Service configuration & settings

> **Status: first cut implemented; the rest is the target.** The `ServiceSettings` model + loader (config/settings.py) and the **CLI > env > file > default** precedence are built and wired into `serve`. Implemented sections: `[store]` (`backend`, `path`, `synchronous`), `[api]` (`host`, `port`), `[inbound]` (`bind_host`), `[delivery]` (`retry_*` + `ordering` — the default retry policy and queue ordering an outbound inherits when it declares none), `[environments]` (`dir`; active env = `[ai].environment`), `[logging]` (`level` + structured-JSON `format` + off-box `forward_*` syslog shipping), `[auth]` (authentication + RBAC), and `[ai]` (AI-assistance policy) — plus `--service-config`. `[retention]` is now enforced (retention/purge + SQLite maintenance), except its `audit_days` key, which is **reserved/keep-forever by design**. Other catalog entries (`[delivery].outbox_workers` / `dead_letter`, some server-DB `[store]` keys) are **accepted-but-ignored** in a config file today so a forward-looking file still loads; build them incrementally.

Principle — two kinds of configuration

MessageFoundry deliberately separates them:

1. **The message graph is code-first.** Connections / Routers / Handlers are authored as Python (config/wiring.py) and loaded from `--config`. This never becomes a settings file — no YAML, no declarative channel config.
2. **Service/operational settings are deployment config**, not code: where the store lives and its credentials, the API bind address, logging, retention, retry defaults, etc. These are what this document covers. They're set by whoever *operates* the service (ops/admin), not by the interface author, and must keep **secrets out of source control**.

Mechanism (proposed)

A single `messagefoundry.toml` (TOML — consistent with `pyproject.toml`; **not** YAML, and not channel config) with one section per group, plus **environment-variable overrides** for secrets, plus **CLI flags** for the common knobs. Precedence (highest first):

```
CLI flag  >  environment variable  >  messagefoundry.toml  >  built-in default
```

- File location: `./messagefoundry.toml` by default, or `--service-config <path>`.
- **Secrets** (e.g. a DB password) should come from **env** (or a secret reference), never plaintext in the file — env wins over the file so a deployment can inject them.
- Env naming: `MEFOR_<SECTION>_<KEY>` (e.g. `MEFOR_STORE_PASSWORD`, `MEFOR_API_PORT`).

- Loaded once at startup into a typed `ServiceSettings` (pydantic) model; the engine + store read from it. `serve` keeps its existing flags as the CLI layer.

Settings catalog

`[store]` — message store / DB

The keys are **implemented in `StoreSettings`**; the SQLite backend is wired, and the SQL Server backend that consumes the server-DB keys lands incrementally (the settings + validation exist now). | Key | Type | Default | Notes | |---|---|---|---| | `backend` | enum | `sqlite` | `sqlite` · `sqlserver` · (later `postgres` / `mysql` / `oracle`) | | `path` | str | `./messagefoundry.db` | SQLite only | | `synchronous` | enum | `normal` | SQLite: `normal` / `full` | | `encryption_key` | secret | — | **env only** (`MEFOR_STORE_ENCRYPTION_KEY`); base64 32-byte **active** key — when set, PHI columns (`raw` / `payload` + `error` / `last_error` / `detail`) are AES-256-GCM-encrypted at rest. Mint one with `messagefoundry gen-key`. Empty = off. See PHI.md §3. | | `encryption_keys_retired` | secret | — | **env only** (`MEFOR_STORE_ENCRYPTION_KEYS_RETIRED`); comma-separated base64 **decrypt-only** keys kept available during a rotation until `messagefoundry rotate-key` finishes re-encrypting under the active key (ASVS 11.2.2). | | `key_provider` | enum | `auto` | selects **how** the active/retired DEK bytes are *sourced* — never how they are used (the cipher, keyring, and `mfenc:v1` format are unchanged; ADR 0019, ASVS 13.3.3). `auto` (default) is the env-then-DPAPI ladder, **byte-identical** to the pre-seam behavior; `env` / `dpapi` pin a single built-in source; `aws_kms` · `azure_kv` · `gcp_kms` · `vault` · `pkcs11` envelope-decrypt a wrapped DEK inside an HSM/KMS/Vault (lazy **optional extras — not built yet**; selecting one **fails closed** at `serve`, never a silent downgrade). Names a *provider*, not key material, so it is **not** a secret. | | `require_encryption` | bool | `false` | when `true`, `serve` **refuses to start** without an encryption key (any environment). Off by default; with it off, a **PHI-carrying** instance (`[ai].data_class = phi`) still gets a loud startup warning. | | `server`, `port` | str/int | — / 1433 | server DBs (required for `sqlserver`) | | `database` | str | — | server DBs (required for `sqlserver`) | | `auth` | enum | `sql` | `sql` · `integrated` · `entra` (SQL Server) | | `username` | str | — | server DBs (required when `auth = sql`) | | `password` | secret | — | **env only** (`MEFOR_STORE_PASSWORD`) | | `encrypt`, `trust_server_certificate` | bool | `true` / `false` | TLS to the DB | | `pool_size` | int | 5 | server DBs | | `connect_timeout`, `command_timeout` | int (s) | 15 / 30 | server DBs | | `db_schema`, `application_name` | str | — / `messagefoundry` | optional (`db_schema` ⇒ env `MEFOR_STORE_DB_SCHEMA`) |

> Selecting `backend = "sqlserver"` validates that `server` / `database` (and `username` when `auth = "sql"`) are present. The backend is **production** (full staged pipeline, response capture, at-rest encryption): it needs the `sqlserver` extra (`pip install 'messagefoundry[sqlserver]'`) plus the Microsoft ODBC Driver 18, and is exercised against a real SQL Server by the CI service-container job. SQLite remains the zero-dependency default.

`[api]`

Key	Type	Default	Notes
`host`	str	`127.0.0.1`	localhost by default. A **non-loopback** bind requires TLS — either in-process (`tls_cert_file` , below) **or** an upstream terminator (`tls_terminated_upstream` + `trusted_proxies`). Without either it is **refused** unless `serve --allow-insecure-bind` is passed (a dev override; bearer tokens + PHI would cross the network in cleartext). With `[auth] enabled = false` a non-loopback bind is refused **unconditionally** (nothing relaxes it).
`port`	int	8765	
`expose_docs`	bool	`false`	serve `/docs` , `/redoc` , `/openapi.json` (off by default — widens surface)
`config_reload_roots`	list[str]	`[]`	extra directories `POST /config/reload` may load from, besides the startup `--config` dir. The loader **executes Python** from these, so list only admin-owned, trusted roots (e.g. an IDE staging dir). Any reload path outside the startup dir + these roots is rejected (403).
`tls_cert_file`	str	*unset*	**`[BUILT]` (WP-13a, ADR 0002):** PEM server-certificate path. **Setting it turns on in-process TLS** — the API serves `https` / `wss` , HSTS engages, and a non-loopback bind is allowed without `--allow-insecure-bind` .
`tls_key_file`	str	*unset*	PEM private-key path (omit if the key is in the cert PEM). Requires `tls_cert_file` .
`tls_key_password`	secret	*unset*	passphrase for an encrypted key — **env only** (`MEFOR_API_TLS_KEY_PASSWORD`), never the file.
`tls_min_version`	str	`1.2`	minimum negotiated TLS version floor (NIST SP 800-52r2): `1.2` or `1.3` .
`tls_ciphers`	str	*unset*	optional OpenSSL cipher string (default = the interpreter's secure defaults).
`tls_client_ca_file`	str	*unset*	CA bundle to **require + verify client certs** (opt-in mTLS, e.g. the console). Requires `tls_cert_file` .
`trusted_proxies`	list[str]	`[]`	**`[BUILT]` (WP-15):** reverse-proxy IP(s) whose `X-Forwarded-For` / `-Proto` are trusted (uvicorn `forwarded_allow_ips`), so the audit/rate-limit source IP is the **real client**, not the proxy. **Empty = trust nothing** (the direct TCP peer is used). Set ONLY to the proxy's address(es), or XFF spoofing returns.

