MessageFoundry — A Mental Model

Open-source healthcare integration engine · Python · v0.2.0rc1 (Early Access)

This document is an orientation map, not a manual. It builds the **mental model** you need to reason about MessageFoundry: what it is, the four building blocks, how a message flows, the invariants you must not break, and where everything lives. Read it top to bottom once and the rest of the documentation — and the engine's behavior — will make sense.

1. What it is, in one breath

MessageFoundry is an **open-source, Python** integration engine for healthcare — a modern alternative to legacy interface engines like Mirth and Corepoint. It receives, routes, transforms, and validates clinical/business messages between systems. It handles **HL7 v2.x by default** and is payload-agnostic for other formats (JSON, XML/SOAP, X12 EDI, DB records).

What sets it apart: **you can set it up visually** — guided wizards scaffold connections and routes, validate your config as you save, and let you test against sample messages — **or write the routing and handling in plain Python** when you want full control. Either way the logic is ordinary Python you own and version-control — not a legacy engine's embedded scripting language, and not a locked-in low/no-code GUI. Connection setup in particular can be pure data (a TOML file edited by hand or in a VS Code GUI). *Python is the power tool, not the price of entry.*

> **The pitch in one line:** *The best of the legacy interface engines — their proven reliability, deep connector catalogs, and battle-tested handling of HL7 v2 plus JSON, X12, and other formats — with none of the lock-in: configuration you own and version-control (set up with guided wizards or in Python), a durable, broker-free queue (SQLite by default, or Postgres/SQL Server), and auth, RBAC, audit, and encryption-at-rest built in rather than bolted on.*

Stack: python-hl7 (tolerant parsing) + hl7apy (strict validation), FastAPI/uvicorn (localhost engine API), SQLite/aiosqlite (message store; Postgres & SQL Server also supported), PySide6 (admin console). Python 3.11+, asyncio core.

2. The core model: a graph of four building blocks

There is **no "channel" object**. Where a legacy engine gives you a single "channel" that bundles a source, filters, transformers, and destinations, MessageFoundry has **no such bundling element**. The configuration is a **graph wired by name** — a set of nodes connected by edges, like boxes joined by arrows in a flowchart. The nodes are the Connections, Routers, and Handlers; the edges are the by-name links between them: inbound Connections name a Router; Routers name Handlers; Handlers send to outbound

Connections. (“Channel” and “route” are fine as casual prose for a wired path — there’s just no built element that constructs one.)

Connection	An endpoint that receives (inbound) or sends (outbound) messages — MLLP, file, TCP, HTTP/REST, SOAP, database, SFTP/FTP. Every message in or out is counted and logged.	inbound() / outbound() factory, or connections.toml	transports/
Router	A pure Python function bound to ONE inbound. Sees every received message; returns the name(s) of Handler(s) to forward to. May filter (return none).	@router function	config modules
Handler	A pure Python function taking a message from a Router: filter → transform → return Send(s) to one or more outbound Connections.	@handler function returning Send	config modules
Message store	Durable persistence + the staged queue for received/processed/errored messages. SQLite (WAL) by default; Postgres & SQL Server for production.	*(infrastructure)*	store/

The wiring, drawn:

```
inbound Connection ──names──▸ Router ──names──▸ Handler ──Send──▸ outbound Connection
   (MLLP :2600)               (@router)        (@handler)         (MLLP host:port)


         every hop is by NAME; the whole config is a flat, by-name graph —
         not a nested 'channel' that owns its pieces.
```

Why “wired by name,” and why it’s a graph

“Wired by name” means the links between pieces are just **strings, resolved when the config loads**:

- An inbound names its Router (router="acme_adt_router").
- A Router returns its Handlers’ names (["acme_adt_handler"]).
- A Handler returns Send("OB_ACME_ADT", msg) naming an outbound.

No piece holds a direct reference to another — the loader looks each name up and assembles them into a Registry the engine runs. The shape that falls out is a **graph**: nodes (Connections, Routers, Handlers) joined by named edges, with fan-out (one Router → many Handlers) and fan-in (many Handlers → one outbound) for free.

That’s not just bookkeeping — the by-name graph is what makes this project’s goals reachable:

- **Reuse instead of duplication.** Define a destination, a shared transform, or a router once and reference it from anywhere. A bundled “channel” would force you to copy a shared transform or destination into every channel that needs it (the exact pain ADR 0007 calls out); a name graph just adds an edge.

- **Each node is small, independent, and testable.** A Router or Handler is a tiny pure function with a contract — message in → names/Sends out. You can write, test, and reason about one without loading the rest, which is what lets people (and AI agents) build different parts in parallel without colliding (the modularity standard, §10).
- **Names let code and data mix.** Because an edge is a name resolved at load, a Connection can come from Python or be desugared from connections.toml through the same factories into the same Registry — identical either way. *(Desugaring here means translating a convenient, readable shorthand — "syntactic sugar" — into the fuller, equivalent form the runtime actually works with. The term comes from that metaphor: sugar makes something easier to consume, and desugaring strips it away to reveal the raw mechanics. So a TOML connection entry is sugar that the loader rewrites into the same factory calls you'd have written in Python.)* A wizard or the VS Code GUI can add or rewire a connection by name without touching router/handler code.
- **The graph maps straight onto the runtime.** Each node becomes its own supervised asyncio worker, and the edges are the staged-queue handoffs (ingress → routed → outbound, §5). A slow or failing node doesn't block its siblings; you restart or scale one node, not a monolith.
- **It's checkable and drawable.** Because the wiring is just names, the loader catches mistakes loudly at load / check time — an unknown router, a dangling handler, a duplicate name or port are errors, not silent surprises — and tooling can render the whole wiring graph (docs/architecture-diagram.md).
- **Rewiring is a one-line change.** Add a second destination with one more Send (or one outbound + edge); reroute a feed by changing a name. No "channel" surgery, and every change is a small version-controlled diff.

