

Hyperspectral time-series clustering of microalgal growth kinetics for strain-aware harvest timing

Rajalakshmi Elumalai^a, Karthikeyan Meenatchi Sundaram^b, Sunil Chinnadurai^b, Ankita Juneja^c, Karthik Rajendran^a,

*

^aThe NET ZERO Lab, Department of Environmental Science and Engineering, SRM University-AP, Amaravati, 522 240, Andhra Pradesh, India

^bDepartment of Electronics and Communication Engineering, SRM University-AP, Amaravati, 522 240, Andhra Pradesh, India

^cDepartment of Chemical Engineering, State University of New York College of Environmental Science and Forestry, Syracuse, NY, 13210, USA

Corresponding author: rajendran.k@srmap.edu.in

Step 1 — Training and validation (produces outputs/model_results.pkl)

"""

01_training_and_validation.py (STANDALONE -- single file, no local imports)

=====

Trains the 3-model DTW clustering pipeline (Hard-DTW K-Means, Soft-DTW K-Means, Hierarchical/DTW), builds the ensemble consensus, and runs internal + biological validation. Saves everything the companion figures script needs to outputs/model_results.pkl.

This file is fully self-contained: everything from your uploaded model_training_evaluation_FIXED.py is pasted in verbatim below (unchanged), followed by a script-friendly driver at the bottom. Drop this ONE file into any folder and run it -- it does not import dtw_pipeline_core.py or any other local module, so "fresh folder" / working-directory issues cannot happen anymore.

NO FIGURES ARE GENERATED HERE (a couple of lightweight QC plots -- silhouette profiles, dendrogram, co-association heatmap -- still fire as a side effect of training/model-selection, same as in your original script; that is unchanged pipeline behaviour, not new figure code).

Run:

```
python 01_training_and_validation.py
```

Requires: numpy, pandas, matplotlib, scipy, scikit-learn, tslearn.

"""

```
import os
```

```
import pickle
```

```
import re
```

```
import logging
```

```

import argparse

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from matplotlib.path import Path as MplPath

from scipy.cluster.hierarchy import dendrogram, linkage as scipy_linkage, fcluster
from scipy.spatial.distance import squareform
from scipy.stats import kruskal

from sklearn.cluster import AgglomerativeClustering
from sklearn.decomposition import PCA
from sklearn.metrics import (
    silhouette_score, davies_bouldin_score, silhouette_samples,
    adjusted_rand_score, normalized_mutual_info_score,
)
from sklearn.preprocessing import LabelEncoder

from tslearn.metrics import cdist_dtw
from tslearn.clustering import TimeSeriesKMeans
from tslearn.preprocessing import TimeSeriesScalerMeanVariance

# numpy >= 2.0 renamed trapz -> trapezoid; keep this script working on either.
_trapz = getattr(np, "trapezoid", None) or np.trapz

logging.basicConfig(level=logging.INFO, format="%(asctime)s [%(levelname)s] %(message)s")
logger = logging.getLogger("DTW_Consensus_Pipeline")

