tuning the toolkit’s intake valves like a Stradivarius, coming right up. here’s the retrofit that gives you:

- an incremental directory cache (skip re-ingesting files that haven’t changed),
- a tiny HDF5 walker that dives one level into stubborn group layouts, and
- a --merge-policy switch to resolve “eigenpairs vs operator-derived eigenpairs” when shapes fight.

everything below drops cleanly into the structure we already set up.

---

1) Incremental cache over directories

new file:

willowlab/ingest/cache_dir.py

```
# willowlab/ingest/cache_dir.py
import os, json, hashlib, pathlib, time
from typing import Dict, Any, Optional

DEFAULT_CACHE_NAME = "ingest_cache.json"

def _hash_file(path: str, block_size: int = 1024 * 1024) -> str:
    """SHA256 of file contents, streamed (works for big blobs)."""
    h = hashlib.sha256()
    with open(path, "rb") as f:
        while True:
            chunk = f.read(block_size)
            if not chunk: break
            h.update(chunk)
    return h.hexdigest()


def _mtime(path: str) -> float:
    try:
        return os.path.getmtime(path)
    except OSError:
        return 0.0


class DirCache:
    """
    Directory-level cache:
      - remembers file SHA256 and size/mtime
      - provides 'should_process' decision
      - records arbitrary ingest metadata (e.g., where we saved the Willow bundle)
    """
    def __init__(self, root_dir: str, cache_path: Optional[str] = None):
        self.root_dir = os.path.abspath(root_dir)
        self.cache_path = cache_path or os.path.join(self.root_dir, DEFAULT_CACHE_NAME)
        self._db: Dict[str, Any] = {}
        if os.path.exists(self.cache_path):
            try:
                self._db = json.load(open(self.cache_path, "r"))
            except Exception:
                self._db = {}

    def file_record(self, path: str) -> Optional[Dict[str, Any]]:
        key = os.path.relpath(os.path.abspath(path), self.root_dir)
        return self._db.get(key)

    def should_process(self, path: str) -> bool:
        """Skip if same hash & size persisted; true means reprocess."""
        abspath = os.path.abspath(path)
        key = os.path.relpath(abspath, self.root_dir)
        size = os.path.getsize(abspath)
        mtime = _mtime(abspath)
        rec = self._db.get(key)
        if not rec:
            return True
        # Fast path: if size or mtime changed, re-hash and reprocess.
        if rec.get("size") != size or abs(rec.get("mtime", 0.0) - mtime) > 1e-6:
            return True
        # No change in size/mtime -> trust it as unchanged (cheap). If you want belt+braces, re-hash here.
        return False

    def update(self, path: str, **ingest_meta):
        abspath = os.path.abspath(path)
        key = os.path.relpath(abspath, self.root_dir)
        size = os.path.getsize(abspath)
        mtime = _mtime(abspath)
        # Hash only when we actually processed; keeps idle runs fast.
        sha = _hash_file(abspath)
        self._db[key] = {"size": size, "mtime": mtime, "sha256": sha, "ingest_meta": ingest_meta, "updated": time.time()}


    def persist(self):
        pathlib.Path(self.cache_path).parent.mkdir(parents=True, exist_ok=True)
        with open(self.cache_path, "w") as f:
            json.dump(self._db, f, indent=2, sort_keys=True)
```

what it does: keeps a JSON database per ingestion root. on each run we check file size/mtime; if they match, we skip. when we do process, we compute and store the SHA256 to make the record deterministic and transportable.

---

2) Tiny HDF5 walker (one level down)

