```python
# src/mantra/embeddings/config.py
from __future__ import annotations

from mantra.config import EmbeddingConfig

__all__ = [
    "EmbeddingConfig",
]
```

```python
# src/mantra/embeddings/hvg_embed.py
from __future__ import annotations

from typing import Tuple

import numpy as np
import scanpy as sc
import scipy.sparse as sp

from mantra.config import EmbeddingConfig

# Optional PHATE
try:
    import phate  # type: ignore
except ImportError:
    phate = None  # type: ignore


def _to_dense(x):
    """Convert sparse -> dense; ensure numpy array."""
    if sp.issparse(x):
        return x.toarray()
    return np.asarray(x)


def _ensure_hvg_view(
    ad: sc.AnnData,
    use_hvg_only: bool,
    hvg_key: str,
) -> sc.AnnData:
    """
    Return an AnnData subset to HVGs (if annotated), otherwise a full copy.

    All embeddings are computed on this HVG view, but stored in the
    original AnnData (ad.obsm[...] on the full object).
    """
    if use_hvg_only and hvg_key in ad.var:
        mask = ad.var[hvg_key].to_numpy().astype(bool)
        n_hvg = int(mask.sum())
        if n_hvg > 0:
            print(f"[HVG] Using HVGs only via '{hvg_key}': n_vars = {n_hvg}", flush=True)
            return ad[:, mask].copy()
        else:
            print(
                f"[HVG] '{hvg_key}' present but no genes flagged; "
                "using all genes."
            )
            return ad.copy()
    else:
        if use_hvg_only:
            print(
                f"[HVG] use_hvg_only=True but '{hvg_key}' not in ad.var; using all genes.",
                flush=True,
            )
        else:
            print("[HVG] use_hvg_only=False; using all genes.", flush=True)
        return ad.copy()


def _add_hvg_truncated(
    ad: sc.AnnData,
    ad_hvg: sc.AnnData,
    n_genes: int,
    key: str = "X_hvg_trunc",
) -> None:
    """
    'HVG truncated' embedding: keep top genes by normalized dispersion
    (or first n) and store raw expression for those genes in .obsm[key].
```

```
    Shape: (n_cells, n_genes)
    """
    print(f"[HVG-trunc] Computing HVG-truncated embedding (n_genes={n_genes})", flush=True)

    if "dispersions_norm" in ad_hvg.var:
        disp = ad_hvg.var["dispersions_norm"].to_numpy()
        order = np.argsort(disp)[::-1]  # descending
        keep_idx = order[: min(n_genes, ad_hvg.n_vars)]
    else:
        print(
            "  No 'dispersions_norm' in ad.var; taking first "
            f"{min(n_genes, ad_hvg.n_vars)} HVGs."
        )
        keep_idx = np.arange(min(n_genes, ad_hvg.n_vars))

    X_hvg = ad_hvg.X[:, keep_idx]
    X_hvg = _to_dense(X_hvg).astype(np.float32)

    ad.obsm[key] = X_hvg
    ad.uns[f"{key}_genes"] = ad_hvg.var_names[keep_idx].tolist()
    print(f"  Stored '{key}' with shape {X_hvg.shape}.", flush=True)


def _add_pca_scanpy(
    ad: sc.AnnData,
    ad_hvg: sc.AnnData,
    n_components: int,
    seed: int,
) -> None:
    """
    PCA via Scanpy (ARPACK-based on covariance). Stores ad.obsm['X_pca'].
