alright—time to turn your chaotic zoo of zips and nameless blobs into a clean, self-feeding pipeline that auto-populates our Willow schema and arms the full test suite. we'll retrofit the sniffers/extractors you drafted, wire them into a deterministic assembler, and add a "zip triage → deep dive → normalize → test" conveyor belt.

below is a turnkey workflow + code you can drop straight into the toolkit. it respects your "scan the zips first, then deep dive the cash cows" plan, copes with missing filenames, and emits standardized .npz/.h5 Willow bundles ready for the quick spike, Trinity Step-1, geometry, and cartography modules.

---

the plan (short + ruthless)

1. zip triage (Stage A): shallow-inspect each zip without full extraction, fingerprint inner files by magic bytes, and auto-extract promising payloads (JT, eigenvalues/eigenvectors, U) into candidate Willow bundles.

2. deep dive (Stage B): for top-priority zips, or their expanded folders beside them, load oddball formats (HDF5/MAT/Parquet), merge partials (eigs here; JT there), and finalize a complete Willow dataset.

3. normalize & persist: convert to the canonical Willow schema (keys + shapes) and save as willow_<content-hash>.npz and optional .h5 mirror. also write a manifest JSON so we can trace provenance.

4. arm the test suite: immediately run your quick smoking gun and then the heavier Trinity Step-1 analyzer; cache artifacts for geometry/cartography.

---

code: retrofit sniffers → assemblers → normalizers → runners

drop these into willowlab/ingest/ (or similar). they merge your utilities and add the missing glue.

1)