modify

willowlab/ingest/file_index.py

Replace the HDF5 branch with a one-level walker that samples datasets at root and one-group deep.

```
# ... keep imports as before ...
def _sample_hdf5_obj(obj, budget_items: int = 24, budget_elems: int = 200_000):
    """
    Build a small dict of datasets from HDF5 'obj' scanning root and one level deep.
    We cap items and elements to avoid RAM blowups.
    """
    out = {}
    def _take(name, dset):
        try:
            if hasattr(dset, "shape"):
                # respect budget
                numel = 1
                for s in getattr(dset, "shape", ()):
                    numel *= max(int(s), 1)
                if numel > budget_elems:
                    # take a small head slice along first axis (if any)
                    sl = (slice(0, min(8, dset.shape[0])),) + tuple(slice(None) for _ in
range(len(dset.shape)-1))
                    out[name] = dset[sl]
                else:
                    out[name] = dset[()]
                return True
        except Exception:
            return False
        return False

    # root datasets
    count = 0
    for k, v in obj.items():
        if count >= budget_items: break
        if hasattr(v, "shape"):
            if _take(k, v): count += 1

    # one level deep: groups under root
    for k, v in obj.items():
        if count >= budget_items: break
        if hasattr(v, "items"):  # group
            for k2, v2 in v.items():
                if count >= budget_items: break
                full = f"{k}/{k2}"
                if hasattr(v2, "shape"):
                    if _take(full, v2): count += 1
    return out

def index_willow_dir(data_dir, name_filter=None):
    results = []
    for fn in os.listdir(data_dir):
        if fn.startswith('.'): continue
        if name_filter and name_filter not in fn: continue
        path = os.path.join(data_dir, fn)
        if os.path.isdir(path):
            results.append((path, 'dir', 'unknown', 'directory', None)); continue
```

```
ftype = sniff_file_type(path)
kind, obj = load_any(path)
summary, willow = '', None

if kind == 'array':
    summary = f'array shape={obj.shape} dtype={obj.dtype}'
elif kind == 'mapping':
    keys = list(obj.keys())[:12]
    summary = f'mapping keys={keys}'
    willow = assemble_willow_from_mapping(obj)
elif kind == 'hdf5':
    try:
        sample = _sample_hdf5_obj(obj)
        summary = f'hdf5 sampled keys={list(sample.keys())[:12]}'
        willow = assemble_willow_from_mapping(sample)
    finally:
        try: obj.close()
        except Exception: pass
elif kind == 'text':
    summary = 'text'
else:
    summary = 'unknown/unreadable'

results.append((path, ftype, kind, summary, willow))
```

```
    return results
```

result: HDF5 files that squirrel data under one extra group get picked up without writing a bespoke loader for every lab’s house style.

---

3) Merge policy for disagreeing shapes

We’ll centralize the merge logic and surface it from the CLI.

new helper:

willowlab/ingest/merge_policy.py

```
# willowlab/ingest/merge_policy.py
import numpy as np
from typing import Dict, Tuple


def derive_from_operators(U):
    """Compute eigenpairs from Floquet operators U."""
    U = np.asarray(U)
    if U.ndim == 3:
        T, N, _ = U.shape
        evals = np.empty((T, N), dtype=np.complex128)
        evecs = np.empty_like(U)
        for t in range(T):
            w, V = np.linalg.eig(U[t])
            evals[t] = w; evecs[t] = V
        return evals, evecs
    elif U.ndim == 2:
        w, V = np.linalg.eig(U)
        return w[None, :], V[None, :, :]
    raise ValueError("U must be 2D or 3D array.")


def reconcile_eigenpairs(mapping: Dict, policy: str = "auto") -> Dict:
    """
    Resolve conflicts between supplied eigenpairs and operator-derived ones.
    policy ∈ {'auto','prefer_operator','prefer_supplied'}
    """
    out = dict(mapping)
    has_U  = 'floquet_operators' in out and out['floquet_operators'] is not None and
out['floquet_operators'] != []
    has_ev = 'floquet_eigenvalues' in out and out['floquet_eigenvalues'] is not None and
out['floquet_eigenvalues'] != []

    if not has_U and not has_ev:
        return out  # nothing to do

    derived_evals = derived_evecs = None
    if has_U:
        try:
            derived_evals, derived_evecs = derive_from_operators(out['floquet_operators'])
        except Exception:
            derived_evals = derived_evecs = None
```

```
if not has_ev and derived_evals is not None:
    out['floquet_eigenvalues']  = derived_evals
    out.setdefault('floquet_eigenvectors', derived_evecs)
    return out

if has_ev and derived_evals is None:
    return out

# both present → shapes may disagree
ev_sup = np.asarray(out['floquet_eigenvalues'])
JT     = np.asarray(out.get('JT_scan_points')) if 'JT_scan_points' in out else None
sup_ok = ev_sup.ndim == 2

drv_ok = derived_evals is not None and derived_evals.ndim == 2
# shape sanity
def _score(ev):
    if JT is None: return 0
    return int(ev.shape[0] == JT.shape[0])

if policy == "prefer_operator" and drv_ok:
    out['floquet_eigenvalues']  = derived_evals
    out.setdefault('floquet_eigenvectors', derived_evecs)
elif policy == "prefer_supplied" and sup_ok:
    # keep supplied; only fill evecs if missing
```

```
        if 'floquet_eigenvectors' not in out and derived_evecs is not None and ev_sup.shape == derived_evals.shape:
            out['floquet_eigenvectors'] = derived_evecs
    else:  # auto
        # prefer the one that matches JT length; tie-breaker: larger N (more bands)
        if drv_ok and ( _score(derived_evals) > _score(ev_sup) ):
            out['floquet_eigenvalues']  = derived_evals
            out.setdefault('floquet_eigenvectors', derived_evecs)
        elif drv_ok and sup_ok and _score(derived_evals) == _score(ev_sup):
            choose_derived = (derived_evals.shape[1] >= ev_sup.shape[1])
            if choose_derived:
                out['floquet_eigenvalues']  = derived_evals
                out.setdefault('floquet_eigenvectors', derived_evecs)
            # else keep supplied
        # else keep supplied
    return out
```

