Installing MessageFoundry — the engine + your configuration repository

This guide explains how an organization installs **MessageFoundry (MEFOR)** and stands up the **private git repository** that holds its integration configuration. It is the practical companion to ADR 0017 (the consumer deployment model).

> **Scope.** This guide covers *install + config-repo + private git + running multiple instances*. For the staged, go/no-go-gated path from a first install to full production (lab → shadow → limited → full), plus security/PHI hardening, reliability tuning, and DR, read the **Early-Adopter Installation & Rollout Guide**. For network exposure / TLS specifics see **DEPLOYMENT.md**; for the Windows service see **SERVICE.md**.

1. The model in one paragraph

MessageFoundry ships as a **read-only, version-pinned engine** (a Python wheel) that your team installs but **never edits**. All of *your* work — the Connections, Routers, and Handlers that define your interfaces — lives in a **separate, private git repository your organization owns**: your *config repo*. One config repo drives **all** of your engine instances (Test and Production at minimum; optionally POC/Staging), with each instance selecting its environment and security posture at runtime. The practical consequence: integration developers author and review interface logic in a normal git/PR workflow, and the engine underneath them is a fixed, auditable dependency they cannot quietly modify.

Three ownership tiers — keeping them separate is the whole design:

Tier	What it is	Who owns / edits it
Engine	The installed `messagefoundry` wheel (pinned version)	The MessageFoundry project — you install it, never edit it
Config repo	Your private git repo: Connection/Router/Handler `.py`, code sets, `environments/<env>.toml`, fixtures	Your integration developers — authored via pull requests
Per-instance settings	`messagefoundry.toml` + `MEFOR_*` environment variables + secrets	Your operators, per deployed instance — **never** committed to the repo

2. Prerequisites

- **Python 3.11+** on each engine host (the engine targets 3.11+).
- **git**, plus a **private git host** — GitHub (private repo), GitLab, Azure DevOps, Bitbucket, or a self-hosted server. Nothing about MessageFoundry requires a public repo; your config repo is yours.
- A source for the engine wheel: **public PyPI** is the distribution channel (`pip install messagefoundry`); an **internal package index** (Artifactory, Azure Artifacts, a private PyPI) can mirror the signed wheel for air-gapped or offline sites.
- Administrator/elevation on the host if you will install the engine as a Windows service (see SERVICE.md).

---

3. Step 1 — Install the engine (a pinned, read-only dependency)

On each host, create a virtual environment and install the engine at a **pinned version**:

```
python -m venv .venv
.\.venv\Scripts\Activate.ps1                # Linux/macOS: . .venv/bin/activate
pip install "messagefoundry==0.2.0rc1"          # pin the exact version (core runtime only)
```

> ⚠️ **Early access.** `0.2.0rc1` is an **Early Access** release on public PyPI — feature-complete and test-validated, but the external review + pen test that gate a security-certified **v1.0** land after launch. The exact-pin command above (`==0.2.0rc1`) resolves today; the earlier `0.2.0rc1` pre-release also remains installable (`pip install messagefoundry==0.2.0rc1`). You can equally install from the engine's **GitHub Release assets** or your organization's **private index**.

Add extras only for what a host actually runs — `messagefoundry[postgres]` (PostgreSQL store), `messagefoundry[sqlserver]` (SQL Server store + the DATABASE connectors, needs OS-level ODBC Driver 18), `messagefoundry[console]` (desktop admin console), `messagefoundry[sftp]` (SFTP connectors).

Key properties this install model gives you:

- **Non-editable.** A normal `pip install` lands the engine in `site-packages` as a regular installed package — not an editable checkout. Developers import its public surface; they do not have the engine source tree in front of them to change.
- **Pinned + reproducible.** The exact version is recorded in your config repo's `requirements.txt` (Step 2). An engine upgrade is a deliberate, reviewable one-line bump — never an accident. For a fully reproducible deploy, extend that into a hash-locked requirements file and install with `--require-hashes` .