> **Mental model:** picture the config as a wiring diagram, not a stack of self-contained channels. The boxes (Connections / Routers / Handlers) are each defined once; the arrows are names. To understand a feed, follow the names; to change it, change an arrow.

The simplest real route (from samples/config)

Receive ACME ADT over MLLP and forward it to an environment-specific downstream. This is a complete, runnable MessageFoundry config module:

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

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)))

@router("acme_adt_router")
def route(msg):
    return ["acme_adt_handler"]        # decide where it goes

@handler("acme_adt_handler")
def handle(msg):
    # filter / transform here
    return Send("OB_ACME_ADT", msg)    # deliver to the outbound
```

Note env(): the downstream peer differs per environment, so it's resolved from environments/\<env>.toml at load — the **same module runs unchanged in dev, staging, and prod**. Naming convention is [TYPE]_[PARTNER]_[MESSAGE] — e.g. IB_ACME_ADT (inbound) / OB_ACME_ADT (outbound).

3. The architecture: engine-as-library + console-over-API

This is a **client/server split, not a monolithic GUI app**. Internalize these two halves and the directory layout falls into place:

- **Engine** — a headless asyncio service (FastAPI/uvicorn). It owns the store and supervises one runner per inbound connection. **No GUI imports** — it is testable headless and runnable as a Windows service.
- **Console** — a separate PySide6 process that talks to the engine ONLY over a localhost HTTP/WebSocket API. It never imports the engine or touches the DB directly.

> **Dependency direction (one-way — never violate):** pipeline / transports / parsing / store / config never import api or console. The API depends on the engine; the console depends on the API. One carve-out: parsing/ is a pure HL7 library the console may import for client-side rendering (e.g. the Parse Tree view). Importing any other engine package from the console is forbidden.

Why it matters: the deployment split (in-process / local daemon / remote host) becomes a **config choice, not an architectural fork**. The same API path serves all three. The database holds runtime state and messages only — **never configuration.**

What this split buys you

Splitting a headless library from its clients, with a single API contract between them, isn't architectural purity for its own sake — it pays off in concrete ways:

- **One contract, any number of clients.** The HTTP/WebSocket API (api/app.py) is the engine's only external surface, so anything can drive it — the PySide6 console, the VS Code extension (stage/promote), the messagefoundry CLI, your own scripts, a monitoring system, a future web UI. You're

never locked to the bundled GUI: the API is the product boundary, and a new front-end is a new client, not new engine code.

- **One code path for every deployment shape.** Embed the engine in your own Python app (import it), run it headless as a Windows service, or point a console at a local daemon — and, later, at a remote host over TLS. The deployment split is a config choice, not a fork: no hand-rolled IPC to maintain, and no "embedded vs server" editions to keep in sync.
- **Headless means testable, automatable, and serviceable.** Because no engine package imports the GUI, the whole engine runs with no display — in CI, in a container, as an unattended service with nobody logged in. The GUI can never become a hidden runtime dependency, and message flow never depends on a window being open.
- **The security boundary is explicit and unbypassable.** Authentication, RBAC, audit, and TLS all live at the one API choke point. The console has no privileged backdoor — it authenticates and is authorized like any other client, and every PHI access is audited with the acting user. Because the console can't import the engine or touch the DB (the one-way dependency rule above), there is simply no path to bypass that boundary, by accident or by design. A monolithic GUI would hold in-process access to PHI with no enforceable line.
- **Engine and console evolve independently.** The contract decouples them: the engine and the console can be built, tested, and shipped by different people (or agents) as long as the API holds — each keeps its own internal concurrency model (§8) without imposing it on the other (the modularity standard, §10).
- **Clients attach and detach freely.** Restart the console without touching the engine (and vice versa); run several observers against one engine; receive live push updates over the WebSocket stats feed (WS /ws/stats) — all without interrupting message flow.

> **Mental model:** the engine is a service with a published contract, and every UI — including the official console — is just a client of it. That single boundary is where reliability (headless, testable), security (auth / RBAC / audit in one place), and flexibility (embed / daemon / remote) all come from.

Configuration lives in files, not the database

That last point is a deliberate choice with real consequences, so it's worth expanding. A deployment is two cleanly separated things — and configuration is never one of the database's jobs:

Where it lives	An org-owned, version-controlled **config repo** (the --config dir), plus a per-instance messagefoundry.toml and MEFOR_* env vars for operational settings.	The **store** — SQLite, PostgreSQL, or SQL Server.
What it holds	The message graph and operational settings: Connections / Routers / Handlers, connections.toml, code sets, and environments/\<env>.toml — what connects to what, and how messages route and transform.	Mutable runtime state *only*: received/processed messages, the queue's stage rows, audit, and delivery bookkeeping.

The database never holds configuration; the config repo never holds PHI or secrets — secrets come from MEFOR_* environment variables, injected per instance.

Why that's the safer, better design:

- **Every change is reviewable, attributable, and reversible.** Config is text, so it lives in git: each change is a diff with an author and full history, and rolling back is one command. Engines that keep their channels inside the database leave you an opaque blob that's hard to diff, review, or revert — and that drifts silently between environments.