sniffers.py

— zip + file sniffers (adapted from your PDFs)

```
# willowlab/ingest/sniffers.py

import os, io, json, gzip, zipfile, hashlib, pathlib

import numpy as np



# --- heuristic key sets (from your drafts) ---

LIKELY_EVAL_KEYS =
('eigenvalues','floquet_eigenvalues','evals','lambdas','lambda','eigvals','quasienergies','theta')

LIKELY_EVEC_KEYS = ('eigenvectors','floquet_eigenvectors','evecs','eigvecs')

LIKELY_U_KEYS    = ('U_floquet','floquet_operator','floquet_operators','U','UF','U_Floquet')

LIKELY_JT_KEYS   = ('JT_values','JT','jt','drive_strength','JT_scan_points','drive','t_values')



def _ensure_complex_evals(arr):

    arr = np.asarray(arr)

    return np.exp(1j*arr) if np.isrealobj(arr) else arr.astype(np.complex128, copy=False)



def _maybe_np_load(b):

    # npz or npy in bytes

    try:

        with np.load(io.BytesIO(b), allow_pickle=False) as npz:

            return {k: npz[k] for k in npz.files}

    except Exception:

        try:
            return np.load(io.BytesIO(b), allow_pickle=False)
        except Exception:
            return None


def _sniff_inner_bytes(b):
    h = b[:16]
    if h.startswith(b'\x93NUMPY'): return 'npy'
    if h.startswith(b'PK\x03\x04'): return 'npz/zip'
    if h.startswith(b'\x89HDF\r\n\x1a\n'): return 'hdf5'
    if h.startswith(b'PAR1'): return 'parquet'
    if b'MATLAB' in b[:256]: return 'mat'
    try:
        t = b[:256].decode('utf-8', errors='ignore').strip()
        if t.startswith('{') or t.startswith('['): return 'json'
        if (',' in t) or ('\t' in t): return 'csv'
    except Exception:
        pass
    return 'unknown'


def _extract_willow_from_mapping(mapping):
    willow = {}
    def find_key(cands):
        for k in mapping.keys():
            kl = k.lower()
            for c in cands:
                if c.lower() in kl: return k
        return None
    kJT = find_key(LIKELY_JT_KEYS)
    if kJT is not None: willow['JT_scan_points'] = np.asarray(mapping[kJT]).ravel()
    ke = find_key(LIKELY_EVAL_KEYS)
    if ke is not None: willow['floquet_eigenvalues'] = _ensure_complex_evals(mapping[ke])
    kev = find_key(LIKELY_EVEC_KEYS)
    if kev is not None: willow['floquet_eigenvectors'] = np.asarray(mapping[kev])
    kU = find_key(LIKELY_U_KEYS)
    if kU is not None: willow['floquet_operators'] = np.asarray(mapping[kU])

    # derive evals/evecs if only U is present
    if 'floquet_eigenvalues' not in willow and 'floquet_operators' in willow:
        U = willow['floquet_operators']
        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
            willow['floquet_eigenvalues'] = evals
            willow.setdefault('floquet_eigenvectors', evecs)
        elif U.ndim == 2:
            w, V = np.linalg.eig(U)
            willow['floquet_eigenvalues'] = w[None, :]
            willow['floquet_eigenvectors'] = V[None, :, :]
            willow.setdefault('JT_scan_points', np.array([np.nan]))
    return willow if willow else None


def scan_zip_for_willow(zip_path, max_bytes=2_000_000):
    """
    Stage A: shallow scan inside a .zip (or .npz) without full extraction.
    Returns (manifest, candidates)
    """
    cands, manifest = [], []
    with zipfile.ZipFile(zip_path, 'r') as zf:
        for zi in zf.infolist():
            if zi.is_dir(): continue
            name = zi.filename
            lower = name.lower()
            if not any(k in lower for k in ('floquet','eval','eig','jt','u_','operator','braid','czuniform','data')):
                continue
            if zi.file_size > max_bytes and not any(k in lower for k in
('floquet','eigen','u_floquet','operator','jt')):
                manifest.append((name, 'skipped(huge)')); continue


            b = zf.read(zi)
            ftype = _sniff_inner_bytes(b)
            manifest.append((name, ftype))

            if ftype in ('npy','npz/zip'):
                payload = _maybe_np_load(b)
                if isinstance(payload, dict):
                    w = _extract_willow_from_mapping(payload)
                    if w: cands.append((name, w))
                elif isinstance(payload, np.ndarray):
                    arr = payload
                    if 'jt' in lower: cands.append((name, {'JT_scan_points': arr}))
                    elif any(s in lower for s in ('eig','lambda','quasi')):
                        cands.append((name, {'floquet_eigenvalues': _ensure_complex_evals(arr)}))
            elif ftype == 'json':
                try:
                    d = json.loads(b.decode('utf-8', errors='ignore'))
                    if isinstance(d, dict):
                        w = _extract_willow_from_mapping(d)
                        if w: cands.append((name, w))
                except Exception:
                    pass
    return manifest, cands


def triage_all_zips(zip_paths):
    report, assembled = [], []

    for zp in zip_paths:

        manifest, cands = scan_zip_for_willow(zp)

        report.append((os.path.basename(zp), manifest))

        assembled.extend([(os.path.basename(zp), n, w) for (n, w) in cands])

    return report, assembled


def content_hash(willow_mapping):

    h = hashlib.sha256()

    for k in sorted(willow_mapping.keys()):

        v = np.asarray(willow_mapping[k])

        h.update(k.encode()); h.update(v.shape.__repr__().encode());
h.update(v.dtype.__repr__().encode())

        h.update(np.ascontiguousarray(v).data)

    return h.hexdigest()[:16]
```

why this matches your plan: it's the exact "zip-level sniffer, no full extract" approach with magic-byte detection and Willow key heuristics, adapted to skip monsters unless they look promising.

---

2)