Key	Type	Default	Notes
`tls_terminated_upstream`	bool	`false`	**`[BUILT]` (WP-15):** declare that a reverse proxy / load balancer terminates TLS in front of the engine. Lets a non-loopback bind satisfy the TLS gate **without** in-process TLS — but only when `trusted_proxies` is set (else refused at load).

> **MLLP-over-TLS** is built too (WP-13b — per-connection `tls` / `tls_*` on the `MLLP(...)` connector, see CONNECTIONS.md), and the §0 **exposed-gate is enforced**: a non-loopback *plaintext* MLLP listener is refused at startup unless `serve --allow-insecure-bind`. Gate #4's transport-TLS subset is complete, and **native TOTP MFA (WP-14) is also built** (`[auth].require_mfa`, local accounts). See ADR 0002.

`[inbound]` — inbound listener defaults

Key	Type	Default	Notes
`bind_host`	str	`127.0.0.1`	the **default** network interface every inbound MLLP/TCP listener binds to. Authors never set a `host` on an inbound connection (a wiring error if they do) — it's a per-environment operator decision here. Binding `0.0.0.0` exposes unauthenticated MLLP to the network, so it's deliberate (DEV typically loopback, PROD a specific NIC behind a firewall). A non-loopback bind **requires** `tls=true` on each MLLP connection (the §0 exposed-gate refuses a plaintext off-loopback listener at startup) unless `serve --allow-insecure-bind` is passed. A single connection may override this with a per-connection `bind_address` (and restrict peers with `source_ip_allowlist`) — MLLP/TCP only; see CONNECTIONS.md.

`[environments]` — per-environment graph values (DEV/PROD)

The **same** code-first graph runs in every environment; only the values it references via `env("key")` differ. The **active** environment is the single cross-cutting selector `[ai].environment` — a **free-form name** (ADR 0017), set in the TOML or via `serve --env <name>`, and **required** (no default); this section only locates the value files.

Key	Type	Default	Notes
`dir`	str	`environments`	directory holding `<env>.toml` flat key→value tables for non-secret values, **versioned** in the repo. Resolved against `base_dir` (below).
`base_dir`	str	`""` (= the working dir)	**Anchor** `dir` resolves against. Empty keeps the original behavior (relative to the process working directory). Set it to the **config-repo root** so env-value resolution no longer depends on where `serve` was launched. A relative value is taken against the working dir; an absolute value is used as-is — **on Windows it must be drive-qualified** (`C:/repo`); a leading-slash `/repo` is drive-relative and still inherits the launch drive (logged as a warning). Overridable per run with `serve --project-root`.

- A graph value that differs by environment is authored as `env("acme_adt_host")`; the running instance resolves it from `<base_dir>/<dir>/<active-env>.toml` overlaid by `MEFOR_VALUE_<KEY>` env vars (secrets — never the file; env wins). Keys are `lower_snake_case`.
- **Anchoring the value files (`base_dir` / `--project-root`).** A standalone **config repo** (ADR 0017) keeps `environments/` at its root — a *sibling* of the `--config` dir. With the default (empty) `base_dir`, the files resolve relative to the **process working directory**, so a `serve` launched from anywhere but the repo root reads **no** env values (a silent empty table, not an error — the missing values then fail loud only when a connector is built). This bites most under **NSSM**, whose working directory is rarely the repo. Pin the anchor so resolution is launch-independent — in the instance's `messagefoundry.toml`: `toml [environments] base_dir = "C:/srv/acme-config" # the config-repo root; environments/<env>.toml live under it` or per run: `messagefoundry serve --config config --env prod --project-root C:/srv/acme-config` (the flag overrides `[environments].base_dir`; precedence is CLI > env > file > default, like every service setting). The startup log prints the **resolved** `environments/<env>.toml` path so you can confirm where values are read from. Running from the repo root keeps working unchanged (the empty default is the working dir).
- A referenced key that is **undefined for the target environment** makes the engine refuse to load or promote that graph (fail loud) — never a silent blank host. See the env files under `environments/` and `samples/config/IB_ACME_ADT.py` for a worked example.
- **Per-face logic inside a transform:** `env()` is a *deferred reference* resolved only when a **connection** spec is built — using it in a handler is an always-truthy object (a bug). To branch a Router/Handler on the deployment, read the active environment **name** with `current_environment()` (the free-form name, e.g. `"prod"` / `"test"`, or `None` in a dry-run): `python from messagefoundry import current_environment # Corepoint: If ActiveFace="Test" Then MSH-11.1 = "T" if current_environment() in ("staging", "dev"): msg.set("MSH-11.1", "T")` The active environment is a deployment constant, so the read is pure + re-run-safe.

Code sets — reference lookup tables (`codesets/`)

A code-first Router/Handler often needs a **reference table** — an Epic diet code → a food-service system value, a facility code → a downstream mnemonic. Rather than a hand-maintained Python dict, drop the table in a **code set** and look it up with `code_set("name")`.

- **Where.** Files live in `codesets/` **relative to the `--config` dir** — a config bundle carries its own reference tables and they **reload with the graph** (POST `/config/reload`). This is distinct from `environments/` (cwd-level endpoint values for `env()`). A missing `codesets/` dir is fine (no code sets). The code-set **name** is the file's stem (`codesets/epic_diets.csv` → `"epic_diets"`).
- **CSV** (`<name>.csv`) — a header row; the **first column is the lookup key**. One other column → the value is that scalar (`str`); several other columns → the value is a `dict` `{header: cell}`. A duplicate key is a **load error** (fail loud).
- **TOML** (`<name>.toml`) — a flat table `key = value` → `{key: scalar}`; a nested `[key]` table → `{key: {…}}` (mirrors the `environments/<env>.toml` shape).

