System Requirements & Sizing

These are the minimum and recommended requirements for running the MessageFoundry engine, its message store, and the administration clients. The engine is a headless Python/asyncio service; the console is a separate desktop application.

> **On the throughput figures.** MessageFoundry does not publish a validated, guaranteed per-node throughput baseline. The sizing tiers in Sizing by message volume are **engineering estimates** derived from the architecture and from synthetic load-test profiles — they are starting points, not guarantees. Always establish your own baseline on production-like hardware before go-live (see Capacity notes).

Hardware

	Minimum (lab / low-volume pilot)	Recommended (single-node production)
CPU	2 cores	4+ cores (transform throughput is per-core; one worker set per connection)
Memory	4 GB	8–16 GB
Disk	10 GB free, any disk	**SSD**, 50+ GB or sized to your retention window, on a low-latency local volume
Store volume	—	Put the message store on a fast local disk (not a network share). Budget for **store + WAL growth**: the staged pipeline writes roughly 3× per message on the embedded store.

> The engine and store may share a host for a low-volume pilot. For production, run a **server database** (PostgreSQL or SQL Server) on its own host, sized by your DBA, and keep the engine host dedicated. For volume beyond one CPU core, see Sizing by message volume.

Operating systems

Platform	Status
Windows Server 2022 / 2025	Primary supported and serviced platform (Windows-service deployment)
Windows Server 2019	Supported
Windows 10 / 11	Supported (development, pilot, console host)
Linux (modern x86-64 distributions)	Engine supported (cross-platform Python); no bundled service installer — run under systemd yourself
macOS	Development / console use only

Runtime

Component	Requirement
Python	**3.11 or later**, 64-bit (3.11, 3.12, 3.13, 3.14 supported; 3.11 and 3.14 are CI-validated)
Service manager (Windows)	A Windows service wrapper, auto-provisioned and SHA-256-pinned by the installer, or pre-staged. Requires administrator / elevation to register the service.
C compiler	Not required for the default install (runtime dependencies ship as wheels)

Databases (message store)

Database	Status	Driver / prerequisite
SQLite (WAL)	Default, bundled — single-node	None (`aiosqlite` , in-process)
PostgreSQL 13+	Production	`messagefoundry[postgres]` extra (`asyncpg` , pure-Python — no OS dependency)
Microsoft SQL Server 2019 / 2022	Production	`messagefoundry[sqlserver]` extra (`aioodbc`) **plus the OS-level Microsoft ODBC Driver 18 for SQL Server**. Read-Committed Snapshot Isolation (RCSI) recommended.
MySQL / Oracle	Not supported	—

> The embedded SQLite store needs no setup and suits pilots and single-node deployments. A **server database is greenfield-only** — there is no in-place migration from a populated SQLite store; drain and cut over. The database tier owns its own backup, HA, and (SQL Server) TDE / purge maintenance. A server database is also the **concurrency / scale substrate** — see below.

Administration clients

Client	Requirement
Desktop console	PySide6 (Qt) desktop application — install with the `console` extra. Runs on Windows, Linux, or macOS as a separate process; connects to the engine over the HTTP/WebSocket API. **Not browser-based.**
VS Code extension	Visual Studio Code (current stable) — route wizard, validate-on-save, test bench, stage → promote.
Web browser	Not required to operate the engine (no web admin UI; the API is HTTP/JSON for tooling).

Network & ports

Purpose	Default	Notes
Engine API (HTTP + WebSocket)	`127.0.0.1:8765`	**Loopback by default**, authentication required. To reach the API from another host, front it with a **TLS-terminating reverse proxy** or tunnel — exposing the listener off loopback should always be done over TLS.
Inbound MLLP / TCP listeners	operator-defined (samples use e.g. `2575`, `2600`)	Open to sending systems via firewall. Keep MLLP on a **trusted network segment**.
Outbound	as configured	Reachability to downstream partners and, for server databases, to the database host.
Installer egress	HTTPS	Outbound access for the service installer to fetch the pinned service-wrapper binary (or pre-stage it).

> **Interface authentication.** Beyond network-segment controls, interfaces can authenticate with **mutual TLS (mTLS)**, **OAuth 2.0 client-credentials**, and **SMART-on-FHIR Backend Services**. Administrative access uses required multi-factor authentication for local accounts; enterprise / Active Directory users authenticate (and get MFA) through their own identity provider via SSO federation.

Sizing by message volume

> **Engineering estimate — not a validated benchmark.** These tiers project the architecture's behavior; they are not committed numbers. Throughput depends heavily on **transform cost per message** (the dominant factor), message size, fan-out, and strict-validation use. **Measure your own feeds** with the load harness before committing (see Capacity notes).