file_index.py

— directory indexer for expanded folders & nameless files

```
# willowlab/ingest/file_index.py
```

```
import os, io, json, gzip, numpy as np

from .sniffers import (LIKELY_EVAL_KEYS, LIKELY_EVEC_KEYS, LIKELY_U_KEYS,
LIKELY_JT_KEYS,

                       _ensure_complex_evals, _extract_willow_from_mapping)


try:

    import h5py

except Exception:

    h5py = None

try:

    import scipy.io as sio

except Exception:

    sio = None


def sniff_file_type(path, read_bytes=16):

    with open(path, 'rb') as f:

        head = f.read(read_bytes)

    if head.startswith(b'\x1f\x8b'): return 'gzip'

    if head.startswith(b'PK\x03\x04'): return 'zip'

    if head.startswith(b'\x89HDF\r\n\x1a\n'): return 'hdf5'

    if head.startswith(b'PAR1'): return 'parquet'

    if head.startswith(b'ARROW1'): return 'feather'

    if head.startswith(b'\x93NUMPY'): return 'npy'

    if b'MATLAB' in head or b'MATLAB 5.0 MAT-file' in head: return 'mat'

    try:
        txt = head.decode('utf-8', errors='ignore').strip()
        if txt.startswith('{') or txt.startswith('['): return 'json'
        if (',' in txt) or ('\t' in txt): return 'csv'
    except Exception:
        pass
    return 'unknown'


def load_any(path):
    ftype = sniff_file_type(path)
    try:
        if ftype == 'npy':
            arr = np.load(path, allow_pickle=False); return 'array', arr
        if ftype == 'zip':
            # treat as npz if possible
            try:
                with np.load(path, allow_pickle=False) as npz:
                    return 'mapping', {k: npz[k] for k in npz.files}
            except Exception:
                return 'unknown', None
        if ftype == 'hdf5' and h5py is not None:
            f = h5py.File(path, 'r'); return 'hdf5', f
        if ftype == 'mat' and sio is not None:
            d = sio.loadmat(path, squeeze_me=True, struct_as_record=False); return 'mapping', d
        if ftype == 'json':
            with open(path, 'r', encoding='utf-8', errors='ignore') as f: return 'mapping', json.load(f)
        if ftype == 'csv':
            arr = np.genfromtxt(path, delimiter=None, dtype=float); return 'array', arr
        # fallback attempts
        try:
            with np.load(path, allow_pickle=False) as npz:
                return 'mapping', {k: npz[k] for k in npz.files}
        except Exception:
            pass
        try:
            with open(path, 'r', encoding='utf-8', errors='ignore') as f:
                return 'text', f.read()
        except Exception:
            return 'unknown', None
    except Exception:
        return 'unknown', None


def assemble_willow_from_mapping(mapping):
    # identical logic as our zip path—kept here for local use
    from .sniffers import _extract_willow_from_mapping
    return _extract_willow_from_mapping(mapping)


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 in ('mapping','hdf5'):
        if kind == 'mapping':
            keys = list(obj.keys())[:12]
            summary = f'mapping keys={keys}'
            willow = assemble_willow_from_mapping(obj)
        else:
            keys = list(obj.keys())
            summary = f'hdf5 keys(top)={keys[:12]}'
            # build a light dict view of likely datasets
            d = {}
            for k in keys:
            try:
                node = obj[k]
                if hasattr(node, 'shape'):
                    d[k] = node[()] if node.size < 1e6 else node[:min(4, node.shape[0])]
            except Exception:
                pass
            willow = assemble_willow_from_mapping(d)
    elif kind == 'text':
        summary = f'text'
    else:
        summary = 'unknown/unreadable'

    results.append((path, ftype, kind, summary, willow))
    if kind == 'hdf5':
        try: obj.close()
        except Exception: pass
return results
```

this covers: nameless files in expanded folders, magic-byte sniffing, and assembly into Willow mappings—exactly what your Deep dive assembler prescribes.

---

3)