- **Usage.** Capture once at a module's top level (preferred) or look it up at call time inside a handler — both resolve: ```python from messagefoundry import code_set, handler, Send

DIET = code_set("epic_diets") # frozen, read-only mapping; captured at import

@handler("to_cbord") def handle(msg): msg["ODS-3"] = DIET.get(msg["ODS-3"], "") # .get(key, default) — blank on a miss fac = code_set("facility_mnemonics").get(msg["MSH-4"]) # call-time lookup also works ... return Send("OB_CBORD_DIET", msg) `` A CodeSet `is a read-only` Mapping `:` cs[key] `(raises` KeyError `naming the set on a miss),` cs.get(key, default) `,` key in cs `,` len(cs) `, iteration. It is **frozen** — one instance is shared across transforms, so a handler must never mutate the reference data. - **Fail loud.**` code_set("missing") `(no such file) or a malformed/duplicate-key CSV/TOML raises a` WiringError `, surfaced by` validate `/` messagefoundry check `/ reload exactly like a missing` env()` value — never a silent empty table. - **Purity caveat.** The lookup is pure (key in → value out), so it's compatible with the staged pipeline's **pure-re-run** invariant (ADR 0001 / CLAUDE.md §2). The one caveat: a hot-reload that **changes** a table between a run and a crash-re-run can make the re-run derive a different output. That's acceptable for reference data (a code set is deliberately operator-editable, and a reload is an explicit, audited act), but it is the one way a transform's re-run can legitimately differ — note it where you document the transform.

Transform state — cross-message correlation (ADR 0005)

Where code sets are **read-only** reference data, **transform state** is **read/write** correlation data a Handler accumulates across messages: an anonymous-patient mapping (persist a real MRN → a stable anonymized id and reuse it on later messages), order↔result correlation, running aggregates. It is authored against two surfaces from `messagefoundry`:

```
from messagefoundry import handler, Send, SetState, state_get

@handler("anonymize")
def anonymize(msg):
    mrn = msg["PID-3.1"]
    anon = state_get("patient_anon", mrn)          # synchronous read; None on a miss
    ops = []
    if anon is None:
        anon = derive_anon_id(mrn)                  # deterministic derivation preferred (see belov
        ops.append(SetState("patient_anon", mrn, anon))
    msg["PID-3.1"] = anon
    return [Send("OB_DOWNSTREAM", msg), *ops]       # Sends and SetStates, mixed in one list
```

- **Write contract — declared, never imperative.** A Handler returns `Send | SetState | list[Send | SetState] | None`; it does **not** mutate state directly. Each `SetState(namespace, key, value)` (the `value` must be JSON-serializable — validated at construction) is an **upsert by** `(namespace, key)` the engine applies **inside the routed→outbound handoff transaction**. `Send`-only Handlers are unchanged — fully **backward compatible**.

- **Exactly-once / re-run safety.** Because the write commits in the **same transaction** as the outbound rows, a crash before commit leaves **no** state (atomic with the handoff) and the attempt that commits applies the write **exactly once per message** — this preserves the staged pipeline's **pure-re-run** invariant (ADR 0001 / CLAUDE.md §2). A non-deterministic value (a random anon id) is still safe because only the committed attempt persists, but **prefer a deterministic derivation** where cross-run identity matters.
- **Read — synchronous, read-through cache.** Handlers are pure synchronous functions and a DB read is async, so `state_get(namespace, key, default=None)` reads an in-memory **read-through cache** the engine maintains (loaded at startup, updated as writes commit) and publishes around each router/transform run — exactly how `code_set()` resolves against an active set. A missing key returns `default` (state is sparse, not a referenced table). **Non-linearization caveat:** a read reflects committed state as of its invocation, but is **not** linearized with a concurrent sibling handler's write — fine for read-mostly correlation; a race-sensitive read-modify-write within one namespace needs author care.
- **Encryption at rest.** State values may carry PHI (MRN↔id), so they are AES-256-GCM-encrypted with the store cipher just like `messages.raw`, and covered by key rotation (`messagefoundry rotate-key`).
- **Retention (TTL).** Set `[retention].state_max_age_days` to age out stale entries (a global age purge; per-namespace policy is a follow-up). Off by default = keep forever. The whole-table cache assumes **bounded** state — unbounded estates (every MRN ever seen) are a documented follow-up (ADR 0005).
- **SQL Server.** State writes ride the staged `transform_handoff`, which is implemented on the SQL Server backend, so the `state` table is **live** (parity with SQLite/Postgres); the read-through cache refreshes post-commit. Cross-node state convergence is N/A (single-node backend).

`state_get` also resolves in **dry-run** / the IDE Test Bench / `messagefoundry check`: each simulated message gets a fresh in-memory view that accumulates that run's own declared writes (so a later handler sees an earlier one's `SetState`), and `dryrun` output lists the declared state ops — **PHI-gated** behind `--show-phi` like a message body.

Reference sets — external-data enrichment (ADR 0006)

Where a **code set** is a static lookup table shipped in the bundle and **transform state** is read/write correlation, a **reference set** is **external data materialized off the message path**: a provider directory, a DB-backed translation table (the Corepoint Data Point / DB Association pattern). The engine syncs the source into a **versioned, encrypted store snapshot** on a cadence; a Handler reads it **purely** at run time. Because the read carries no external call, the staged pipeline's pure-re-run invariant holds (the only non-determinism is a snapshot flip landing between a run and a crash-re-run — the same accepted caveat as a code-set hot-reload).

- **Declare** a set in a wiring module (registers it into the graph, like `inbound`): ```python from messagefoundry import Reference, FileRef, env, handler, Send, reference

Reference("provider_npi", source=FileRef(path=env("provider_npi_csv")), refresh_seconds=3600)

@handler("enrich") def enrich(msg): npi = reference("provider_npi").get(msg["PV1-7.1"]) # pure dict lookup, no I/O if npi: msg.set("PV1-7.13", npi) return Send("OB_DOWNSTREAM", msg) `` - ** reference(name) **

returns a frozen, read-only ReferenceSet (rs[k] / rs.get(k, d) / k in rs). A missing **key** returns the default (external data is sparse); a missing/unsynced **set** raises (fail loud) at run time → that message's ERROR disposition. Call it **inside a Handler/Router**, not at module top level (the snapshot exists only once the store is open + synced — unlike code_set). - **Sources:** FileRef(path=..., encoding=...) — a local CSV/TOML in the **code-set format**, re-read on the refresh cadence (the path for an externally-produced export; path may be env()). DatabaseRef(server=..., database=..., statement=..., key_column=..., value_column=...) — the engine runs a read-only SQL query on the cadence (SQL Server via the [sqlserver] extra, **production / supported**; secrets via env() ; the dial-out is gated by the fail-closed [egress].allowed_db allowlist). key_column is the lookup key; value_column (if set) is the value, else the value is a dict of the other columns. - **Sync.** The engine's ReferenceSyncRunner materializes each set once at startup (before listeners serve, so reference(...) resolves on the first message) and every refresh_seconds . A source failure is **isolated**: it's logged + alerted and the **last-good snapshot is kept** (the write isn't attempted), so one bad source never blocks the others or the message path. - **At rest:** snapshot values are AES-GCM-encrypted (they may carry PHI) and covered by key rotation, exactly like state /message bodies; the [egress].allowed_db gate will govern the (increment-2) DB source. **SQLite-only** (the SQL Server store has an inert stub). - **[reference] settings:** refresh_interval_seconds (loop tick, default 3600), sync_on_startup (default true), max_staleness_seconds (reserved, 0 = off). - **Dry-run / check ** resolve file-backed sets best-effort (literal paths) so a reference-using transform validates; DB-backed or env()`-path sets are absent in a pure dry-run.