```

SEED = 42

```
def set_seed(seed: int = SEED):
```

```
    np.random.seed(seed)
```

```
# =====
```

```
# 0. POINT THIS AT YOUR OWN DATASET
```

```
# =====
```

```
# <<< CHANGE THIS to your dataset folder, or pass --data_dir when running >>>
```

```
DATA_DIR = r"D:\final rr\EXCEL DATA SPECIES WISE DATA MAIN"
```

```
FILENAME_PATTERN = re.compile(r"Day[_-]?(\d+)_([A-Za-z]+)_(\d+)_(\d+)", re.IGNORECASE)
```

```
# The 20 canonical species codes from your Fig_2 reference image.
```

```
# Edit this list if your own dataset uses different codes/species.
```

```
# FIX #1 (species-code consistency): earlier revisions of this script changed
```

```
# "SCNE" -> "SCEN" here (and in SPECIES_TAXONOMY below) without updating the
```

```
# raw filenames, which actually use "SCNE". That mismatch is what produced
```

```
# the "1 species label(s) could not be confidently canonicalized" warning and
```

```
# the later "No SPECIES_TAXONOMY entry for: ['SCNE']" warning that silently
```

```
# dropped that species from taxonomy validation. Restored to "SCNE" so it
```

```
# matches the actual data on disk. If your filenames really do use "SCEN",
```

```
# change ONLY this constant back -- but then also rename it consistently in
```

```
# every dict below, and re-check the raw folder names first.
```

```
CANONICAL_SPECIES = [
```

```
    "ANBA", "APHA", "CHLO", "CLAM", "COCO", "COEL", "DESM",
```

```
    "DUNA", "GLEO", "MESH", "MONO", "NANO", "NOST", "PAVL", "PICH",
```

```
"PLAT", "SCNE", "SPIR", "TEDE", "TETR",  
]
```

```
# Full binomial names, for readable labels in figures/reports (edit as needed)
```

```
SPECIES_FULL_NAMES = {
```

```
  "ANBA": "Anabaena sp.", "APHA": "Aphanocapsa montana", "CHLO": "Chlorella vulgaris",  
  "CLAM": "Chlamydomonas sp.", "COCO": "Chroococcus minor", "COEL": "Coelastrella sp.",  
  "DESM": "Desmodesmus sp.", "DUNA": "Dunaliella sp.", "GLEO": "Gloeocapsa sp.",  
  "MESH": "Messastrum sp.", "MONO": "Monoraphidium sp.", "NANO": "Nannochloropsis oculata",  
  "NOST": "Nostoc sp.", "PAVL": "Pavlova lutheri", "PICH": "Picochlorum sp.",  
  "PLAT": "Platymonas sp.", "SCNE": "Scenedesmus sp.", "SPIR": "Spirulina sp.",  
  "TEDE": "Tetradismus sp.", "TETR": "Tetraselmis chuii",  
}
```

```
# -----
```

```
# Coarse taxonomic reference (phylum / class), used ONLY for independent
```

```
# post-hoc validation of clusters (section 6a) -- NEVER for clustering itself.
```

```
# EDIT THIS if your species list differs, or if you have a more authoritative
```

```
# taxonomic source for your specific strains.
```

```
# -----
```

```
SPECIES_TAXONOMY = {
```

```
  "ANBA": "Cyanobacteria", "APHA": "Cyanobacteria", "CHLO": "Chlorophyta",  
  "CLAM": "Chlorophyta", "COCO": "Cyanobacteria", "COEL": "Chlorophyta",  
  "DESM": "Chlorophyta", "DUNA": "Chlorophyta", "GLEO": "Cyanobacteria",  
  "MESH": "Chlorophyta", "MONO": "Chlorophyta",  
  "NANO": "Ochrophyta (Eustigmatophyceae)", "NOST": "Cyanobacteria",  
  "PAVL": "Haptophyta", "PICH": "Chlorophyta", "PLAT": "Chlorophyta",  
  "SCNE": "Chlorophyta", "SPIR": "Cyanobacteria", "TEDE": "Chlorophyta",  
  "TETR": "Chlorophyta",  
}
```

```
MAX_EDIT_DISTANCE = 1
```

```
def _levenshtein(a: str, b: str) -> int:
```

```
    if a == b:
```

```
        return 0
```

```
    la, lb = len(a), len(b)
```

```
    if la == 0:
```

```
        return lb
```

```
    if lb == 0:
```

```
        return la
```

```
    prev = list(range(lb + 1))
```

```
    for i, ca in enumerate(a, start=1):
```

```
        curr = [i] + [0] * lb
```

```
        for j, cb in enumerate(b, start=1):
```

```
            cost = 0 if ca == cb else 1
```

```
            curr[j] = min(prev[j] + 1, curr[j - 1] + 1, prev[j - 1] + cost)
```

```
        prev = curr
```

```
    return prev[lb]
```

```
def _validate_canonical_species_distinctness(canonical=CANONICAL_SPECIES,  
max_edit_dist=MAX_EDIT_DISTANCE):
```

```
    flagged = []
```

```
    for i, a in enumerate(canonical):
```

```
        for b in canonical[i + 1:]:
```

```
            dist = _levenshtein(a, b)
```

```
            if dist <= max_edit_dist:
```

```
                flagged.append((a, b, dist))
```

```
    if flagged:
```

for a, b, dist in flagged:

```
    logger.warning(
        "Canonical species codes %r and %r are only %d edit(s) apart "
        "(<= MAX_EDIT_DISTANCE=%d): a typo of one could be mistaken for "
        "the other. Review CANONICAL_SPECIES / MAX_EDIT_DISTANCE.",
        a, b, dist, max_edit_dist,
    )
```

else:

```
    logger.info(
        "Canonical species codes are all > %d edit(s) apart from each other; "
        "bounded-edit-distance fuzzy correction is unambiguous.", max_edit_dist,
    )
```

`_validate_canonical_species_distinctness()`

`def canonicalize_species(raw_species, parent_folder_species=None, canonical=CANONICAL_SPECIES,`

`max_edit_dist=MAX_EDIT_DISTANCE, source_file=None):`

`"""Resolves raw filename/folder species text into a strict canonical code,
auto-fixing small typos via parent-folder context and edit distance."""`