normalize.py

— convert any candidate mapping → canonical Willow bundle

```
# willowlab/ingest/normalize.py

import os, json, numpy as np

from dataclasses import asdict

from ..schema import WillowDataset

from .sniffers import content_hash


REQUIRED = ("JT_scan_points","floquet_eigenvalues")


def to_willow_dataset(mapping, meta=None):

    meta = dict(meta or {})

    JT  = np.asarray(mapping.get("JT_scan_points")) if "JT_scan_points" in mapping else None

    lam = np.asarray(mapping.get("floquet_eigenvalues")) if "floquet_eigenvalues" in mapping
else None

    V   = mapping.get("floquet_eigenvectors")

    U   = mapping.get("floquet_operators")

    S   = mapping.get("entropy")

    E   = mapping.get("effective_energy")

    eta = mapping.get("eta_oscillations")

    ch2 = mapping.get("chern_mod2")

    xings = mapping.get("spectral_flow_crossings")

    Ovl = mapping.get("overlap_matrices")


    ds = WillowDataset(

        JT_scan_points=JT if JT is not None else np.arange(lam.shape[0]) if lam is not None else
np.arange(1),
        floquet_eigenvalues=lam,
        floquet_eigenvectors=V,
        floquet_operators=U,
        resolvent_trace=None,
        entropy=S,
        effective_energy=E,
        eta_oscillations=eta,
        chern_mod2=ch2,
        spectral_flow_crossings=xings,
        overlap_matrices=Ovl,
        meta=meta
    )
    ds.check_basic()
    return ds


def persist_dataset(ds: WillowDataset, out_dir: str):
    os.makedirs(out_dir, exist_ok=True)
    mapping = {k: v for k,v in ds.__dict__.items() if k != "meta"}
    h = content_hash(mapping)
    base = os.path.join(out_dir, f"willow_{h}")
    np.savez_compressed(
        base + ".npz",
        JT_scan_points=ds.JT_scan_points,
        floquet_eigenvalues=ds.floquet_eigenvalues if ds.floquet_eigenvalues is not None else
np.array([]),
        floquet_eigenvectors=ds.floquet_eigenvectors if ds.floquet_eigenvectors is not None else np.array([]),
        floquet_operators=ds.floquet_operators if ds.floquet_operators is not None else np.array([]),
        entropy=ds.entropy if ds.entropy is not None else np.array([]),
        effective_energy=ds.effective_energy if ds.effective_energy is not None else np.array([]),
        eta_oscillations=ds.eta_oscillations if ds.eta_oscillations is not None else np.array([]),
        chern_mod2=ds.chern_mod2 if ds.chern_mod2 is not None else np.array([]),
        spectral_flow_crossings=ds.spectral_flow_crossings if ds.spectral_flow_crossings is not None else np.array([]),
        overlap_matrices=ds.overlap_matrices if ds.overlap_matrices is not None else np.array([]),
    )
    with open(base + ".meta.json", "w") as f:
        json.dump(ds.meta, f, indent=2, default=str)
    return base + ".npz", base + ".meta.json"
```

---

4)

assemble_workflow.py

— one command to rule them all

```
# willowlab/ingest/assemble_workflow.py
import os, json, pathlib
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 ..io import load_willow

from ..trinity import WillowTrinityStep1

from ..tests.t_spec_ent import test_duality


PRIORITY_HINTS = ("floquet_braiding", "braiding_2_plaquettes")  # from your plan


def _pick_best(candidates):

    # choose one with both eigenvalues and JT

    for src, name, w in candidates:

        if ('floquet_eigenvalues' in w) and ('JT_scan_points' in w):

            return (src, name, w)

    # fall back to eigs-only (we'll synthesize JT index)

    for src, name, w in candidates:

        if 'floquet_eigenvalues' in w:

            return (src, name, w)

    # fall back to JT-only (not ideal)

    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):

    os.makedirs(out_dir, exist_ok=True)
    # Stage A: zip triage

    report, assembled = triage_all_zips(zip_paths)

    # bump priority matches to the front

    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: if incomplete, harvest from expanded dirs next to each zip

    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:

            inv = index_willow_dir(d)

            # try to merge partials

            for _, _, _, _, w in inv:

                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 'floquet_eigenvalues' in merged and 'JT_scan_points' in merged:

                mapping = merged; break


    if mapping is None:

        raise RuntimeError("No usable Willow mapping found in zips or expanded dirs.")
    meta = {"source_zip": src, "source_entry": inner, "note": "assembled by WillowLab workflow"}
    ds = to_willow_dataset(mapping, meta=meta)
    npz_path, meta_path = persist_dataset(ds, out_dir)

    # Arm the test suite (quick spike + Trinity + optional duality)
    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}
    # Optional duality if present
    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)
```

this is your “new data assemblers workflow”: aims at zips first, then deep dives adjacent folders, auto-assembles to Willow, persists, and immediately runs tests—precisely your request.

