Architecture Overview

A modular, contract-bounded design

MessageFoundry is built as a **modular, loosely-coupled architecture with contract-defined boundaries (information hiding).** Components are decomposed so that a change inside one rarely forces changes in others, and so that separate teams can build in parallel against stable interfaces. This is the governing principle; the topology, store, and module map below are all instances of it.

Modular design. The system is decomposed into independent components: the headless **engine**, the **code-first configuration** authored against it (Connections / Routers / Handlers plus per-environment values), the **monitoring console**, the **VS Code IDE extension**, the **test harness**, and the **CLI tools** (`generate` / `check` / `dryrun` , alongside `serve`) — with the Windows service and CI as the operational wrapping. Within the engine itself the boundaries are `config` / `parsing` / `store` / `transports` / `pipeline` / `auth` / `api` . Modularity bounds how much of the system any one change has to hold and lets concurrent work proceed without conflict.

Information hiding. The foundational principle is information hiding / encapsulation, after David Parnas's 1972 paper "*On the Criteria To Be Used in Decomposing Systems into Modules.*" Each module hides a design decision behind an interface, specifically so that separate people can work on modules in parallel and a change inside one doesn't ripple out to others.

Separation of concerns — high cohesion, loose coupling.

- **High cohesion** — everything related to one concern lives inside one component (the engine's message processing; the console's UI). A change stays local.
- **Loose coupling** — components depend on each other as little as possible, and only through stable seams. Low coupling keeps the blast radius of any change small.
- **Why it works:** the risk of conflicting changes scales with shared surface area. Loose coupling minimizes that surface; high cohesion keeps each change confined to one module.

Contract-first / interface-driven design. Loose coupling only holds if the seams are explicit, so the interface is agreed first and each side builds behind it independently. In MessageFoundry:

- the **HTTP / WebSocket API** is the contract between the engine and its clients (console, IDE);
- the **connector registry** is the contract for pluggable transports;
- a **one-way dependency rule** (`pipeline` / `transports` / `parsing` / `store` / `config` never import `api` or `console`) keeps the seams from leaking.

Organizational dimension. Systems tend to mirror the communication structure of the teams that build them (Conway's Law). Module boundaries are drawn deliberately so that work can be split cleanly, and each

component has a single owner at a time (a bounded context, in domain-driven-design terms).

Topology: engine-as-library plus a local API

The engine is an importable Python package (`messagefoundry`). Clients — primarily the PySide6 console — drive it over an HTTP + WebSocket API (FastAPI / uvicorn). The same API serves three deployment shapes without code changes:

- **Embedded** — another Python program owns the engine in-process via `create_app(engine)` (for example the async test client). The console is never embedded; it is always a separate process that reaches the engine only over the API.
- **Local daemon** — the engine runs as a Windows service or Linux daemon, and the console attaches over the API.
- **Remote** — the same API over the network, with authentication and TLS.

The logical boundary (the library API) comes first; whether the engine and its clients run in one process or several is a deployment choice rather than an architectural one.

Deployment guidance: the API binds to `127.0.0.1` (loopback) by default. Exposing a listener off loopback should always be done over TLS, with the interface authentication described under *Security* below.

Topology, described

Clients are separate processes that reach the engine only through the API, and the engine packages never import `api` or `console` :

- **Clients** — the PySide6 console, the VS Code extension, and the test harness. The console and IDE talk to the engine only through the API. The test harness can additionally exercise transports directly (for example, MLLP send/receive). A client may import the pure parsing library directly.
- **API surface** — the FastAPI + uvicorn HTTP/WebSocket layer, fronted by `auth` (authentication + RBAC, deny-by-default, with a hash-chained audit log). This is the engine's only external surface.
- **Engine (headless, asyncio, no GUI imports)** — `pipeline` (the registry runner: listener, router, transform, and delivery workers); `transports` (the connector registry — MLLP, File, X12, FHIR, DICOM C-STORE SCP); `parsing` (a pure HL7 / X12 / DICOM library); `store` (the staged queue, encrypted at rest); and `config` (code-first wiring of Connections, Routers, Handlers, and environments).