`[auth]` — authentication & RBAC

Implemented (see SECURITY.md). Authentication is **required** by default; the AD bind password is a **secret** supplied via env (`MEFOR_AUTH_AD_BIND_PASSWORD`), never the file. | Key | Type | Default | Notes | |---|---|---|---| | `enabled` | bool | `true` | required by default; off only for embedding/tests | | `session_idle_timeout_minutes` | int | 30 | idle auto-logoff (inactivity window; background re-checks don't reset it) | | `session_absolute_hours` | int | 12 | absolute session lifetime | | `max_sessions_per_user` | int | 5 | cap concurrent sessions per user (ASVS 7.1.2; `0` = unlimited); a login beyond the cap revokes the user's oldest active session | | `password_min_length` | int | 15 | local-password policy — ASVS 5.0-aligned, length-first | | `password_require_uppercase` / `_lowercase` / `_digit` / `_symbol` | bool | `false` | character classes — **opt-in** (ASVS 5.0 forbids mandatory composition); on only for a legacy standard that still mandates them | | `password_check_breached` | bool | `true` | reject known common/breached passwords against a bundled offline top-10k list (no live HIBP call) | | `password_check_context` | bool | `true` | reject passwords containing app/vendor/HL7 terms (e.g. `messagefoundry` , `mefor` , `hl7` , `corepoint`) | | `lockout_threshold` | int | 5 | failed logins before lock (per account) | | `lockout_minutes` | int | 15 | lockout duration | | `bootstrap_expiry_hours` | int | 72 | the first-run bootstrap admin is auto-disabled once a second administrator exists, and — while still unclaimed (never password-changed) — this many hours after creation. `0` = no time expiry | | `login_rate_limit_enabled` | bool | `true` | in-process sliding-window limiter on `/auth/login` , `/auth/negotiate` , `/me/password` (in front of the per-account lockout) | | `login_rate_limit_per_ip` | int | 10 | max attempts per client IP per window (`0` disables) | |

`login_rate_limit_global` | int | 60 | max attempts across all clients per window (`0` disables) | | `login_rate_limit_window_seconds` | float | 60 | sliding-window length | | `phi_read_rate_limit_enabled` | bool | `true` | per-actor anti-automation throttle on the PHI-read endpoints `/messages` , `/messages/{id}` , `/dead-letters` (ASVS 2.4.1) — bounds scripted PHI harvesting on top of pagination + access auditing | | `phi_read_rate_limit_per_actor` | int | 120 | max PHI reads per user per window (generous — clears console/human use; `0` disables this dimension) | | `phi_read_rate_limit_global` | int | 0 | max PHI reads across all users per window (`0` = off) | | `phi_read_rate_limit_window_seconds` | float | 60 | sliding-window length | | `notify_security_events` | bool | `true` | email the affected user on lockout / first-success-after-failures / password-email-role-disable changes (ASVS 6.3.5/6.3.7). Reuses the `[alerts]` SMTP transport, sent to the user's own address; no SMTP configured → email skipped. The `GET /me/security-events` feed (over the audit log) is always available regardless of this toggle. | | `require_mfa` | bool | `false` | require a native **TOTP** second factor for the **Administrator** role (WP-14, ASVS 6.3.3). Off by default (preserves loopback behavior byte-for-byte); turn on for an off-loopback bind that serves local accounts. **`serve` enforces the posture at exposure:** an **off-loopback** PHI bind with this off **refuses to start** in production and **warns** in a non-production PHI environment (synthetic stays quiet), like the keyless-store / open-egress gates. Safe to enable even AD-only — it gates only local Administrator accounts. Non-admins may opt in by enrolling. AD/Kerberos MFA is delegated to the directory (never an engine TOTP). See SECURITY.md "Multi-factor authentication". | | `mfa_recovery_code_count` | int | 10 | single-use recovery codes minted at TOTP enrollment (the lost-authenticator escape hatch; `0` disables them, leaving an admin reset as the only recovery path). Range 0–50. | | `admin_new_ip_step_up` | bool | `false` | admin-interface contextual-risk signal (WP-L3-13, ASVS 8.4.2): when on, a step-up (sensitive admin) request from a client IP the session has not verified from emits an `auth.admin_action_new_ip` audit + notice and **forces a fresh step-up** (a re-verify from that address clears it). Advisory + step-up-forcing only — never changes an RBAC decision, never blocks the non-admin path; the audit + notice fire once per (session, new address). Off by default (byte-identical on loopback — `127.0.0.1` and `::1` are treated as one host); recommended on for an off-loopback admin deployment. See SECURITY.md "Administrative-interface defense-in-depth". | | `ad_enabled` | bool | `false` | turn on Active Directory login | | `ad_server` | str | — | e.g. `ldaps://dc1.example.com:636` (required when `ad_enabled`) | | `ad_domain` | str | — | UPN suffix, e.g. `example.com` | | `ad_user_search_base` | str | — | required when `ad_enabled` | | `ad_group_search_base` | str | — | base for nested-group resolution | | `ad_bind_dn` | str | — | service-account DN used for lookups | | `ad_bind_password` | secret | — | **env only** (`MEFOR_AUTH_AD_BIND_PASSWORD`) | | `ad_use_nested_groups` | bool | `true` | resolve nested groups (`LDAP_MATCHING_RULE_IN_CHAIN`) | | `ad_tls_verify` | bool | `true` | validate the LDAPS certificate | | `ad_tls_ca_cert_file` | str | — | trust an internal CA for LDAPS without disabling verification | | `ad_allow_insecure_ldap` | bool | `false` | explicit opt-in to a non- `ldaps://` bind (trusted-network dev only) | | `kerberos_enabled` | bool | `false` | Windows SSO (experimental, **0.2 target — not supported in v0.1**; needs `ad_enabled`) | | `kerberos_spn` | str | — | service principal, e.g. `HTTP/host.example.com` |

> AD-group→role mappings live in the DB and are managed by an admin (`PUT /ad-group-map` or the console Users page), not in this file.

`[ai]` — AI coding assistance policy

Implemented (see AI.md). Controls the IDE AI assistant across the **OFF→PHI-safe** range; the policy is centrally governed and **posture-clamped**. `mode` / `data_scope` plus the active environment NAME + posture (`environment` / `data_class` / `production`) are the keys that act in the MVP — the rest are forward-compat placeholders for the future engine broker (accepted-but-ignored today). | Key | Type | Default | Notes | |---|---|---|---| | `mode` | enum | `byo` | `off` · `byo` · `managed_claude` · `managed_claude_baa` (the last two are **future** — not serviceable by the current IDE) | | `data_scope` | enum | `code_only` | `code_only` · `synthetic` · `deidentified` · `phi` , least→most sensitive; capped by `production` posture and by `mode` (only `managed_claude_baa` reaches `phi`) | | `environment` | str | — | free-form active-environment **name** (ADR 0017); selects `environments/<name>.toml` + `current_environment()` . **Required** for `serve` (no default) | | `data_class` | enum | derived | `synthetic` · `phi` — does this instance carry real PHI (drives the at-rest/egress advisories). Derived from a built-in name (`dev` ⇒synthetic, `staging` / `prod` ⇒phi) when unset; **required** for a custom name | | `production` | bool | derived | production-tier posture (drives the `data_scope` ceiling + prod-DEBUG refusal), decoupled from the name. Derived (`dev` / `staging` ⇒false, `prod` ⇒true) when unset; **required** for a custom name | | `provider` | str | `claude` | **forward-compat, unused in MVP** (P1 broker) | | `model` | str | `claude-opus-4-8` | **forward-compat, unused in MVP** | | `baa_attested` | bool | `false` | **forward-compat, unused in MVP** | | `endpoint` | str | — | **forward-compat, unused in MVP** |

> Only `code_only` context is ever sent in the MVP (graph names + active editor code) — **never message bodies**. The full resolution/clamping algorithm, the `GET /ai/policy` endpoint, the `messagefoundry ai-policy` CLI, and the `ai:assist` RBAC permission are documented in AI.md. Env keys: `MEFOR_AI_MODE` , `MEFOR_AI_DATA_SCOPE` , `MEFOR_AI_ENVIRONMENT` , etc.