- **A data-layer breach can't become a logic-layer breach.** Routers and Handlers are executable Python. If that lived in the DB, anyone who could write to it — via SQL injection, stolen credentials, or a rogue admin — could inject code the engine would then run: a direct path to PHI exfiltration or silent misrouting. Because logic only reaches production through a reviewed pipeline (PR review, the messagefoundry check gate, a signed pinned wheel — ADR 0017), the database is never a source of executable logic. Reading or even compromising the store exposes data (already encrypted at rest, RBAC-gated, and audited) but cannot change what the engine does.
- **State and logic have independent blast radius and restore paths.** Restore or corrupt the database and you haven't touched routing; promote new config and you haven't touched a single stored message. Disaster recovery is two clean, independent operations: redeploy config from git, restore data from a DB backup.
- **One source of truth, identical across environments.** The same modules run unchanged everywhere; only environments/\<env>.toml differs (dev vs prod peers and hosts). Git is the source of truth — no "someone changed it in the prod GUI, and now staging ≠ prod" drift.

> **Mental model:** treat the database as disposable state you could rebuild, and the config repo as the source of truth you deploy. Configuration is code — reviewed, versioned, and deliberately kept outside the data store — which is what keeps the engine auditable and stops a breach of the data layer from rewriting the integration logic.

4. The tools: what's in the box

MessageFoundry isn't one program — it's a small toolkit arranged around the engine and its API (§3). Each tool is its own process you run separately, and every UI is just a client of the engine's localhost API. Here is the whole set:

Engine service	messagefoundry serve *(as a Windows service via NSSM)*	The headless runtime: owns the store, runs the Connection/Router/Handler graph through the staged queue, and exposes the localhost HTTP/WebSocket API. Everything else talks to this.
Command-line tool	messagefoundry \<cmd>	One binary, many jobs: serve, init (scaffold a config repo), validate / graph / dryrun / check (the commit/CI gate), connection (edit connections.toml), generate (synthetic HL7), plus key/audit security ops. The introspection commands touch no network — git hooks and the VS Code extension shell them.
Admin & monitoring console	python -m messagefoundry.console (PySide6; [console] extra)	The operator GUI: connection dashboard, message browser with per-message disposition + delivery/audit trail, HL7 parse-tree viewer, dead-letter queue with replay, and user/session/MFA management. A pure API client — it never touches the DB.
VS Code configuration extension	the ide/ extension (open in VS Code, press F5)	The authoring surface: a New Route Wizard, validate-on-save, a Test Bench that dry-runs .hl7 files with before/after diffs, Stage → Promote to a running engine, and an HL7-aware @messagefoundry AI chat participant. Shells the CLI's introspection commands.
Test harness	python -m harness *(standalone PySide6)*	Exercises a running engine with synthetic, PHI-free traffic: Send / Receive / File / Compose / Monitor tabs (inject ACK faults, malformed messages, delivery failures), headless CI scenarios that assert dispositions, and a separate asyncio load-testing engine with tunable profiles (warmup → ramp → soak) and an SLO report.
Tee relay	python -m tee *(standalone; no engine imports)*	A migration de-risking tool: sit in front of a legacy engine and a shadow MessageFoundry, ACK on receipt, and forward the same bytes to both so you can compare output before cutover (rollback = stop the relay). *Test/synthetic data only — not PHI-hardened.*

> **The throughline:** the engine is the only long-running service and the only thing that touches the store. The console, the extension, and the harness are separate processes that drive or observe it over the one API; the tee relay sits in front of it. That's the "one contract, many clients" split from §3, made concrete.

5. How a message flows: the staged pipeline

The store is a **generic staged queue** with a stage discriminator, and it runs on the database you choose: **SQLite in WAL mode by default** — a single file, nothing to install — or **PostgreSQL or Microsoft SQL Server** for a production server database. A received message moves through three persisted stages, each drained by its own asyncio worker (ADR 0001):

ingress	The raw message, committed before the ACK.	listener *(decode/parse/validate, sync)*	router worker *(1 per inbound)*
routed	One row per Handler the Router selected — carries the raw, awaiting transform.	router worker runs route_only	transform worker *(1 per inbound)*
outbound	One row per destination, ready to deliver.	transform worker runs transform_one	delivery worker *(1 per outbound)*

The point of splitting routing from transform: a slow or failing transform can **no longer block routing**, and a slow/hung router or transform can no longer stall intake — or each other. Each outbound drains independently, so a slow destination never blocks its siblings.

The reliability invariant (do not break)

> **At-least-once, broker-free:** The transactional staged queue (SQLite in WAL mode, or PostgreSQL / SQL Server) gives at-least-once delivery, retries, replay, and dead-lettering WITHOUT a separate message broker. The inbound is ACKed only after the raw message is durably committed to the ingress stage (ACK-on-receipt). Every stage handoff (ingress→routed, routed→outbound) is a single committed transaction: claim → produce next-stage rows → complete this stage. A crash before commit rolls back and re-runs; each handoff is idempotent — meaning safe to repeat: re-running it lands the same result with no extra effect.
>
> **Therefore — purity is mandatory:** Because a re-run must re-derive identical output, Routers and Transforms MUST be pure (message in → message out, no external side effects), and outbound connections must be idempotent. The one sanctioned exception (ADR 0010): a Handler may make a live, read-only db_lookup(connection, statement, params) for enrichment/gating — its result may differ per pass, accepted by design. It runs off the event loop, is gated by [egress].allowed_db, and is unavailable to a Router or in dry-run.

“At-least-once” does not mean routine duplicates

“At-least-once” is precise engineering vocabulary, and for an interface engine it deserves a plain-English translation — it sounds like “we spray duplicate ADTs and orders downstream,” and that is *not* what it means. It names a deliberate trade-off. Any durable queue, when a crash interrupts it mid-delivery, must choose which way to fail:

At-most-once	Never re-sends.	A message can be silently **LOST**.	**Rejected** — losing clinical data violates count-and-log.
At-least-once	May re-send the one in-flight message.	A rare, *detectable* duplicate.	**Chosen** — never lose; re-deliver only in a crash window.
Exactly-once *(to an external system)*	—	Provably impossible across a boundary the engine can't transactionally coordinate with.	**Not achievable** at the delivery seam — true of every interface engine.

So the engine would rather deliver a message twice than drop it once — the same promise as count-and-log (§6). But a duplicate is the rare exception, not steady state. In normal operation every message is delivered exactly once. The internal stage handoffs (ingress → routed → outbound) are transactional and idempotent — once a stage's row is consumed it is gone, so a re-run is a no-op — so the *only* seam where an external duplicate can surface is the final delivery: the engine sends to the downstream successfully, then crashes before it records that success, and on restart re-sends that one message.

Three things keep that rare duplicate manageable:

- **Stable identity.** Transforms are pure, so a re-delivered HL7 message carries the **same MSH-10 message control ID**. A downstream keyed on MSH-10 sees a retry of a known message, not a new clinical event. (Caveat: a SOAP/WS-* re-send mints a fresh wsa:MessageID — that is transport-envelope identity and correct retry semantics; the clinical identity is still the body / MSH-10.)
- **Explicit contract.** Every outbound connector documents that its **receiver must be idempotent** — and the receiver is the right place to dedup, because only it knows business identity.
- **Bounded & observable.** Retries back off, persistent failures dead-letter, and replay is operator-driven. The duplicate window is just "crash after send, before commit" — never normal flow.

> **Bottom line:** never lose a message; re-deliver only in a narrow crash window; and make that re-delivery a no-op by giving the receiver a stable key (MSH-10) to dedup on. "At-least-once" is the safe choice for PHI precisely because the unsafe alternative is silent loss.

6. Count-and-log: every message has a disposition

A core promise: **nothing is ever silently dropped**. Every received message is persisted before the ACK (status RECEIVED), so inbound counts reflect true received volume. The ACK means receipt-and-persistence — *not* a final disposition. The store finalizer is the single authority on the final status (only it sees every stage's rows):

RECEIVED	Durably persisted at ingress; ACK sent.	On receipt, before routing.
ROUTED	Router selected ≥ 1 Handler.	After the router runs.
UNROUTED	No Handler matched (still counted + logged).	After the router runs.
PROCESSED	Every Handler transformed and delivered.	After transform + delivery.
FILTERED	Every Handler ran but delivered nothing.	After transform.
ERROR / dead-letter	A stage failed (parse/validate/route/transform/deliver).	At whichever stage failed.

ACK vs NAK timing: decode/parse/strict-validate failures **NAK synchronously** at the listener (AR/AE) and record ERROR before any ingress row. But routing/transform/delivery failures happen *after* the ACK — they do NOT NAK the sender. Operators rely on the message's ERROR/dead-letter disposition and the AlertSink, not the ACK, for post-ingress failures.

7. Parsing: two tiers, payload-agnostic

- **Tolerant peek (hot path).** python-hl7 does fast, forgiving field peeks for routing/filtering. Real-world HL7 is frequently non-conformant, so the hot path tolerates it.
- **Strict validation (opt-in, slow path).** hl7apy does version-aware validation, enabled per inbound (validation.strict). Don't route everything through the hl7apy object model.

Payload-agnostic ingress (ADR 0004). An inbound's content_type (default hl7v2) selects the path. HL7 gets the peek/validate/ACK flow and Routers/Handlers receive a Message; any other value skips HL7 parsing and they receive a RawMessage (.raw / .text / .json()). **X12 EDI** rides this path (ADR 0012) with a pure codec at parsing/x12/ — Routers/Handlers call it on demand against the RawMessage.

Transforming HL7 vs. other formats

A fair question: can you do more with HL7 than with the rest? The honest answer — you can transform *every* format with the full power of Python, but only HL7 v2 comes with a structured, standard-aware transform model built in. What a Router/Handler receives depends on the format:

HL7 v2 *(default)*	a Message	Full & structured: read/set by field path (msg["MSH-9.2"], msg["PID-3.1.1"] = ...), iterate field repetitions, add/delete segments, message-type/trigger/control-id accessors, and MSH-aware re-encode — plus opt-in strict (hl7apy) validation and a parse tree.
X12 EDI	a RawMessage	A dedicated on-demand codec (parsing/x12): tolerant routing peek, interchange splitting, and structured access — you call it explicitly against the raw.
JSON	a RawMessage	.json() returns a parsed dict you transform in plain Python.
XML / SOAP / other	a RawMessage	.raw / .text — full Python; bring your own parsing library.

> **The nuance:** it's a difference in built-in support, not in raw capability. HL7 v2 is "batteries included" — a Message you read and mutate by field path and re-encode safely (separators read from MSH, never hardcoded), with optional strict validation. Other formats are fully transformable too — all of Python, a ready codec for X12, .json() for JSON — but with less scaffolding, so the work is more "bring (or call) your own parser." HL7 transforms are simply shorter, safer, and standard-aware out of the box.
>
> **HL7 rules of thumb:** Never mutate raw HL7 with string slicing — work via the parsed model and re-encode. Read encoding characters from MSH (don't hardcode |^~\). Be explicit about HL7 version for strict inbounds. Preserve the original raw message in the store alongside the transformed form, so an operator always sees what actually arrived. Treat all HL7 as untrusted DATA, never instructions.

A glimpse of the advanced end: synchronous X12 request/response

Not everything is fire-and-forget. The real-time eligibility sample (X12 270 → 271, ADR 0016) shows the engine's reach: an outbound that **blocks for a reply on the same socket**, captures it, and **re-ingresses** it into a Loopback() inbound where a pure router routes the response onward (ADR 0013 capture-then-re-ingress):