Verify the release before you install (supply-chain integrity)

MessageFoundry ships **one signed wheel to many PHI-bearing instances**, so verify the artifact's provenance *before* installing it — pinning a version (or a hash) proves you got a *fixed* file, not that it's the one MessageFoundry built. Every release carries **SLSA build provenance** (binding the wheel's SHA-256 →

the source commit → the GitHub Actions builder) and a **Sigstore signature**. Check both with the **GitHub CLI** (`gh` ≥ 2.49), and optionally `sigstore` (`pip install sigstore`); install **only** the file that passes.

```
$V = "0.2.0rc1"   # the exact version you intend to install

# Download the wheel + its Sigstore bundle from that release's assets
gh release download "v$V" --repo MEFORORG/MessageFoundry `
  --pattern "messagefoundry-$V-*.whl" --pattern "messagefoundry-$V-*.whl.sigstore*"

# Verify SLSA build provenance:  artifact -> source commit -> builder workflow
gh attestation verify "messagefoundry-$V-py3-none-any.whl" --repo MEFORORG/MessageFoundry

# (defense in depth) Verify the Sigstore signature pins the release workflow identity
python -m sigstore verify identity "messagefoundry-$V-py3-none-any.whl" `
  --cert-identity "https://github.com/MEFORORG/MessageFoundry/.github/workflows/release.yml@refs/ta
  --cert-oidc-issuer "https://token.actions.githubusercontent.com"

# Only if BOTH pass, install the exact file you verified
pip install ".\messagefoundry-$V-py3-none-any.whl"
```

The same attestation also covers the **public PyPI** copy of the wheel (it is byte-identical to the GitHub-built artifact, so the digest matches), so you can download-verify-then-install from the index:

```
$V = "0.2.0rc1"
pip download "messagefoundry==$V" --no-deps -d .\verify
gh attestation verify (Get-ChildItem ".\verify\messagefoundry-$V-*.whl").FullName --repo MEFORORG/M
pip install --no-index --find-links .\verify "messagefoundry==$V"
```

A registry/mirror substitution or a relabelled file **fails** the check. For a fully pinned deploy, pair this with `pip install --require-hashes -r requirements.lock` (the identity check above is the part `--require-hashes` cannot give you — it proves bytes-match-lockfile, not who built them). The `-rc` -tagged pre-releases publish to production PyPI too; the `--cert-identity` ref above must match the tag you are installing (e.g. `refs/tags/v0.2.0rc1`).

---

4. Step 2 — Create your config repo with the scaffolder

The engine ships an `init` command that lays down a complete, **`check` -green-from-the-first-commit** config repo skeleton:

```
messagefoundry init ./my-config-repo
cd my-config-repo
```

It writes:

```
my-config-repo/
├─ config/IB_EXAMPLE_ADT.py        # a runnable starter feed (receive ADT over MLLP → archive to fi
├─ environments/dev.toml           # non-secret per-environment values for env("…") lookups
├─ environments/prod.toml
├─ messages/sets/example_adt.hl7   # a synthetic fixture (NO real PHI) that gates `check`
├─ messagefoundry.toml             # THIS instance's settings (environment + posture + store + API
├─ requirements.txt                # pins the engine:  messagefoundry==0.2.0rc1
├─ .github/workflows/check.yml     # CI: install the pinned engine + run `messagefoundry check` on
├─ .vscode/settings.json           # points the VS Code extension at config/ + messages/
├─ .gitignore  .gitattributes      # excludes stores, secrets, captures, venvs, caches
└─ README.md
```

Validate it immediately:

```
pip install -r requirements.txt
messagefoundry check --config config --messages messages/sets
```

`check` runs validate + dry-run against the synthetic fixture — the same gate your CI runs on every pull request. From here, your developers replace the starter feed with real Connections/Routers/Handlers, authored against the engine's public surface (`inbound` / `outbound` / `@router` / `@handler` / `Send` / `Message` / `MLLP` / `File` / `env` / `code_set` …). They never touch engine code to do it.

---

5. Step 3 — Make it a *private* git repo

The scaffolded directory is ready to become your private repo:

```
git init
git add .
git commit -m "Initial MessageFoundry config (scaffolded)"
# then point it at your private host, e.g.:
git remote add origin git@github.com:your-org/mefor-config.git   # a PRIVATE repo
git push -u origin main
```

What makes this private and safe:

- **Repo visibility.** Create the remote as **Private** on whatever host you standardize on. The engine imposes no requirement here — your config repo is yours, access-controlled by your git host's permissions (teams, SSO, branch protection).
- **No secrets, ever.** The scaffolded `.gitignore` already excludes `*.db`, `.env`, captures, and build cruft. **Secrets — DB passwords, keys, credentials — are injected at runtime via `MEFOR_*` environment**

variables, never written to the repo. The versioned `environments/<env>.toml` files hold only **non-secret** endpoints (hostnames, ports), which is exactly what you want diffable and reviewable.