`[logging]`

Key	Type	Default	Notes
`level`	enum	`info`	log level; never run prod at `debug` (PHI) — `serve` refuses it (Gate #1)
`format`	enum	`text`	stdout rendering: `text` (default) or structured `json` (one object per line). Stdlib only — no structlog
`forward_enabled`	bool	`false`	ship a copy of every record off-box to a syslog/SIEM collector (sec-offbox-log) so evidence survives a host compromise; requires `forward_host`
`forward_host`	str	—	syslog/SIEM collector host (**required** when `forward_enabled`)
`forward_port`	int	`514`	collector port (1–65535)
`forward_protocol`	enum	`udp`	`udp` (fire-and-forget) or `tcp` . The forwarder never blocks the engine indefinitely: a TCP collector down at startup is skipped with a warning, and a runtime stall is bounded by a socket timeout (record dropped). Synchronous send — prefer `udp` /a local agent for high volume
`forward_format`	enum	`json`	wire format sent off-box, independent of stdout `format` . JSON guarantees one record per line; `text` framing is best-effort (multi-line tracebacks span lines)
`file` , `max_bytes` , `backups`	str/int	—	**planned** rotation (NSSM captures stdout today)

> PHI redaction + control-char scrubbing are **always-on handler filters** (not a toggle) applied to **every** sink, including the off-box forwarder; the syslog transport is plaintext, so front it with a local TLS-forwarding agent or a trusted network. See PHI.md §7.

`[retention]`

Enforced by the engine's retention/purge task (pipeline/retention.py). A purge **NULLs the PHI *body*** past its window while **keeping the metadata row** (counts, disposition, and the audit trail stay intact — the Mirth Data-Pruner pattern); it never deletes a `messages` row and never touches a body still in flight. Everything defaults to `0` / `""` = keep/off, so retention is opt-in. | Key | Type | Default | Notes | |---|---|---|---| | `messages_days` | int | `0` | past N days, null inbound bodies (`raw` / `summary` / `error`) of **fully-resolved** messages (no `pending` / `inflight` delivery), keeping metadata. `0` = keep | | `dead_letter_days` | int | `0` | past N days, null the bodies of **dead-lettered** outbound rows (their own window — a dead row stays replayable until purged). `0` = keep | | `state_max_age_days` | int | `0` | past N days, **delete** transform-state entries (ADR 0005) last written before the cutoff — keeps the in-memory state cache + table bounded. A simple global age purge (by `set_at`); per-namespace policy is a follow-up. `0` = keep | | `audit_days` | int | `0` | **reserved / not enforced.** The `audit_log` is a tamper-evident hash chain and HIPAA expects ~6-year retention, so audit is **keep-forever by design**; archive-first pruning is a tracked follow-up. Accepted so a forward-looking file still loads | | `max_db_mb` | int | `0` | advisory only: warn (WARNING log + an `AlertSink`

`storage_threshold` event) when the DB (+ `-wal` / `-shm`) exceeds this. Never auto-deletes. `0` = off | | `purge_interval_seconds` | float | `3600` | how often the purge/maintenance loop runs a pass | | `wal_checkpoint_seconds` | float | `0` | `PRAGMA wal_checkpoint(TRUNCATE)` cadence (SQLite). `0` = off (rely on auto-checkpoint). Evaluated once per pass, so a value below `purge_interval_seconds` is effectively rounded up to it | | `vacuum_at` | str | `""` | daily local `"HH:MM"` to run `VACUUM` (SQLite; reclaims space freed by purges). `""` = off. A daily off-peak time, **not** a cron expression (no new dependency); VACUUM holds a write lock on the whole DB while it runs |

> **SQLite-only.** Retention/maintenance runs on the SQLite backend. On the SQL Server backend it is a DBA concern (TDE + a SQL Agent purge/shrink job). Each pass that does real work writes one `retention_purge` `audit_log` entry (cutoffs + counts, **no** message content). Design: PHI.md §8.

`[delivery]`

Key	Type	Default	Notes
`retry_max_attempts`	int	*unset*	attempts before a delivery dead-letters. **Unset = retry forever** (the conservative default — a transient failure/ `AE` NAK is never silently lost; under FIFO the head blocks its lane until it succeeds or is purged). Set a finite value to opt back into retry-then-dead-letter. A permanent `AR` reject fails fast regardless.
`retry_backoff_seconds`, `retry_backoff_multiplier`, `retry_max_backoff_seconds`	num	5 / 2 / 300	exponential backoff between attempts (per-outbound `retry=` overrides)
`ordering`	enum	`fifo`	default queue ordering per outbound: `fifo` (strict in-order, head-of-line on failure) or `unordered` (batch + rotate-past-failures). Per-outbound `ordering=` overrides.
`internal_error`	enum	`continue`	what a delivery worker does on an **internal/code error** (a non- `DeliveryError` exception from `send` — our bug, not the partner's): `continue` (dead-letter the row + advance) or `stop` (halt the connection's worker, preserve the message for replay, raise a `connection_stopped` alert). Per-outbound `internal_error=` overrides. Partner NAKs / transport failures are unaffected.
`buildup_max_depth`	int	*unset*	raise a `queue_buildup` alert when an outbound lane's pending depth reaches this. Unset = depth dimension off (a healthy ceiling is throughput-specific, so there's no safe default). Per-outbound `buildup=BuildupThreshold(...)` overrides.
`buildup_max_oldest_seconds`	num	300	raise `queue_buildup` when the lane's **oldest** pending message has waited this long (a stuck/retry-forever head is the classic cause). On by default — a head stuck >5 min is a problem in any environment. Set to unset/ `0` -disable via a per-outbound override.
`outbox_workers`	int	per-outbound	delivery concurrency (planned)
`dead_letter`	enum	`keep`	`keep` / `drop` -after-N (planned)

`[pipeline]`

Key	Type	Default	Notes
`max_correlation_depth`	int (≥1)	8	**Re-ingress loop cap** (ADR 0013 Increment 2). When a captured reply is re-ingressed (`reingress_to=` / `Loopback()`), the re-ingressed message carries a `correlation_depth` ; a message at this depth still routes, but the next hop (depth+1) **dead-letters** its re-ingress work-row and marks the origin `ERROR` . Coarse by design — it bounds *total work*, not topology, so a chain that legitimately bounces A→B→A a few times needs headroom. 8 is safe for typical request→response→route feeds; raise it for deep correlation chains, lower it to fence a misbehaving loop. (A value of 0 would dead-letter every re-ingress, so the floor is 1.)