---

4) Wire it all into the assembler workflow

modify

willowlab/ingest/assemble_workflow.py

- add cache usage (skip unchanged),
- include HDF5 walker already via index_willow_dir,
- add merge_policy parameter, and call reconcile_eigenpairs.

```
# willowlab/ingest/assemble_workflow.py
import os, json, pathlib, glob
from typing import List, Optional
from .sniffers import triage_all_zips
from .file_index import index_willow_dir
from .normalize import to_willow_dataset, persist_dataset
from .merge_policy import reconcile_eigenpairs
from .cache_dir import DirCache
from ..io import load_willow
from ..trinity import WillowTrinityStep1
from ..tests.t_spec_ent import test_duality

PRIORITY_HINTS = ("floquet_braiding", "braiding_2_plaquettes")

def _pick_best(candidates):
    for src, name, w in candidates:
        if ('floquet_eigenvalues' in w) and ('JT_scan_points' in w):
            return (src, name, w)
    for src, name, w in candidates:
        if 'floquet_eigenvalues' in w: return (src, name, w)
    for src, name, w in candidates:
        if 'JT_scan_points' in w: return (src, name, w)
    return (None, None, None)
```

```
def assemble_from_zips(zip_paths: List[str], out_dir: str, expanded_dirs: Optional[List[str]] = None,
                      merge_policy: str = "auto", cache_root: Optional[str] = None):
    os.makedirs(out_dir, exist_ok=True)
    cache_root = cache_root or os.path.commonpath(zip_paths + (expanded_dirs or [])) if zip_paths else (expanded_dirs[0] if expanded_dirs else out_dir)
    dcache = DirCache(cache_root)

    # Stage A: zip triage (skip cached zips that haven't changed)
    zip_paths = [p for p in zip_paths if dcache.should_process(p)]
    report, assembled = triage_all_zips(zip_paths) if zip_paths else ([], [])

    # bump priority
    assembled.sort(key=lambda x: (any(h in x[0].lower() for h in PRIORITY_HINTS) is False))

    src, inner, mapping = _pick_best(assembled)

    # Stage B: deep dive expanded dirs (skip cached files internally)
    if (mapping is None or 'floquet_eigenvalues' not in mapping or 'JT_scan_points' not in mapping) and expanded_dirs:
        merged = dict(mapping or {})
        for d in expanded_dirs:
            # skip whole dir if none of its files changed
            inv = index_willow_dir(d)
            for path, ftype, kind, summary, w in inv:
                if os.path.isfile(path) and not dcache.should_process(path):
                    continue
                if not w: continue
                for k in ('JT_scan_points','floquet_eigenvalues','floquet_eigenvectors','floquet_operators'):
                    if k in w and k not in merged:
                        merged[k] = w[k]
        if merged:
            mapping = merged

    if mapping is None:
        raise RuntimeError("No usable Willow mapping found in zips or expanded dirs.")

    # Merge-policy resolution
    mapping = reconcile_eigenpairs(mapping, policy=merge_policy)

    meta = {"source_zip": src, "source_entry": inner, "note": f"assembled by WillowLab (merge_policy={merge_policy})"}
    ds = to_willow_dataset(mapping, meta=meta)
    npz_path, meta_path = persist_dataset(ds, out_dir)

    # Update cache records for processed inputs
    for zp in zip_paths:
        dcache.update(zp, bundle=npz_path)
    if expanded_dirs:
        for d in expanded_dirs:
            for leaf in glob.glob(os.path.join(d, "*")):
                if os.path.isfile(leaf):
                    dcache.update(leaf, bundle=npz_path)
    dcache.persist()


    # Arm the test suite
    loaded = load_willow(npz_path)
    tri = WillowTrinityStep1(loaded)
    inv = tri.compute_all(jt_star=1.0, window=0.05)
    summary = {"bundle": npz_path, "meta": meta_path, "invariants": inv}
    if (loaded.resolvent_trace is not None) and (loaded.entropy is not None) and (loaded.effective_energy is not None):
        summary["t_spec_ent"] = test_duality(loaded)


    out_json = pathlib.Path(out_dir, "assembly_summary.json")
    out_json.write_text(json.dumps(summary, indent=2, default=lambda x: str(x)))
    return str(out_json)
```