- **No PHI in the repo.** Test fixtures are **synthetic only**. Real message bodies live in the engine's secured store at runtime, not in git.
- **CI as the gate.** The included `check.yml` installs the pinned engine and runs `messagefoundry check` on every PR. Turn on **branch protection** on `main` so changes land only through reviewed, check-passing pull requests — interface logic gets the same rigor as application code.
- **Branch → PR → review → merge** is the entire authoring workflow. There is no separate "engine change" path for developers, because there is no engine to change.

---

6. Step 4 — Configure each instance (settings + posture + secrets)

Each deployed instance gets its own `messagefoundry.toml` (the scaffolder generates a starting one) plus its `MEFOR_*` environment. Two things every instance must state:

- **`[ai].environment`** — a free-form name (`test`, `prod`, `poc`, …) that selects `environments/<name>.toml`.
- **Security posture, explicit and decoupled from the name:** `[ai].data_class` (`synthetic` | `phi` — does this instance carry *real* PHI?) and `[ai].production` (is this a production tier?). Built-in names `dev` / `staging` / `prod` derive a sensible default posture; **any custom name must state posture explicitly** — the engine fails closed rather than guess.

Secrets and host-specific overrides come from the environment, e.g. `MEFOR_VALUE_<KEY>` for values used by `env("…")` in the graph, and `MEFOR_<SECTION>_<KEY>` for service settings. Precedence is **CLI flag > `MEFOR_*` env > `messagefoundry.toml` > built-in default**. Full reference: CONFIGURATION.md.

---

7. Step 5 — Run / deploy an instance

```
messagefoundry serve --config config --env test --project-root C:\srv\mefor\my-config-repo
```

- **`--project-root`** (or `[environments].base_dir` in `messagefoundry.toml`) anchors `environments/<env>.toml` resolution to the repo root, so values resolve **regardless of the working directory**. This matters when the engine runs as a **Windows service under NSSM**, where the launch directory isn't your repo. (Omit it only when you always launch from the repo root.)
- The engine **binds `127.0.0.1` by default** and **requires authentication**. To expose a channel off-loopback, configure **native TLS** (API: `[api].tls_cert_file` / `tls_key_file` or a trusted upstream terminator; MLLP inbound: per-connection `tls = true`) — a non-loopback bind without TLS is **refused at startup**. See DEPLOYMENT.md.
- For production, run the engine as a **Windows service via NSSM** — see SERVICE.md.

Launching the admin console (no command line)

Operators monitor and run an instance from the **PySide6 admin console** — a separate desktop app that talks to the engine over its localhost API. Install the console extra, then drop a Desktop / Start-Menu shortcut so it opens with a double-click, no terminal:

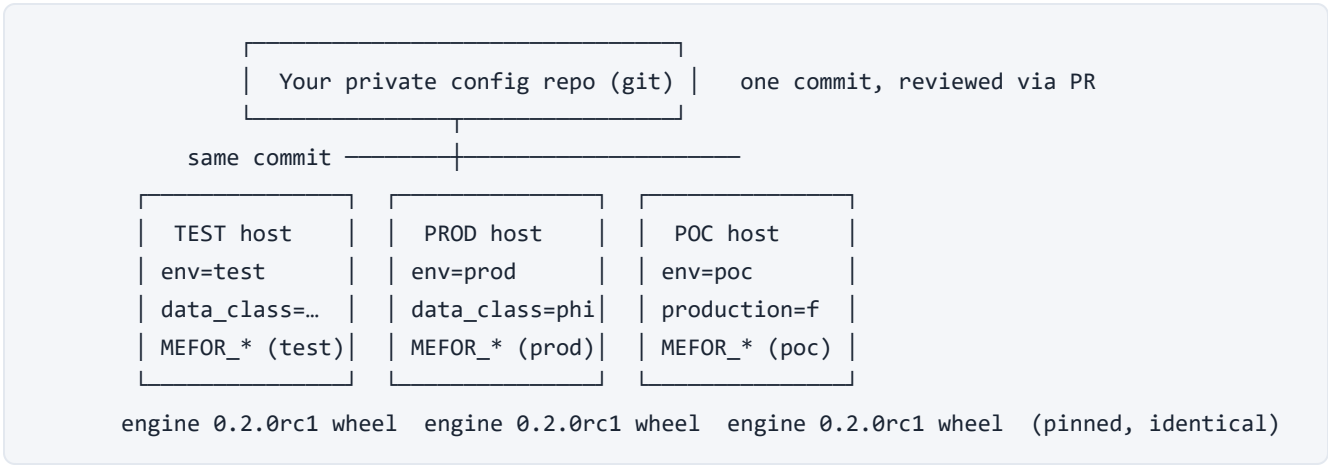
```
pip install "messagefoundry[console]==0.2.0rc1"     # PySide6 + keyring, into the same venv
.\scripts\console\install-console-shortcut.ps1   # per-user icon (add -AllUsers, elevated, for mach
```