`[egress]`

Fail-closed **outbound destination allowlist** (WP-11c; ASVS 13.2.4/13.2.5/14.2.3) — bounds where the engine may **send** PHI, so a fat-fingered or hostile destination can't exfiltrate it. Each list is **opt-in**: empty = unrestricted (today's behavior); once a transport's list is set, an outbound of that transport not on it is **refused at config load/reload** (a `WiringError` → 422 / refused reload), checked against the resolved (`env()` -substituted) destination.

Key	Type	Default	Notes
`allowed_mllp`	list	`[]`	allowed MLLP destinations; each entry is `host` (any port) or `host:port`. Via env: comma-separated `MEFOR_EGRESS_ALLOWED_MLLP`
`allowed_tcp`	list	`[]`	allowed raw-TCP (`Tcp(...)`) destinations; each entry is `host` (any port) or `host:port`. An inbound `Tcp(...)` is a local listener and is not gated. Via env: comma-separated `MEFOR_EGRESS_ALLOWED_TCP`
`allowed_file_dirs`	list	`[]`	allowed File output directories; a destination's directory must resolve at/under one of these
`allowed_http`	list	`[]`	allowed REST/SOAP (HTTP) destination hosts; each entry is `host` (any port) or `host:port` (ADR 0003). Via env: comma-separated `MEFOR_EGRESS_ALLOWED_HTTP`
`allowed_db`	list	`[]`	allowed DATABASE destination servers; each entry is `host` (any port) or `host:port` (ADR 0003). Via env: comma-separated `MEFOR_EGRESS_ALLOWED_DB`
`allowed_remote`	list	`[]`	allowed RemoteFile (SFTP/FTP/FTPS) hosts — gates the connector in **both** directions (source poll + destination upload); each entry is `host` (any port) or `host:port`. Via env: comma-separated `MEFOR_EGRESS_ALLOWED_REMOTE`
`deny_by_default`	bool	`false`	**fail-closed master switch**: when `true`, a transport with an **empty** allowlist refuses *every* destination of that type (so each permitted destination, dial-out source, and `db_lookup` server must be listed). Default `false` keeps the per-list opt-in above.

> `serve` warns at startup in a `prod` / `staging` environment when egress is fully open (no allowlist set **and** `deny_by_default` off) — a transform could then send PHI anywhere. Lock it down with `deny_by_default = true` or the per-transport lists above.
>
> The webhook/SMTP **alert** sinks carry no message bodies (no PHI) and keep their own host allowlists in `[alerts]` (`webhook_allowed_hosts` / `smtp_allowed_hosts`).

`[shadow]`

Parallel-run / **shadow-instance** egress suppression (#15). A *shadow* MessageFoundry processes real (teed) traffic to validate it against a legacy engine, but must **not** deliver to live partners (the legacy engine is still the real sender). An outbound in **simulate** mode runs the full pipeline + count-and-log and finalizes the message `PROCESSED`, but **suppresses the real egress** (no bytes/SQL leave the box) and retains the would-send payload for parity comparison.

Key	Type	Default	Notes
`simulate_all_egress`	bool	`false`	**deployment-wide master switch**: when `true`, **every** outbound runs egress-suppressed regardless of its own `simulate=` — so a shadow stand-up can't accidentally leave one outbound live. Default `false` = each outbound's own `simulate=` flag applies.

> Per-outbound control is the precise mechanism — set `simulate = true` on an individual outbound (`outbound(..., simulate=True)` or `simulate = true` in `connections.toml`); this section is the blunt instance-wide override. A simulated lane is surfaced as `simulated` on `GET /connections` and shown as `[SIMULATED]` in the console. Simulate suppresses **egress only** — the `[egress]` allowlist, connector construction, and handler state writes are unaffected. With egress suppressed there is no real partner reply, so a **capturing / `reingress_to`** outbound captures (and re-ingresses) **nothing** in simulate — the message just finalizes `PROCESSED`.

`[alerts]`

Where the delivery pipeline's operational alerts (`connection_stopped`, `queue_buildup`) are delivered. **Both transports are off by default** — with neither configured, events are logged at `WARNING` (the `LoggingAlertSink`). A transport turns on when its essentials are present. Payloads carry the connection name + queue shape only — **never a message body** (no PHI). Delivery is best-effort and runs on a background task, so it never blocks or hangs a delivery lane.

Key	Type	Default	Notes
`webhook_url`	str	*unset*	enable the **webhook** transport: HTTP `POST` the event as JSON here (fronts Slack/Teams/PagerDuty/custom inbound webhooks).
`webhook_timeout`	num	10	seconds per POST
`webhook_allowed_hosts`	list	`[]`	egress allowlist for the webhook host (`[]` = any); SSRF defense (ASVS 15.3.2/1.3.6)
`email_smtp_host`	str	*unset*	SMTP server; with `email_from` + `email_to` set, enables the **email** transport
`email_smtp_port`	int	587	SMTP port
`email_from`	str	*unset*	sender address (required for email)
`email_to`	list	*unset*	recipient(s) (required for email). Via env: comma-separated `MEFOR_ALERTS_EMAIL_TO`
`email_use_tls`	bool	`true`	issue STARTTLS before sending
`email_username`	str	*unset*	SMTP login user (omit for unauthenticated relays)
`email_password`	str	*unset*	**secret** — supply via `MEFOR_ALERTS_EMAIL_PASSWORD` , never the file
`email_timeout`	num	30	seconds per send
`smtp_allowed_hosts`	list	`[]`	egress allowlist for the SMTP host (`[]` = any); parity with `webhook_allowed_hosts` (WP-11c)
`realert_seconds`	num	300	suppress re-notifying the same (event, connection) more often than this (anti-spam for a flapping lane). A matching rule's `cooldown_seconds` overrides it.
`rules`	list	`[]`	ordered `[[alerts.rules]]` table array — per-event severity, transport routing, thresholds, suppression, cooldown (see below). Empty = today's behaviour (every event → every transport at `warning`).

`[[alerts.rules]]` — per-event routing (ADR 0014)

Each rule is a row in an **ordered** `[[alerts.rules]]` array; the **first matching rule wins** (so put the most specific rules first). An event matching **no** rule keeps the default: notify **every** configured transport at `warning` with the global `realert_seconds` — so adding a rule never silently silences an event you didn't name. Matching is pure config (no code/ `eval`).