How throughput is bounded (read this first)

A single engine process runs **all** message work — decode → peek → route → transform → re-encode — on **one CPU core** (one asyncio event loop; the GIL prevents pure-Python parallelism across threads). So per-process throughput is governed, in order, by:

1. **Transform cost per message** — usually the binding constraint. As an illustrative estimate, a comparable vendor benchmark shows real transformation cutting pass-through throughput by roughly 60% (≈1000 msg/s → ≈400 msg/s). A light / pass-through feed sits near the top of a tier; a heavy transform sits near the bottom.
2. **Durable-write cost** — every stage handoff (ingress → routed → outbound → delivered) is a committed transaction. In-process **SQLite is fastest per write**; a **server database is slower per single write** (network + MVCC) but is the concurrency substrate (next point).

To exceed one core, scale **intra-node** on a server database: many connections / lanes / delivery workers draining **one shared server database** (PostgreSQL or SQL Server) concurrently via `SELECT ... FOR UPDATE SKIP LOCKED` plus row leases. Throughput scales with workers until the **database's commit capacity** is the wall. (SQLite is single-writer and does **not** scale this way — it is the single-process / single-node store.) Engine HA is **single-leader active-passive** — the graph runs on the leader only. A **multi-process, sharded-by-inbound** scale-out (multiple engines, each owning a disjoint set of inbounds, on the shared database) is a **future direction, not a current capability**.

Tiers

Tier	Peak sustained (est.)	Indicative daily volume	Deployment shape	Store	Suggested hardware (engine host)
Pilot / light	up to ~50 msg/s	up to ~1–4 M/day	1 process, single node	SQLite	2 cores / 4 GB
Standard single-node	~50–200 msg/s	~4–15 M/day	1 process, single node	SQLite, or PostgreSQL / SQL Server	4 cores / 8 GB
High single-node	~200–500 msg/s	~15–40 M/day	1 process, tuned (lean transforms; finite-retry on hot lanes)	Server database recommended (PostgreSQL / SQL Server)	4–8 cores / 16 GB
High single-node, concurrent	up to ~500 – low-thousands msg/s	up to ~40 M+/day	1 process, many connections / lanes draining concurrently via `SKIP LOCKED`	**PostgreSQL / SQL Server** (required — not SQLite)	8+ cores / 32 GB + a dedicated database host sized to the commit load

Reading the tiers

- *Peak sustained* is a **per-second** capacity estimate. Healthcare feeds are bursty; real **average** rate (and therefore daily volume) is typically a fraction of peak, so the *indicative daily volume* columns assume a

realistic duty cycle, not `peak × 86,400`.

- The **single-stream / single-core ceiling** is roughly the "High single-node" row — a few hundred msg/s with real transforms, approaching ~1000 msg/s only for light / pass-through work. Past that on one feed you are over one core's budget.
- The **estimated maximum as currently architected** is the **"High single-node, concurrent"** row: one engine process, many connections / lanes draining the shared server database concurrently via `SKIP LOCKED`, bounded by the database's commit ceiling. There is no fixed published cap — on a well-provisioned box with a tuned server database this lands in the **low thousands of msg/s**, beyond which you are **database-bound** and scale the database tier. (A multi-process, sharded-by-inbound scale-out beyond one engine is a **future direction**.) Group-commit and a lazy MSH-only routing peek are identified levers to raise the per-core ceiling.

> **Single-stream server-DB caveat.** Because each staged handoff is a committed round-trip, a single delivery worker against a *remote* server database drains far slower than in-process SQLite (a SQL Server CI smoke profile observes roughly 30 deliveries/s for one stream). High volume on a server database comes from **concurrency** — many connections / lanes / processes draining in parallel — not single-stream speed. Size the database host for that concurrent commit load.

Capacity notes

- Validate with the load harness: run a `smoke` → `fanout-baseline` → `soak` ramp, exercise the `cheap` / `edit` / `slow` transform modes to find your per-core transform ceiling, and compare SQLite vs a server database on identical traffic. Treat the **zero-loss reconciliation** as the headline gate — throughput is meaningless if messages were lost.
- The embedded store has roughly 3× write amplification and a single-writer ceiling; move to PostgreSQL or SQL Server when that becomes the bottleneck.
- Scale **intra-node** on a server database (one delivery worker per outbound; many connections / lanes draining concurrently via `SKIP LOCKED`; keep retry policies finite where head-of-line blocking on a shared FIFO lane would otherwise stall a lane). A multi-process scale-out beyond one engine is a **future direction**. Engine **HA** is **active-passive failover** (opt-in leader / standby cluster on shared PostgreSQL — see CLUSTERING.md); delegate **database-tier** HA to the database plus a load-balancer VIP.

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