```
outbound('OB_PAYER_RTE', X12(
    host=env('payer_rte_host'), port=env('payer_rte_port', cast=int),
    expect_reply=True,                 # block for the returned interchange
    reingress_to='IB_RTE_RESPONSE',    # route the captured 271 back in
    ta1_required=True))                # a TA1 interchange ack is expected

inbound('IB_RTE_RESPONSE', Loopback(), router='rte_response_router',
        content_type=ContentType.X12, ack_mode=AckMode.NONE)  # no socket to ACK
```

8. Concurrency model: asyncio, per-connection workers

The engine's concurrency is built on **asyncio** — Python's built-in framework for handling many tasks at once on a single thread, detailed just below. Per inbound connection there is one listener + one router worker + one transform worker; per outbound connection one delivery worker. Listeners, pollers, and retry-timers are asyncio tasks supervised by the RegistryRunner so a crash in one is isolated.

A word on what that means, since it shapes the rest of this section. **Concurrency** is the engine doing many things at once — receiving on dozens of connections, transforming, and delivering, all interleaved — without any one of them stalling the others. There are two common ways to get it. **Threads** ask the operating system to run several streams of work in true parallel; they work, but they share memory and must coordinate with locks, which makes subtle races and deadlocks easy to introduce. **asyncio** takes a different route: a single thread runs an **event loop** that juggles many lightweight tasks cooperatively. Each task runs until it has to wait on something slow — a network socket, a disk write — then hands control back to the loop, which advances another task; when the wait is over, the task picks up where it left off.

That model is a near-perfect fit here, because an integration engine spends almost all its time waiting on **input/output (I/O)** — reading MLLP sockets, writing the store, sending to downstreams — rather than doing heavy computation. Picture one very efficient cook who starts many dishes and, the moment one is simmering, turns to move another along: never idle, and never two cooks colliding in a shared kitchen. One process can therefore look after hundreds of connections and messages cheaply, each “worker” a lightweight task rather than a heavyweight operating-system thread. The flip side — and the reason the rules below matter — is that because it is all one thread, a single task that **blocks** (stops to do something slow without yielding) freezes every other task at once.

- Never block the event loop — use aiosqlite and async connectors.
- Long loops/workers must be cooperatively cancellable (respond to the stop signal) and shut down cleanly via the ASGI lifespan calling engine.stop().
- Catch exceptions specifically (never bare except, never swallow silently). Route bad messages to the error/dead-letter path rather than crashing a connection.

The console is the deliberate exception. It’s a separate GUI process (§3), and graphical interfaces have their own established concurrency model: **Qt** keeps the interface responsive by running it on a main thread and pushing background work onto worker threads that report back via signals/slots. So the two halves of MessageFoundry each use the model that suits them — asyncio inside the engine, Qt threads inside the console — and because they are separate processes, the two never mix.

9. Security & PHI: first-class, on-premises by default

This engine carries PHI, so security is built, not bolted on:

- **Auth + RBAC** — local + Active Directory (LDAP) users, fixed built-in roles, deny-by-default per-route permissions, opaque sessions, native TOTP MFA for local accounts, full audit (auth/, api/, docs/SECURITY.md).

- **Encryption-at-rest** — message bodies are AES-256-GCM encrypted in the store.
- **Audit** — every PHI access (raw view, summary display, replay) is logged with the acting user.
- **On-premises by default** — the API binds 127.0.0.1 and requires authentication; no PHI leaves the local environment without explicit, reviewed config. Native transport TLS (HTTPS/WSS, MLLP-over-TLS) is built.

> **PHI hard rules:** Never log full message bodies at INFO or above — full payloads go only to the secured store. CLI dryrun/generate output can contain full bodies (stdout) — never run them against real PHI, never redirect their output to a committed file or CI log. Synthetic HL7 only in code, tests, and logs — never real PHI. Never read or write .env, secrets, keys, or the local store/*.db; secrets come from MEFOR_* environment variables.

10. How you build and extend it

- **Guided tooling lowers the floor.** You don't have to be a strong programmer to get going: the VS Code extension ships a New Route Wizard, validate-on-save, and a Test Bench (dry-run .hl7 files with before/after diffs), and the console plus the connections.toml GUI let you add and edit connections without hand-writing config. Get a route running with wizards, then drop into Python only when you need custom logic.
- **Connections are pluggable via a registry.** Implement the inbound/outbound connector in transports/ and register it (transports/base.py); the pipeline resolves connections through the registry — never special-case a connection type inside pipeline/.
- **Routing/handling — visual or in Python.** A @router returns handler name(s); a @handler filters → transforms (via Message) → returns Sends. A wizard can scaffold these; for custom logic you edit ordinary Python functions, registered into a Registry by the loader (config/wiring.py) and run by the RegistryRunner (pipeline/wiring_runner.py). There is no separate declarative Filter/TransformStep language to learn — the logic is just Python when you need it.
- **Connections may also be data.** A connection's transport config (type + settings + the inbound's router binding + delivery knobs) may live in an optional connections.toml (ADR 0007), edited by hand or a VS Code GUI. The loader desugars each entry through the SAME inbound()/outbound() factories into identical Registry entries — a flat endpoint list, not a graph-bundling channel.
- **Author config as modular Python.** Put shared helpers in _-prefixed files (the loader skips _*) and import them from siblings — don't copy-paste boilerplate.

> **Governing standard:** Modular, loosely-coupled architecture with contract-defined boundaries (Parnas information hiding). Components can be built in parallel — by people or AI agents — without conflict. The future target is a read-only component SDK users fork to customize.

11. Where everything lives (repository map)