    """
    print(f"[PCA] Computing Scanpy PCA with n_comps={n_components}", flush=True)
    sc.tl.pca(
        ad_hvg,
        n_comps=n_components,
        svd_solver="arpack",
        random_state=seed,
    )
    X_pca = ad_hvg.obsm["X_pca"][:, :n_components].astype(np.float32)
    ad.obsm["X_pca"] = X_pca
    print("  Stored 'X_pca' with shape", X_pca.shape, flush=True)


def _add_neighbors_diffmap_umap(
    ad: sc.AnnData,
    ad_hvg: sc.AnnData,
    n_components: int,
    n_neighbors: int,
    seed: int,
    run_diffmap: bool,
    run_umap: bool,
) -> None:
    """
    Build kNN graph on PCA space (from ad_hvg), then optionally:
      - Diffusion Map â\206\222 ad.obsm['X_diffmap']
      - UMAP          â\206\222 ad.obsm['X_umap']
    """
    if not (run_diffmap or run_umap):
        return

    print(
        f"[neighbors] neighbors: n_neighbors={n_neighbors}, "
        f"based on X_pca; n_comps={n_components}",
        flush=True,
    )
```

```
    sc.pp.neighbors(
        ad_hvg,
        n_neighbors=n_neighbors,
        use_rep="X_pca",
        random_state=seed,
    )

    if run_diffmap:
        print("  Computing Diffusion Map...", flush=True)
        sc.tl.diffmap(ad_hvg, n_comps=n_components)
        X_dm = ad_hvg.obsm["X_diffmap"][:, :n_components].astype(np.float32)
        ad.obsm["X_diffmap"] = X_dm
        print("  Stored 'X_diffmap' with shape", X_dm.shape, flush=True)

    if run_umap:
        print("  Computing UMAP...", flush=True)
        sc.tl.umap(
            ad_hvg,
            n_components=n_components,
            random_state=seed,
        )
        X_umap = ad_hvg.obsm["X_umap"].astype(np.float32)
        ad.obsm["X_umap"] = X_umap
        print("  Stored 'X_umap' with shape", X_umap.shape, flush=True)


def _add_phate(
    ad: sc.AnnData,
    ad_hvg: sc.AnnData,
    n_components: int,
    n_neighbors: int,
    seed: int,
) -> None:
    """
    PHATE embedding stored as ad.obsm['X_phate'], if phate is installed.
    """
    if phate is None:
        print("[PHATE] phate not installed; skipping.", flush=True)
        return

    print(
        f"[PHATE] Computing PHATE (n_components={n_components}, knn={n_neighbors})",
        flush=True,
    )
    X = _to_dense(ad_hvg.X)
    ph = phate.PHATE(
        n_components=n_components,
        knn=n_neighbors,
        n_jobs=-1,
        random_state=seed,
    )
    X_phate = ph.fit_transform(X).astype(np.float32)

    ad.obsm["X_phate"] = X_phate
    print("  Stored 'X_phate' with shape", X_phate.shape, flush=True)


def compute_embeddings(ad: sc.AnnData, cfg: EmbeddingConfig) -> sc.AnnData:
    """
    Main library entry point.

    Mutates `ad` in-place by adding embeddings into .obsm and returns it.
    Uses the EmbeddingConfig from mantra.config.
    """
    np.random.seed(cfg.seed)

    # Work on HVG subset for manifold methods
    ad_hvg = _ensure_hvg_view(
```

```
        ad,
        use_hvg_only=cfg.use_hvg_only,
        hvg_key=cfg.hvg_key,
    )

    # 1) HVG truncated
    if cfg.run_hvg_trunc:
        hvg_trunc_n = cfg.hvg_trunc or cfg.n_components
        _add_hvg_truncated(ad, ad_hvg, n_genes=hvg_trunc_n, key="X_hvg_trunc")

    if cfg.only_hvg_phate:
        # Only HVG trunc + PHATE
        if cfg.run_phate:
            _add_phate(
                ad,
                ad_hvg,
                n_components=cfg.n_components,
                n_neighbors=cfg.n_neighbors,
                seed=cfg.seed,
            )
    else:
        # 2) PCA
        if cfg.run_pca:
            _add_pca_scanpy(ad, ad_hvg, n_components=cfg.n_components, seed=cfg.seed)

        # 3) Diffusion Map + UMAP
        _add_neighbors_diffmap_umap(
            ad,
            ad_hvg,
            n_components=cfg.n_components,
            n_neighbors=cfg.n_neighbors,
            seed=cfg.seed,
            run_diffmap=cfg.run_diffmap,
            run_umap=cfg.run_umap,
        )

        # 4) PHATE
        if cfg.run_phate:
            _add_phate(
                ad,
                ad_hvg,
                n_components=cfg.n_components,
                n_neighbors=cfg.n_neighbors,
                seed=cfg.seed,
            )

    return ad
```

```python
# src/mantra/embeddings/__init__.py
from __future__ import annotations

from .config import EmbeddingConfig
from .hvg_embed import compute_embeddings

__all__ = [
    "EmbeddingConfig",
    "compute_embeddings",
]
```