---

5) CLI wrapper (so you can point it at a directory)

```
# willowlab/ingest/cli_assemble.py
import argparse, os, glob, pathlib, json
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 for normalized bundles + summary")

    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)

    print(f"Wrote {summary_path}")



if __name__ == "__main__":

    main()
```

usage:

```
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
```

---

choices & guardrails baked in

- nameless files? the assembler names outputs by content hash, not filenames, and stores a .meta.json with provenance. no names required—just data.
- zips before deep dives: the triage skips huge entries unless they look “golden,” then escalates to expanded folders for merging partials. matches your “scan zips first then deep dive the cash cows” doctrine.
- angles vs eigenvalues: real quasienergies are auto-lifted to e^{i\theta}. ✔
- HDF5/MAT/Parquet: supported in the deep dive indexer; heavy datasets are read shallowly to avoid RAM geysers.
- immediate testing: every assembled bundle runs the quick spike and Trinity Step-1 so you get a green/red signal instantly.

---

finish strong: “quick spike” baked in (for completeness)

If you want the single function at your fingertips during ad-hoc pokes:

```
# willowlab/ingest/quick_spike.py
import numpy as np


def quick_smoking_gun(willow_data, jt_star=1.0, window=0.05, hard_thresh=1e6, dyn_factor=100.0):
    lam = np.asarray(willow_data['floquet_eigenvalues'])
    JT  = np.asarray(willow_data['JT_scan_points'])
    T, N = lam.shape
    phi = np.angle(np.prod(lam, axis=1)) / N
    lam_aligned = lam * np.exp(-1j * phi[:, None])
    with np.errstate(divide='ignore', invalid='ignore'):
        trace_res = np.sum(1.0 / (1.0 - lam_aligned), axis=1)

    i_star = int(np.argmin(np.abs(JT - jt_star)))

    local = (JT >= jt_star - window) & (JT <= jt_star + window)

    local_else = local & (np.arange(T) != i_star)

    baseline = np.median(np.abs(trace_res[local_else])) if np.any(local_else) else
np.median(np.abs(trace_res))

    peak = np.abs(trace_res[i_star])

    min_d1 = np.min(np.abs(1.0 - lam_aligned[i_star]))

    explodes = (peak > hard_thresh) or (peak > dyn_factor * max(baseline, 1.0))

    return {"explodes": bool(explodes), "JT_star": float(JT[i_star]), "peak": float(peak),

            "baseline": float(baseline), "ratio": float(peak/max(baseline,1e-12)),

            "min_dist_to_one": float(min_d1), "trace_series": trace_res}
```

that’s your PDF logic, modularized.

---

where to aim it first (battle plan)

- Floquet_Braiding.zip and Braiding_2_plaquettes.zip → Stage A, then immediate assembly and tests.
- expand these two beside the zips and pass the expanded dirs too; the workflow will merge partials (eigs from one file, JT from another).
- if neither lights up, escalate to the 1.8 GB monster with the same tooling (it’ll still shallow-scan first).

---

final mile: it’s all wired to the toolkit

- outputs are canonical Willow bundles that any of our modules accept (Trinity, geometry, cartography, T^{14}).
- artifacts include assembly_summary.json (invariants + paths), so you can hand the bundle straight to the cartography demo we built earlier.
- portability means this will digest anyone's "quantum data du jour"—as long as they give us U, eigenpairs, or JT in any of the formats we sniff.

if you want me to sharpen one more blade, the next upgrade would be:

- incremental cache over directories (skip re-ingesting unchanged files by content hash),
- a tiny HDF5 walker (one extra level) for stubborn group layouts,
- and a --merge-policy flag to prefer operator-derived eigenpairs over supplied ones when shapes disagree.

you've got a single ingest point, a disciplined assembler, and an armed test bench. point it at the zips, keep the expanded directories nearby, and let it auto-populate the schema while we go hunt for EP mountain passes and 14-D trophies.