed, 0 partial, 133 not applicable, of 345). This is a **self-assessment, not an external audit** — an independent code review and penetration test are recommended at Level 3, are not required, and are planned. The architecture **supports a HIPAA-compliant deployment**; compliance ultimately depends on how you deploy and operate it.

A short pre-flight checklist

- [ ] Every incumbent channel inventoried: source, destinations, message types, routing, transforms, ACK behavior, security, volume.
- [ ] Each channel decomposed into inbound Connection(s), Router(s), Handler(s), and outbound Connection(s).
- [ ] Shared transforms and destinations identified and defined once, referenced by name.
- [ ] Config repo scaffolded; `environments/<env>.toml` set per environment; secrets supplied via environment variables, never committed.
- [ ] `messagefoundry check` green; transforms confirmed with the Test Bench against synthetic messages.
- [ ] Connectivity probed to every downstream; egress allowlist populated; TLS configured for any off-loopback listener.
- [ ] Parallel run completed through the tee relay in simulate mode, output compared to the incumbent (test data only).
- [ ] Per-feed cutover plan with the incumbent channel left idle for fast rollback.
- [ ] Monitoring, alerts, and dead-letter replay verified in the console; soak period defined.

Further reading

- MessageFoundry connectors and settings: https://github.com/MEFORORG/MessageFoundry/blob/main/docs/CONNECTIONS.md
- The mental model — the four building blocks and how a message flows: https://github.com/MEFORORG/MessageFoundry/blob/main/docs/MENTAL-MODEL.md
- The tee relay for parallel-run validation: https://github.com/MEFORORG/MessageFoundry/blob/main/docs/TEE-RELAY.md
- Security model and PHI handling: https://github.com/MEFORORG/MessageFoundry/blob/main/docs/SECURITY.md