Key	Type	Default	Notes
`event_type`	str	`any`	match this event: `any`, `connection_stopped`, `queue_buildup`, `storage_threshold`, or `cert_expiry`
`connection`	str (glob)	`*`	glob over the connection name (e.g. `OB_*`, `IB_ACME_*`)
`min_depth`	int	*unset*	`queue_buildup` only — match only when pending depth is at/over this
`min_oldest_seconds`	num	*unset*	`queue_buildup` only — …or the oldest pending message has waited at least this long
`severity`	str	`warning`	`info` | `warning` | `critical` — tagged onto the event (webhook JSON + email subject) for downstream triage
`transports`	list	*all*	which transports fire: subset of `["webhook", "email"]`; **unset = all configured**; **`[]` = SUPPRESS** (drop silently)
`cooldown_seconds`	num	*global*	override `realert_seconds` for matching events (e.g. re-page a critical sooner)

```
[alerts]
webhook_url = "https://hooks.example.com/services/XXX"   # webhook transport on
email_smtp_host = "smtp.example.com"                      # email transport on
email_from = "alerts@example.com"
email_to   = ["oncall@example.com"]

# Page (webhook) immediately and re-page every minute when any inbound connection stops.
[[alerts.rules]]
event_type = "connection_stopped"
connection = "IB_*"
severity = "critical"
transports = ["webhook"]
cooldown_seconds = 60

# A deep backlog on any outbound is critical; a shallow one only emails.
[[alerts.rules]]
event_type = "queue_buildup"
min_depth = 1000
severity = "critical"

[[alerts.rules]]
event_type = "queue_buildup"
severity = "info"
transports = ["email"]

# Stay quiet about a known-bursty test feed.
[[alerts.rules]]
connection = "OB_LOADTEST"
transports = []   # suppress every event for this connection
```

> A rule routing to a transport that isn't configured (e.g. `transports = ["email"]` with no SMTP settings) is rejected at startup, so a typo can't silently black-hole an alert. Severity travels in the payload; **timed multi-stage escalation** ("email now, page in 15 min") is future work (ADR 0014 §3) — rules give the static routing primitive it would build on.

`[cert_monitor]`

Periodic TLS-certificate **expiry monitor**. Now that native off-loopback TLS is the supported posture (`DEPLOYMENT.md`), a silently expiring certificate is a hard PHI-feed outage at renewal time. The engine scans the certs it actually serves with — the `[api].tls_cert_file` and every connection's `tls_cert_file` (MLLP server/client identity) — and raises a `cert_expiry` alert (an `[alerts]` event — route it with a `[[alerts.rules]]` rule) when one is expired or within `warn_days` of expiry. Only the **public certificate** is read (its `notAfter`), never a private key. On by default with a 30-day window; set `warn_days = 0` to disable.

Key	Type	Default	Notes
`warn_days`	int	30	alert when a served cert expires within this many days; `0` **disables** the monitor
`check_interval_seconds`	num	43200	rescan cadence (default 12h); the per-cert re-alert throttle is `[alerts].realert_seconds`

```
[cert_monitor]
warn_days = 45            # start warning 45 days out

# Page (don't just email) when a served cert is close to expiry.
[[alerts.rules]]
event_type = "cert_expiry"
severity = "critical"
transports = ["webhook"]
```

`[cluster]` — active-passive HA coordination (Track B)

Server-DB-backed. Introduces the multi-node coordination seam — a `nodes` table, a per-node heartbeat, (Track B Step 4) **leader election**, and (Step 6) **cross-node reference + config-reload convergence** — *without changing single-node behavior*. It runs as **active-passive** HA: one leader runs the whole graph, a standby takes over on failure. (The horizontal **active-active** scale-out path — per-lane ownership running the graph on every node — was **dropped (2026-06-18) and its code removed**; it is not a planned milestone.) With `enabled = false` (the default) the engine uses a no-op coordinator and runs **byte-identically** to before. Enabling it requires a **server-DB** store **and** `[store].pool_size >= 2` — a clustered node drives concurrent background work (the membership/lease-renewal maintenance loop + the per-stage workers) against the pool, so a pool of 1 would serialize everything (prefer `>= 3`). A cross-section validator refuses either violation at config load. Two backends qualify:

- `postgres` — the full coordinator: leader election, the row leases, and the leader reclaim sweep, run as active-passive HA (the leader runs the graph; a standby takes over on failure).
- `sqlserver` — **active-passive too**: the same self-fencing leadership lease (one leader drains the graph; a standby takes over on failure). A single active node (the leader) processes at a time, so the `reclaim_expired_leases` background sweep below applies on Postgres; on-promotion recovery covers both backends.

SQLite remains single-node (cluster coordination is refused on it).

With `[cluster].enabled` on Postgres, **leader election is built** as a **self-fencing lease** (Workstream A2): exactly one node across the cluster holds the `leader_lease` row and is the **leader** — it renews the lease every `heartbeat_seconds` (to `DB_now + leader_lease_ttl_seconds`, on the database's own clock so node clock skew is irrelevant to who may hold it), and a standby acquires only once that lease has **expired**. A leader that cannot renew within `leader_fence_timeout_seconds` (which must be `< leader_lease_ttl_seconds`) **self-fences** — it stops acting as leader *before* the lease can expire and a

standby acquire it, so a network-partitioned old leader never double-processes (the split-brain guard). The leader-only **WRITE singletons** run on that one node while followers **no-op** them (reactive-by-polling, so failover is automatic on the next tick): - `[retention]` **purge/VACUUM/audit** — runs on the leader only. - **the lease-reclaim sweep** — the leader periodically calls `reclaim_expired_leases` (cadence `reclaim_interval_seconds`) to recover **crashed** nodes' in-flight rows (only rows whose lease has *expired*, never a live sibling's). In clustered mode the engine therefore **skips** the single-node unconditional `reset_stale_inflight` startup recovery, which would steal a live sibling's in-flight rows.

Poll-source intake is leader-gated (Track B Step 4b). A **poll** source — `file` (a watched directory), `database` (a polled table), `remote-file` (an SFTP/FTP directory) — reads a **shared external resource**: if more than one node polled it, the same file/row would be ingested twice. So only the **leader** polls a poll source. Under active-passive HA the whole graph — **listen** sources (`mllp`, `tcp`) and all the **staged-queue workers** (router / transform / delivery) alike — runs on the **leader only**; a standby binds no listeners and runs no workers (so poll-source gating is belt-and-suspenders, and the queue's `FOR UPDATE SKIP LOCKED` + row leases serve intra-node concurrency and failover recovery rather than concurrent multi-node draining). The brief overlap during a leadership transition (the old leader's last in-flight poll vs. the new leader's first) is bounded by the same at-least-once guarantees that cover a crash mid-poll — the file-rename / row-claim atomicity and the downstream queue's idempotent handoff make a re-read a tolerated duplicate, never data loss. The worst-case transition window is bounded by the lease timing: a partitioned leader keeps polling until it self-fences (within `leader_fence_timeout_seconds`), and a standby cannot take over until the lease expires (`leader_lease_ttl_seconds`) — and `fence < TTL` guarantees the old leader has stopped first. For a `database` source the row-claim atomicity is the operator's `poll_statement` / `mark_statement` (claim with a status flag or `UPDATE ... RETURNING`); the engine owns the atomic rename only for file sources.

If the leader stops cleanly it expires its lease so a follower acquires leadership at once; if it crashes or is partitioned, the lease ages out and a follower acquires after at most `leader_lease_ttl_seconds`. **Single-node operation is unchanged** (the no-op coordinator is always leader, so every poll source always scans, runs the unconditional startup reset, and spawns no leader sweep).