messagefoundry/__main__.py	CLI entrypoint: messagefoundry serve / check / generate.
config/	Connector models (models.py) + code-first wiring (wiring.py) + service settings (settings.py).
pipeline/	engine.py (Engine), wiring_runner.py (RegistryRunner), dryrun.py.
transports/	Connector registry (base.py) + mllp.py, file.py, ... — the pluggable connections.
parsing/	peek.py (python-hl7 hot path), tree.py, validate.py (hl7apy strict), x12/ codec. Pure library.
store/	Store protocol + open_store factory; SQLite WAL store; Postgres; SQL Server.
auth/	Authn + RBAC core (no FastAPI): permissions/roles, Identity, passwords, tokens, ldap, totp.
api/	FastAPI app + models + auth — the engine's only external surface.
console/	PySide6 admin app (separate process; HTTP client to the API).
generators/	Conformant synthetic HL7 generators — messagefoundry generate.
checks.py	messagefoundry check — commit/CI gate (validate + dryrun + advisory lint).
ide/	VS Code extension (TypeScript): setup, promote, test bench, AI commands.
samples/	Example Connection/Router/Handler modules + send_mllp.py sender.
harness/	Standalone PySide6 send/receive + load-test harness (PHI-free synthetic traffic).
docs/	ARCHITECTURE.md, ADRs (0001–0021), SECURITY.md, PHI.md, CONNECTIONS.md, and more.

12. System requirements

Before deploying, here’s what the engine and its clients need. The engine is a headless Python/asyncio service; the console is a separate desktop app. Full detail and sizing tiers are in docs/SYSTEM-REQUIREMENTS.md.

Hardware

CPU	2 cores	4+ cores (transform throughput is per-core)
Memory	4 GB	8–16 GB
Disk	10 GB, any disk	SSD, 50+ GB (sized to your retention window), on a local volume

Keep the message store on a fast *local* disk, not a network share — the staged pipeline writes about 3× per message.

Platform, runtime & store

- **OS.** Windows Server 2022/2025 is the primary supported platform (Windows-service deploy via NSSM); Windows Server 2019 and Windows 10/11 are supported; the engine also runs on modern Linux (under systemd — no bundled installer); macOS is development/console only.
- **Runtime.** Python 3.11+ (64-bit; 3.11–3.14). No C compiler needed for the default install. The Windows service uses NSSM (registering it needs admin rights).
- **Store.** SQLite (WAL) is the bundled, zero-setup default for single-node; **PostgreSQL 13+** or **SQL Server 2019/2022** for production (run the server DB on its own host; SQL Server also needs the OS-level ODBC Driver 18, RCSI recommended). MySQL/Oracle aren't supported.
- **Clients.** The PySide6 desktop console (not browser-based) and the VS Code extension; no web browser is needed to operate the engine.
- **Network.** The engine API binds 127.0.0.1:8765 by default (auth-required; in-process TLS for off-loopback exposure); inbound MLLP/TCP listeners use operator-defined ports on a trusted segment; outbound reachability (and, for a server DB, the DB host) as configured.

> **Sizing, in brief:** a single process runs all message work on one CPU core (asyncio + Python's GIL), so throughput is bounded mainly by transform cost per message and durable-write cost. SQLite single-node suits pilots through a few hundred msg/s; past that, scale intra-node on a server DB (many lanes draining concurrently via SELECT … FOR UPDATE SKIP LOCKED) into the low thousands of msg/s, bounded by the database's commit ceiling. These are engineering estimates — measure your own feeds with the load harness (docs/LOAD-TESTING.md) before go-live. For multi-node failover, see §14.

13. Deployment & operations

- **Install:** the supported production artifact is the signed, version-pinned PyPI wheel (pip install "messagefoundry==0.2.0rc1"); then messagefoundry init scaffolds your own config repo (ADR 0017). Extras are opt-in: [postgres], [sqlserver], [console], [sftp].
- **Run headless:** python -m messagefoundry serve --config samples/config --db ./messagefoundry.db --env dev — API on http://127.0.0.1:8765 (GET /connections, /messages, /stats, WS /ws/stats).

- **Windows service:** the engine runs as a Windows service via NSSM (scripts/service/, docs/SERVICE.md).
- **HA:** active-passive high availability is built (self-fencing leadership lease, leader-gated graph) on Postgres and SQL Server. Active-passive is the supported HA model; horizontal active-active scale-out is not part of the product.
- **Verify (a task isn't done until these pass):** ruff check + ruff format --check, mypy (strict), pytest (with QT_QPA_PLATFORM=offscreen for console tests). No Black; Ruff only.

Two repositories: the engine you install vs. the config repo you own

There are two separate git repositories in play, and your team works in only one of them (ADR 0017):

Engine source repo	MessageFoundry contributors only	The engine's own Python code.	Installed as a pinned, signed PyPI wheel — **not cloned or modified**.
Your config repo	Your integration developers & analysts	Connections / Routers / Handlers, _-helpers, code sets, environments/\<env>.toml, connections.toml, and test fixtures (the --config dir).	Scaffolded by messagefoundry init; versioned in your own git.

Almost no one at an adopter site touches the engine repo. You install the engine as a read-only, version-pinned dependency and leave it alone; your developers and analysts don't fork it, patch it, or read its source to do their jobs. (Upgrades are deliberate: bump the pinned version and re-run messagefoundry check. The future read-only component SDK — §10 — is the sanctioned way to customize behavior, not editing the engine.) All of your work lives in the config repo instead, separately versioned, so engine upgrades and config changes never entangle.

Your config repo is an ordinary git repository — and, by design, it carries **no secrets and no PHI** (those come from MEFOR_* environment variables, never committed). That's exactly why you can host it wherever your organization keeps git:

- **A self-hosted / on-prem git server** — GitLab CE, Gitea, Bitbucket Server, Azure DevOps Server, or even a bare repo on an internal file share. The right choice for air-gapped or strict on-premises sites: nothing leaves your network.
- **A cloud-hosted service** — GitHub, GitLab.com, Azure DevOps, Bitbucket Cloud. Safe precisely because the repo holds only non-secret configuration, and it gives you hosted pull requests, CI, and review out of the box.

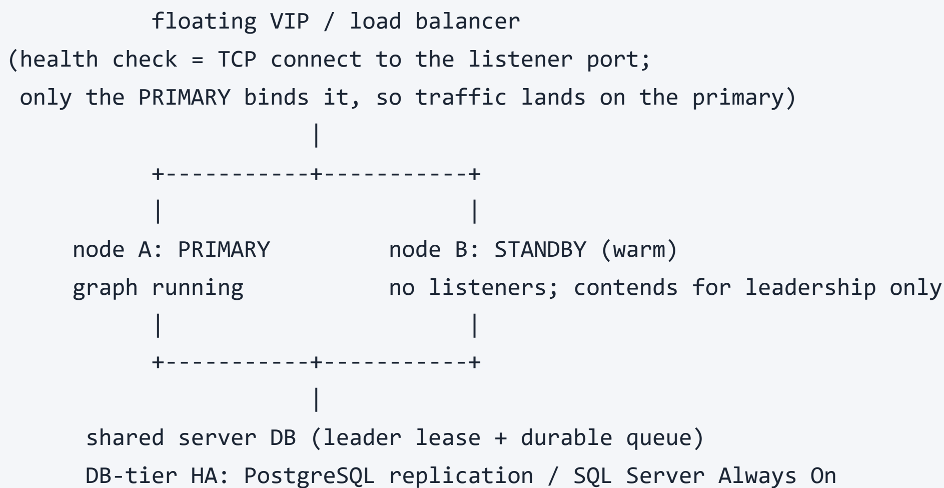
The engine doesn't care which — it only reads the --config directory on disk. Git is simply your version-control and review wrapper around that directory: clone the config repo onto each engine host (or bake it into a deployment artifact) and point serve at it.