The shortcut launches `messagefoundry-console.exe` — a **windowed** launcher (no flashing console window) carrying the MessageFoundry badge. Double-click it: the console connects to the engine (`http://127.0.0.1:8765` by default — typically the boot-start service) and prompts for sign-in, so nothing else is needed for the local case. Point it at a different engine with `-Url https://engine.internal:8765` (off-loopback requires TLS), and remove the icons with `uninstall-console-shortcut.ps1`. (A shell still works too: `messagefoundry-console` or `python -m messagefoundry.console`.)

---

8. Multiple instances from one repo (Test, Production, POC…)

This is the payoff of the model: **the same reviewed commit of your config repo is deployed to every instance.** You do **not** maintain per-environment branches. Each host differs only in its `messagefoundry.toml` (environment name + posture) and its `MEFOR_*` environment:



Promotion = merge to `main` → deploy that commit everywhere. A Test instance resolves `environments/test.toml`; Prod resolves `environments/prod.toml`; a Test instance can never accidentally read Prod values. (Each instance is otherwise an independent engine with its own store; for multi-node high availability of a single instance, see CLUSTERING.md.)

---

9. Why integration developers can't modify the engine

Three layers reinforce the boundary:

1. **Packaging.** The engine is an installed, **non-editable** wheel. Developers work exclusively in the config repo; the engine source isn't in their working tree.
2. **Version pinning.** `requirements.txt` pins an exact engine version; any change to it is a visible, reviewable PR — and CI re-runs `check` against the new version before it can merge.
3. **(Optional) OS enforcement.** Operators can make the venv's `site-packages` read-only (Windows ACL) so the installed engine can't be edited in place even by mistake.

There is simply no developer workflow that routes through engine source — by construction.

---

10. Upgrading the engine

1. Bump the pin in `requirements.txt` (e.g. `messagefoundry==0.2.0rc1`) on a branch.
2. `pip install -r requirements.txt` and run `messagefoundry check` locally; open a PR — CI re-validates your whole config against the new engine.
3. Merge, and roll the new commit to Test first, then Production. Because everything is pinned and your config is gated by `check`, upgrades are deliberate and reversible (pin back). The full upgrade/rollback runbook is in EARLY-ADOPTER-GUIDE.md §13.

Where to go next

Topic	Reference
Staged install-to-production rollout, hardening, DR	EARLY-ADOPTER-GUIDE.md
The consumer deployment model (rationale)	ADR 0017
Service settings / environments	CONFIGURATION.md
Connections / the graph / `connections.toml`	CONNECTIONS.md
Windows service install	SERVICE.md
Network exposure / TLS	DEPLOYMENT.md
Security / auth / RBAC / audit	SECURITY.md
PHI handling / encryption	PHI.md
Multi-node high availability	CLUSTERING.md

Bottom line

Install the pinned engine into a venv; run `messagefoundry init` to scaffold a private config repo; push it to your private git host with branch protection + the included CI check; deploy as many instances as you need

from that one repo, each with its own environment name, posture, and secrets. Your developers get a clean, reviewable, code-first authoring experience, and the engine stays a fixed, auditable dependency they never edit.