Per-lane FIFO survives failover. Because the graph runs on the **leader only**, per-lane FIFO is naturally serialized by that single processor — there is no concurrent multi-node draining of a lane to reorder. Across a failover the order still has to be preserved for a lane whose head was in flight on the crashed/fenced prior leader: the ordinary FIFO claim (`claim_next_fifo`) reclaims that **stranded head** — this lane's expired-lease inflight row, back to pending **in the same transaction, before the head SELECT** — so the recovered head blocks the lane rather than being skipped, and a later row can never deliver ahead of it (the recovery does not wait on the leader's periodic sweep). This **replaced** the dropped active-active per-lane lease mechanism (the removed `lane_leases` table / per-lane ownership). The wall-clock row lease carries the NTP assumption: keep `[store].lease_ttl_seconds` comfortably above clock skew + the claim cadence. **Single-node is byte-identical** (the no-op coordinator is always leader); SQLite and SQL Server behave the same single-active-processor way.

Cross-node convergence is built (Track B Step 6). Two shared-state concerns now converge across nodes automatically: - **Reference sets** — materialize-from-source is **leader-gated** (only the leader re-reads

the external file/DB source and writes the shared, versioned snapshot), and **every** node then **read-throughs** that snapshot into its own in-process read cache via the store's `converge_reference_cache` (matching on the per-set version). So the external source is read **once** per cluster and no follower is left on a stale cache — replacing the prior "every node re-syncs" model. Single-node is byte-identical: the no-op coordinator is always leader (materializes every pass) and the convergence call is a no-op on SQLite (the sole writer's cache is always current). - **Config reload** — an operator `POST /config/reload` on **one** node bumps a single-row `cluster_config` **version token**; every **other** node's config-convergence loop observes the higher version and reloads **its own** (identically-deployed) config dir to converge. The initiating node advances its applied version when it bumps, so it does **not** re-reload (no feedback loop). A `dry_run` never bumps; single-node never spawns the loop. This assumes **homogeneous config** across nodes (the token coordinates *when* to reload; each node reloads its own dir) — the same assumption as the dead-letter-missing-destinations/handlers startup sweeps.

> The coordination seam is **built**: leader election (self-fencing lease), leader-gated singletons + poll-source intake, failover-safe per-lane FIFO (the stranded-head reclaim above), cross-node reference + config convergence, transform-STATE cross-node read-through (Step 6b), and the read-only `/cluster` ops API (Step 7) — a one-time startup `INFO` summarizes the operational assumptions. This is the supported **active-passive** HA model (one leader drains the graph; a standby takes over on failure), on **Postgres or SQL Server**. The horizontal **active-active** scale-out path (many nodes processing concurrently) was **dropped (2026-06-18) and its code removed** — it is not a planned milestone. On both backends, failover recovers the prior leader's in-flight rows on promotion, safe because the old leader self-fences before its lease expires.

Key	Type	Default	Notes
`enabled`	bool	`false`	turn on the coordination seam; requires a server-DB store (`[store].backend` = `postgres` or `sqlserver`) and `[store].pool_size >= 2`
`node_id`	str	*unset*	override the auto id (`host:pid:hex`); pin for a stable identity / tests. Unset → reuses the store's lease owner-id, so node-id == owner-id
`heartbeat_seconds`	num	10	how often a node refreshes its `last_seen` heartbeat **and** renews its leadership lease (no separate leader-check knob). Must be > 0
`node_timeout_seconds`	num	30	a node is considered dead when its `last_seen` is older than this (the `/cluster/nodes` freshness filter). The leadership **lease** — not this timeout — is what transfers leadership. Must be > 0, and must exceed `heartbeat_seconds`
`reclaim_interval_seconds`	num	30	how often the **leader** runs the lease-reclaim sweep that recovers crashed nodes' in-flight rows (followers no-op). Must be > 0
`leader_lease_ttl_seconds`	num	30	the leadership lease TTL (active-passive self-fencing). The leader renews to `DB_now + this` ; a standby acquires only once the lease has expired (on the DB clock, so node skew is irrelevant). Must be > 0
`leader_fence_timeout_seconds`	num	20	a leader that can't renew within this (its own monotonic clock, no DB I/O) self-fences — the split-brain guard. Must be > 0, `> heartbeat_seconds` , and `< leader_lease_ttl_seconds`

`[approvals]`

Optional **dual-control (maker-checker)** approval for high-value actions (ASVS 2.3.5) — see SECURITY.md. **Off by default**; turning it on holds the listed operations for a *distinct* second approver holding `approvals:approve` , with the requester unable to approve their own.

Key	Type	Default	Notes
`enabled`	bool	`false`	turn on dual-control; off = every action executes inline as before
`operations`	list[str]	`["connection_purge", "dead_letter_replay"]`	which operations require approval; each must be a known op key (a typo is refused at startup)
`expiry_hours`	num	72	a pending request can no longer be approved after this many hours (`0` = never expires)

`[engine]`

Key	Type	Default	Notes
`shutdown_timeout_seconds`	int	30	graceful stop
`data_dir`	str	`.`	base for relative paths

`[service]` (NSSM / Windows)

Mostly lives in `scripts/service/` today: service name, auto-restart, stdout/stderr log paths.

`[security]`

Authentication & RBAC are configured in `[auth]` above (see SECURITY.md). `encryption_at_rest` — **future**. The planned approach (AES-GCM through the store's `_encode` / `_decode` seam for SQLite, plus required volume encryption; SQL Server TDE on that backend) is documented in PHI.md.

Example

```
# messagefoundry.toml
[store]
backend = "sqlserver"
server = "sql01.hospital.local"
database = "MessageFoundry"
auth = "sql"
username = "mefor_service"
encrypt = true

[api]
host = "127.0.0.1"
port = 8765

[logging]
level = "info"
format = "json"                       # structured stdout (one JSON object per line)
forward_enabled = true                # ship a copy off-box to a syslog/SIEM collector
forward_host = "siem.hospital.local"
forward_port = 514
forward_protocol = "tcp"              # udp (default) | tcp

[retention]
messages_days = 30      # null inbound bodies after 30 days, keep metadata
dead_letter_days = 90   # null dead-letter bodies after 90 days
vacuum_at = "03:30"     # daily off-peak VACUUM to reclaim space
```

```
# secret via env (never in the file)
set MEFOR_STORE_PASSWORD=...
```

Build order (incremental)

1. ✅ **Done** — `ServiceSettings` model + loader (file + env + CLI precedence); `[api]` / `[logging]` and `[store] backend=sqlite|path|synchronous` wired into `serve` (`--service-config` + the `--db` / `--host` / `--port` / `--log-level` overrides).
2. `[delivery]` defaults → feed the default `RetryPolicy`.
3. `[store]` server-DB keys land **with** the SQL Server backend.
4. ✅ **Done** — `[retention]` purge/maintenance job (body-null + WAL/VACUUM, audited; `audit_days` reserved). `[logging]` structured-JSON `format` + off-box `forward_*` syslog shipping land (sec-offbox-log); PHI redaction is an always-on handler filter (no structlog).

Open decisions (to confirm)

- **TOML file + env + CLI** as above — or env-only / all-CLI? (TOML chosen for consistency with `pyproject.toml` and ops-friendliness; secrets via env.)
- Where settings are **edited from** — Console (operational) and/or a read-only view in the IDE.
- Whether per-connection overrides (e.g. a connection's own retry) stay in code (today) or also move into settings. Recommendation: **keep per-connection logic in code**, service settings are defaults.