The API depends on the engine. The engine packages depend only on each other in one direction, never on the API or console.

The message store *is* the queue

This is the single most important reliability decision. MessageFoundry uses a transactional **staged queue** built on SQLite in WAL mode — one generic `queue` table with a `stage` discriminator (`ingress` | `routed` | `outbound`). It implements the full router / transform split:

```
inbound msg ─▶ decode / parse / (strict-validate)   [synchronous — still NAK on failure]
                    │
                    ▼ persist raw to the INGRESS stage  ─▶ commit ─▶ ACK source   [ACK-on-receipt
                    │
                    ▼ router worker (per inbound): run Router
              produce one ROUTED row per selected handler + complete the ingress row   (one txn)
                    │
                    ▼ transform worker (per inbound): run that handler's transform
              produce an OUTBOUND row per destination + complete the routed row   (one txn)
                    │
                    ▼ per-destination delivery worker
              deliver ─▶ mark done | mark failed + reschedule (retry policy)
```

The **ACK is sent on receipt** — once the raw message is durably committed to the ingress stage, before routing, transform, or delivery — so a slow or hung downstream stage can no longer stall intake.

Each stage **handoff** is a single committed transaction (claim → produce next-stage rows → complete this stage), so a message is never lost or partially handed off. A crash before commit rolls the stage back and it re-runs; each handoff is idempotent against the re-run because the consumed row is gone, and a startup recovery pass reclaims in-flight rows of every stage. This gives **at-least-once delivery, retries, and replay without a separate message broker.** Because at-least-once relies on a re-run re-deriving identical output, **routers and transforms must be pure** and **outbound connections must be idempotent**; correlation IDs are persisted for de-duplication. Each destination drains independently — a slow or failed one never blocks its siblings, and a slow transform never blocks routing.

Disposition flows with the message, decided by a single store finalizer: `RECEIVED` at ingress → `ROUTED` / `UNROUTED` after the router → `PROCESSED` (all delivered) / `FILTERED` (every handler ran but delivered nothing) / `ERROR` (dead-lettered at any stage) once nothing is still in flight. The finalizer is the single authority: it never finalizes while any earlier-stage row is still pending, so a delivered handler can't mark a message done while a sibling handler's routed row still awaits transform. The ACK means receipt-and-persistence, not a final disposition — a routing or transform failure happens after the ACK, so it is logged and dead-lettered (operators rely on the disposition and on alerts) rather than NAK'd.

The router / transform split keeps write amplification modest — roughly three durable transactions per single-handler message, plus one per additional handler.

The staged flow, described

Step	What happens	On failure
Listener	Decode, parse, optionally strict-validate the inbound message (MLLP / File / X12 / FHIR / DICOM).	Synchronous NAK (AR/AE) + `ERROR`, before anything is persisted.
Ingress stage	Raw message committed durably; **ACK (AA) returned on receipt**.	—
Router worker (per inbound)	Run the pure `@router`; produce one routed row per selected handler.	Post-ACK: logged + dead-lettered.
Routed stage	One row per selected handler.	—
Transform worker (per inbound)	Run the pure `@handler` transform; produce one outbound row per destination.	Post-ACK: logged + dead-lettered.
Outbound stage	One row per destination.	—
Delivery worker (per outbound)	Idempotent send, with retry and dead-letter.	Mark failed + reschedule per retry policy.
Store finalizer	Single disposition authority; records `RECEIVED` → `ROUTED` / `UNROUTED` → `PROCESSED` / `FILTERED` / `ERROR`.	—

Each `==>` handoff above is a single committed transaction.