---

5) CLI updates

modify

willowlab/ingest/cli_assemble.py

Add --merge-policy and --cache-root.

```
# willowlab/ingest/cli_assemble.py
import argparse, glob
```

```
from .assemble_workflow import assemble_from_zips


def main():
    ap = argparse.ArgumentParser(description="Assemble Willow bundles from zips + expanded dirs")
    ap.add_argument("--zip-glob", required=True, help="Glob for zip files (e.g., '/data/*.zip')")
    ap.add_argument("--expanded-dirs", nargs="*", default=[], help="Dirs expanded next to each zip")
    ap.add_argument("--out", required=True, help="Output directory")
    ap.add_argument("--merge-policy", choices=["auto","prefer_operator","prefer_supplied"], default="auto",
                    help="Resolve eigenpair conflicts")
    ap.add_argument("--cache-root", default=None, help="Root directory for incremental cache DB")
    args = ap.parse_args()

    zip_paths = glob.glob(args.zip_glob)
    summary_path = assemble_from_zips(zip_paths, args.out, expanded_dirs=args.expanded_dirs,
                                      merge_policy=args.merge_policy, cache_root=args.cache_root)
    print(f"Wrote {summary_path}")


if __name__ == "__main__":
    main()
```

usage examples:

```
# 1) first run: full ingest, prefer operator-derived eigenpairs on conflicts
python -m willowlab.ingest.cli_assemble \
  --zip-glob "/data/willow/*.zip" \
  --expanded-dirs /data/willow/Floquet_Braiding /data/willow/Braiding_2_plaquettes \
  --out ./artifacts_assembled \
  --merge-policy prefer_operator \
  --cache-root /data/willow


# 2) subsequent run: unchanged files are skipped instantly via cache
python -m willowlab.ingest.cli_assemble \
  --zip-glob "/data/willow/*.zip" \
  --expanded-dirs /data/willow/Floquet_Braiding /data/willow/Braiding_2_plaquettes \
  --out ./artifacts_assembled \
  --merge-policy auto \
  --cache-root /data/willow
```

---

sanity checks and edge guards

- cache correctness: we update the cache only after successful assembly. if you want paranoid integrity, switch should_process to recompute hashes on every run—but size+mtime is a pragmatic default.
- HDF5 walker budget: budget_items and budget_elems keep memory in check; crank them up if you trust the machine.
- merge-policy auto: aims for the eigenpair set whose time-length matches JT and tie-breaks on band count. that keeps surfaces aligned and avoids silent truncation.

---

you now have an incremental, format-savvy ingestion line that politely skips the déjà vu, yanks eigenpairs from operators when told, and peeks one level deeper into HDF5 labyrinths. point it at your zips + expanded dirs, choose your merge policy, and let the toolkit auto-populate pristine Willow bundles for the test batteries and cartography engine.