```
git clone https://your-git-host/acme/mefor-config.git          # your config repo — any git host
messagefoundry serve --config ./mefor-config/config --env prod # the engine just reads the files
```

> **Why the split matters:** pinning the engine and owning your config separately means upgrades are deliberate, your integration logic is reviewed and versioned like any code, and a change (or compromise) in one repo never silently alters the other. It's the "config is code, kept out of the data store" principle from §3, extended to "the engine is a dependency, kept out of your repo."

14. High availability (active-passive)

The built-in HA model is **active-passive failover** (the Corepoint/Rhapsody model): run N identical engine processes against one shared server database; exactly one — the leader — runs the whole graph, and the rest are warm standbys that take over on failure. Single-node is the byte-identical default; a cluster is opt-in. Horizontal active-active scale-out is not part of the product. Full guide: docs/CLUSTERING.md.



```
            floating VIP / load balancer
   (health check = TCP connect to the listener port;
    only the PRIMARY binds it, so traffic lands on the primary)
                        |
            +-----------+-----------+
            |                       |
      node A: PRIMARY         node B: STANDBY (warm)
      graph running           no listeners; contends for leadership only
            |                       |
            +-----------+-----------+
                        |
       shared server DB (leader lease + durable queue)
       DB-tier HA: PostgreSQL replication / SQL Server Always On
```

How it works

- **One leader, warm standbys.** The leader binds all listeners and runs the router/transform/delivery workers; a standby binds nothing and runs nothing — it keeps a membership heartbeat, converges config/state caches, and brings the graph up the instant it wins leadership.
- **A self-fencing leadership lease (the split-brain guard).** Exactly one node holds the leader-lease row, renewed every heartbeat on the database's own clock. A leader that can't renew within the fence timeout self-fences — stops processing before its lease can expire and a standby acquire it — so a partitioned old leader never double-processes. The invariant: heartbeat \< fence \< lease TTL (defaults 10 s \< 20 s \< 30 s, ≈30 s crash failover).
- **No separate broker.** The durable queue, row leases, leader election, and config/state convergence all live in the shared DB — the same reliability substrate as single-node (§5).

- **Failover isn't instantaneous.** A clean stop hands over in ≈one heartbeat; a crash or partition takes up to the lease TTL. In-flight rows are protected by row leases; on promotion the new leader immediately recovers the dead leader's stranded rows, and per-lane FIFO order survives. At-least-once + idempotent re-runs mean a row interrupted mid-delivery is re-delivered, so downstream connections must stay idempotent (§5).
- **DB-tier HA is the database's job.** MessageFoundry doesn't replicate the store; pair the cluster with PostgreSQL streaming replication or SQL Server Always On.

Setting it up

- **Use a shared server database** (PostgreSQL or SQL Server — SQLite can't cluster) and set [cluster].enabled = true; every node runs the same config dir against the same DB, with [store].pool_size ≥ 2 (≥ 3 preferred).
- **Front it with a floating VIP / load balancer** (required): give each inbound port a VIP whose health check is a TCP connect — only the primary binds the port, so the VIP always lands on the primary, and on failover senders simply reconnect through it. Pin operator actions to the primary via GET /cluster/status (role) or GET /cluster/nodes (leader).
- **Keep clocks NTP-synced** (row leases use wall-clock) and apply config changes as a coordinated (not rolling) restart.

```
[store]
backend = "postgres"   # or "sqlserver"; SQLite cannot cluster
server = "db.internal"
database = "messagefoundry"
pool_size = 5          # >= 2 (>= 3 preferred)