Concurrency

The engine has an asyncio core. There is one listener plus a **router worker** and a **transform worker** per inbound connection, and one delivery worker per outbound connection. Listeners (MLLP / TCP), pollers (file / DB), the router and transform workers, and retry timers are all asyncio tasks supervised by the registry runner, so a crash in one is isolated and the worker is respawned. The router and transform workers keep draining their inbound's already-ACKed messages even while the source is stopped. Each worker class has its own wake event, so a producer wakes only its downstream consumer.

That isolation extends to **startup wiring.** A single connection that fails to build or bind at start (an unresolvable environment value or certificate, an egress refusal, a port already in use, or a refusal to expose cleartext off loopback) is recorded as `failed`, logged, and alerted — and the engine **starts the rest of the graph and serves the API** instead of aborting. A failed outbound still gets its delivery worker, so rows routed to it are retried rather than dropped, and a reload or restart that builds it heals the lane. A failed inbound simply isn't listening. Reload itself stays fail-fast: it validates the whole new registry before quiescing a healthy graph, so only startup degrades gracefully.

Parsing: tolerant-first, strict-on-demand

- **python-hl7** parses tolerantly and powers the field *peek* used for routing. This is the hot path.
- **hl7apy** does version-aware and profile validation, opt-in per inbound connection (`validation.strict`). It is slower and kept off the routing hot path.

Real-world HL7 v2 is frequently non-conformant. Strict-by-default would drop messages that must still route, so tolerance is the default and strictness is opt-in.

Configuration: connections plus code-first routing

The configuration model is a **graph wired by name, authored as Python** — there is no enclosing "channel" object:

- a **Connection** is a named inbound or outbound endpoint (MLLP, file, and so on);
- an inbound Connection names a **Router** — a Python script that sees every received message, decides which Handler(s) to forward to, and may filter;
- a **Handler** is a Python script that filters, transforms, and sends to outbound Connection(s).

Configuration is version-controlled, diff-able Python. The database stores runtime state and messages only — never configuration. (This is the opposite of engines that bury channel definitions as XML inside the database.)

Connections, Routers, and Handlers are authored against the `messagefoundry` surface (`inbound` / `outbound` / `@router` / `@handler` / `Send` / `MLLP` / `File` / `Message`); a directory of such modules loads into a registry that the engine runs.

The configuration graph, described

A typical inbound flow looks like this:

- **inbound** `IB_ACME_ADT` (MLLP) **names a router.**
- The `@router` sees every message, filters, and forwards by name to one or more handlers.
- `@handler` `to_EHR` filters and transforms, then **sends** to outbound `OB_EHR_ADT` (MLLP).
- `@handler` `to_archive` filters and transforms, then **sends** to outbound `OB_ARCHIVE` (File).

Security and PHI

Messages contain PHI, so access control and data protection are first-class concerns. Identity, RBAC, and the action audit log are detailed in the security documentation; data-at-rest, transport, logging, retention, and de-identification are detailed in the PHI documentation. Per-interface trust boundaries and a STRIDE threat model are in the threat model.

Identity and access. The API enforces authentication and role-based access control, deny-by-default. PHI views are gated by distinct permissions (raw view versus summary view). A user-attributed, append-only, hash-chained **audit log** records who viewed, searched, or replayed messages. Local accounts have built-in,

required **multi-factor authentication**; enterprise and Active Directory users get MFA through their own identity provider via SSO federation.

Interface authentication. Connections support **mutual TLS (mTLS)**, **OAuth 2.0 client-credentials**, and **SMART-on-FHIR Backend Services** for authenticating interface traffic.

Data at rest and in transit. Message bodies are encrypted at rest with **AES-256-GCM** through the store cipher when a key is set, with owner-only database and WAL file permissions and volume encryption covering the rest. Transport security (TLS / MLLPS), log redaction, and retention/purge enforcement round out the data-protection controls.

Posture. MessageFoundry has completed an internal **OWASP ASVS 5.0 Level 3 self-assessment** (212 requirements met, 0 failed, 0 partial, 133 not applicable, of 345). This is a self-assessment, not an external audit. ASVS Level 3 recommends — but does not require — an independent code review and penetration test; both are planned. The architecture is designed to **support a HIPAA-compliant deployment**; HIPAA compliance ultimately depends on how an organization deploys and operates the system.