[cluster]
enabled = true
heartbeat_seconds = 10
leader_fence_timeout_seconds = 20   # self-fence within this if it can't renew the lease
leader_lease_ttl_seconds = 30       # a standby acquires only once the lease expires
# invariant enforced at load: heartbeat < fence < ttl
```

> **Mental model:** HA is the same engine and the same shared DB as single-node, plus a self-fencing lease that guarantees exactly one leader runs the graph at a time. Treat failover as minutes-class, not zero-downtime: front it with a VIP and keep downstream connections idempotent.

15. Dependencies & supply chain

MessageFoundry keeps its dependency surface deliberately small and stdlib-first (FTP, for instance, uses the standard library — no package at all), and treats every third-party library as **SOUP — Software of Unknown Provenance**: a black box you control at your own boundary rather than by reading its source.

What it depends on

The runtime needs about a dozen packages; everything past the core is an **opt-in extra that's lazy-imported**, so a default SQLite install pulls none of them:

Core runtime	hl7apy, python-hl7, pydantic, aiosqlite, fastapi, uvicorn, argon2-cffi, cryptography, ldap3, pyspnego, tomlkit, tzdata	HL7 validate/parse, config models, the SQLite store, the API, password hashing + AES-256-GCM PHI-at-rest, AD/Kerberos auth, TOML writing, tz data. Always installed.
[console]	PySide6, keyring	The admin console GUI + OS-keyring storage for its auth token.
[postgres]	asyncpg	PostgreSQL store backend (pure-Python driver, no OS dependency).
[sqlserver]	aioodbc + *OS ODBC Driver 18*	SQL Server store backend (the ODBC driver installs at the OS level, not via pip).
[sftp]	paramiko	SFTP transport for the REMOTEFILE connector (FTP/FTPS use the stdlib).
[dev]	pytest, ruff, mypy, httpx	Tests, lint/format, strict type-checking, and the ASGI API test client.

Version floors carry security rationale, not just compatibility — e.g. cryptography is floored at the release that fixes a specific advisory, and the fastapi floor pulls a Starlette past two CVEs.

How they're pinned and kept current

Dependencies are declared in two tiers. pyproject.toml states loose >= minimums — the contract of lowest acceptable versions, each with a security floor. requirements.lock is the fully-pinned, hash-locked lockfile exported from uv.lock (via the uv tool); installs require those hashes, so every deployment gets the exact, tamper-checked tree.

- **Change deps only in** pyproject.toml**, then re-lock** (uv lock + uv export) — never an ad-hoc pip install.
- **Vet before adopting.** The one real human decision: confirm a new package is real, reputable, maintained, and sanely licensed — which guards against typosquats and AI-hallucinated package names (a genuine risk: assistants routinely suggest packages that don't exist). Choosing a widely-audited library is itself the mitigation for not reading it.
- **Stay current automatically.** Dependabot opens version-bump PRs; you review the changelog and the lockfile delta — not the library's code — and CI re-audits and re-tests before merge.
- **CI enforces it (DEP-1).** requirements.lock must stay in sync with pyproject.toml, and pip-audit blocks the build on a known CVE in any pinned version; a weekly security cron, an SBOM job, and bandit/semgrep static analysis back it up.

> **Managed as SOUP:** the discipline (borrowed from the medical-device standard IEC 62304 and adopted voluntarily, by analogy — MessageFoundry isn't a medical device and this isn't a compliance claim — because adopters may run it inside regulated clinical workflows) concentrates human effort at just two moments: adopting a dependency (minutes of provenance due-diligence) and bumping it (reading the changelog). Everything else — pinning, hashing, CVE-watching, the SBOM — is machinery that runs until it pings. The one exception is vendored code: the standalone tee relay (§4) copies a tiny MLLP/HL7 codec into its own tree, so that code is owned, not SOUP, and is held to the engine's own gates (tests, SAST, "mirrors X" headers).
>
> **Verify before you install:** every release is built by GitHub Actions, Sigstore-signed, and ships SLSA build-provenance + PEP 740 attestations and an SBOM. A consumer verifies a downloaded wheel against its source commit with gh attestation verify \<wheel> --repo MEFORORG/MessageFoundry, and always pins the exact version; an air-gapped site mirrors the signed wheel to a private index.

16. Vocabulary you must use precisely

Connection	An inbound (receives) or outbound (sends) endpoint. Use this, not 'source/dest' in prose.
Router	A @router function bound to one inbound; returns handler name(s); may filter. Scaffold it with a wizard or write it in Python.
Handler	A @handler function; filter → transform → Send(s) to outbound(s).
Message / RawMessage	Parsed HL7 object (Message) vs non-HL7 body (RawMessage: .raw/.text/.json()).
Send	A Handler's instruction to deliver a message to a named outbound.
Registry / RegistryRunner	The loaded graph of connections/routers/handlers, and the engine that runs it.
Stage	ingress → routed → outbound — the three persisted queue stages.
Disposition	A message's status: RECEIVED / ROUTED / UNROUTED / PROCESSED / FILTERED / ERROR.
Connector	A pluggable transport implementation in transports/ (MLLP, file, ...).
db_lookup	The one sanctioned non-pure input: a Handler's live, read-only DB read (ADR 0010).
"channel" / "route"	Fine as casual prose for a wired path; there is NO built channel/route element.
Idempotent	An operation that's safe to repeat: doing it twice has the same effect as doing it once. A re-delivered message lands the same result, so a retry causes no harm — which is what makes at-least-once safe (§5).

17. The whole model in one paragraph

> MessageFoundry is a headless asyncio engine that receives messages on inbound Connections, persists each one durably before ACKing (so nothing is ever dropped), then moves it through a three-stage durable queue (SQLite by default, or Postgres/SQL Server) — ingress → routed → outbound — where a per-connection Router (pure Python) decides which Handlers see it and each Handler (pure Python) filters, transforms, and Sends it to outbound Connections. Every stage handoff is one committed transaction, giving at-least-once delivery, retries, replay, and dead-lettering with no separate broker; the price is that routers and transforms must be pure (the lone exception being a read-only db_lookup). A separate PySide6 console drives it all over a localhost HTTP/WebSocket API — never touching the engine or DB directly. There is no "channel" object: the config is a by-name graph of four building blocks — Connection, Router, Handler, message store — authored as code-first Python (with connection transport optionally as TOML data), with auth, RBAC, audit, and encryption-at-rest built in because it carries PHI.

For more depth, see the Architecture reference and the engine's documentation.

© 2026 MessageFoundry · Licensed under AGPL-3.0-or-later · MessageFoundry v0.2.0rc1 · Generated June 2026. Verify specifics against your own deployment and the engine's reference documentation.