Module map

Package / module	Responsibility
`messagefoundry.config`	Connector models and code-first wiring registry / loader, plus service settings.
`messagefoundry.parsing`	Tolerant peek (python-hl7) and strict validation (hl7apy); the parse tree and the `Message` transform model; and pure non-HL7 codecs — X12 EDI, FHIR, DICOM (headers / SR, over pydicom), and a base64 binary-carriage codec.
`messagefoundry.store`	Durable message store / **staged queue** (one `queue` table keyed by stage), SQLite WAL by default; every receipt is logged with a disposition that flows with the message. A `Store` protocol and an `open_store` factory abstract the backend; production PostgreSQL and SQL Server backends are at parity.
`messagefoundry.transports`	Inbound and outbound connections (MLLP, file, X12, FHIR, DICOM C-STORE SCP inbound over pynetdicom, and more), resolved through a registry — never special-cased in the pipeline.
`messagefoundry.anon`	Deterministic, secret-per-dataset pseudonymization / de-identification (fail-closed), exposed to capture tooling and the test harness.
`messagefoundry.pipeline`	Per-message routing and handling (the registry runner) plus per-inbound-connection supervision; an offline `dryrun` mode.
`messagefoundry.api`	The FastAPI surface for clients — the engine's only external interface.
`messagefoundry.auth`	Authentication + RBAC core (no FastAPI): permissions and roles, identity, password hashing, opaque session tokens, MFA, and LDAP / Active Directory integration — enforced by `api`.
`messagefoundry.generators`	Conformant synthetic HL7 generators (ADT, ORM, ORU, ...) behind `messagefoundry generate`.
`messagefoundry.console`	The PySide6 admin app — a separate process and an HTTP client to the API only.
CLI entrypoint and gate	`serve` / `generate` / `check`, plus the `check` commit / CI gate.

The dependency direction is one-way: `pipeline` / `transports` / `parsing` / `store` / `config` never import `api` or `console`. The API depends on the engine; the console depends on the API.

Beyond the engine packages (separate components, not part of `messagefoundry.*`): the code-first configuration an operator authors (their config modules plus per-environment value overrides), the VS Code IDE extension, the test harness, the sample configuration and message corpus, the Windows service scripts, and the CI / supply-chain workflows.

Dependencies

Runtime and build dependencies are declared in `pyproject.toml` (the source of truth) as `>=` minimums, with a pinned, hashed resolution in committed lockfiles. MessageFoundry requires **Python 3.11+**.

Runtime

- `hl7` (python-hl7) — fast, tolerant parsing for the routing hot path
- `hl7apy` — version-aware validation and profiles (opt-in strict path)
- `pydantic` — configuration / connector models and validation
- `aiosqlite` — async SQLite for the message store / queue
- `fastapi` + `uvicorn[standard]` — the engine API
- `argon2-cffi` — argon2id password hashing
- `cryptography` — AES-256-GCM at-rest encryption for the store
- `ldap3` + `pyspnego` — Active Directory / LDAP authentication and Windows SSO

Optional extras

- `console` → `PySide6` (LGPL — chosen so the open-source console is distributable)
- `postgres` / `sqlserver` → production database backends at parity with SQLite (the SQL Server driver also needs the OS-level Microsoft ODBC Driver 18, which is not pip-installable; both are lazy-imported so SQLite-only installs skip them)
- `dicom` → `pydicom` + `pynetdicom` (the DICOM codec — headers / SR only, no numpy — and the C-STORE SCP inbound connector; lazy-imported so non-DICOM installs skip it)
- `dev` → `pytest`, `pytest-asyncio`, `httpx` (ASGI test client for the API), `ruff`, `mypy`

Build and tooling — `hatchling` (build backend), Ruff (format and lint), mypy (strict), pytest.