`raw_u = raw_species.upper().strip()`

`if raw_u in canonical:`

`return raw_u, False, False`

`if parent_folder_species:`

`parent_u = parent_folder_species.upper().strip()`

`if parent_u in canonical and _levenshtein(raw_u, parent_u) <= 2:`

`logger.info("Automatically resolved filename typo %r -> %r based on parent folder (file: %s)",`

```

        raw_species, parent_u, source_file or "")

    return parent_u, True, False

candidates = [c for c in canonical if _levenshtein(raw_u, c) <= max_edit_dist]

if len(candidates) == 1:
    corrected = candidates[0]
    logger.info("Species-name correction: %r -> %r%s", raw_species, corrected,
                f" (file: {source_file})" if source_file else "")
    return corrected, True, False

if len(candidates) > 1:
    logger.warning(
        "Ambiguous species name %r%s: within edit distance %d of multiple canonical "
        "codes %s. Leaving uncorrected rather than guessing.",
        raw_species, f" (file: {source_file})" if source_file else "", max_edit_dist, candidates,
    )
    return raw_u, False, True

return raw_u, False, True

```

```

# Biologically-relevant spectral bands (nm) used to build Species x Day trajectories.
# Edit these if your instrument's wavelength range/pigment bands differ.
# FIX #2 (circularity, part A): declared once at module level so both the
# clustering step (which must EXCLUDE it) and the growth-curve validation
# step (which must USE it, and only it) agree on the same band. See the
# longer comment where this is consumed in main() for the full rationale.
GROWTH_VALIDATION_BAND = "Water_NIR_970"

```

```
DTW_BAND_INDICES = {
    "Chlorophyll_680": (670, 690),
    "Carotenoid_480": (470, 500),
    "Phycobiliprotein_620": (610, 640),
    "Lipid_CH_930": (910, 950),
    "Protein_Amide_910": (890, 920),
    "Water_NIR_970": (960, 980),
}
```

```
# =====
# 0a. Channel auto-detection / diagnostic
# =====
```

```
def _discover_available_channels(file_list, max_files_to_check=8):
```

```
    channels = set()
```

```
    checked = 0
```

```
    for path in file_list:
```

```
        if checked >= max_files_to_check:
```

```
            break
```

```
        try:
```

```
            xl = pd.ExcelFile(path)
```

```
            spec_df = xl.parse("Spectral_Data")
```

```
        except Exception:
```

```
            continue
```

```
        checked += 1
```

```
        for col in spec_df.columns:
```

```
            if col == "Wavelength_nm" or "_" not in col:
```

```
                continue
```

```
            channels.add(col.rsplit("_", 1)[-1])
```

```
return sorted(channels)
```

```
def _band_channel_coverage_report(file_list, band_defs=DTW_BAND_INDICES, sample_size=30):
```

```
    if not file_list:
```

```
        raise FileNotFoundError("No files provided to the channel/band diagnostic.")
```

```
    rng = np.random.RandomState(SEED)
```

```
    if len(file_list) <= sample_size:
```

```
        sample = list(file_list)
```

```
    else:
```

```
        idx = rng.choice(len(file_list), size=sample_size, replace=False)
```

```
        sample = [file_list[i] for i in sorted(idx)]
```

```
    channels = _discover_available_channels(sample, max_files_to_check=len(sample))
```

```
    if not channels:
```

```
        raise ValueError(
```

```
            "Could not find any '<ROI_Name>_<channel>' columns in the 'Spectral_Data' sheet "
```

```
            "of the sampled files. Check that your files have the expected column layout."
```

```
        )
```

```
    logger.info("Channel/band diagnostic: sampling %d file(s), channels found: %s",
```

```
               len(sample), channels)
```

```
    rows = []
```

```
    for channel in channels:
```

```
        nonzero_counts = {b: 0 for b in band_defs}
```

```
        total_checked = {b: 0 for b in band_defs}
```

```
        for path in sample:
```

```
            try:
```

```

    xl = pd.ExcelFile(path)

    spec_df = xl.parse("Spectral_Data")
except Exception:

    continue

if "Wavelength_nm" not in spec_df.columns:

    continue

wl = spec_df["Wavelength_nm"].values.astype(np.float32)

roi_cols = [c for c in spec_df.columns if c.endswith(f"_{channel}")]

if not roi_cols:

    continue

X = spec_df[roi_cols].values.astype(np.float32).T

for band_name, (lo, hi) in band_defs.items():

    mask = (wl >= lo) & (wl <= hi)

    if not mask.any():

        continue

    band_vals = X[:, mask]

    total_checked[band_name] += 1

    if np.isfinite(band_vals).any() and np.nanstd(band_vals) > 1e-9 \

        and not np.allclose(np.nan_to_num(band_vals), 0.0):

        nonzero_counts[band_name] += 1

for band_name in band_defs:

    total = total_checked[band_name]

    frac_real = (nonzero_counts[band_name] / total) if total else 0.0

    rows.append({

        "channel": channel, "band": band_name,

        "fraction_files_with_real_signal": round(frac_real, 3),

        "n_files_checked": total,

    })

```

```

report_df = pd.DataFrame(rows).sort_values(["channel", "band"]).reset_index(drop=True)

per_channel_mean = (report_df.groupby("channel")["fraction_files_with_real_signal"]
                    .mean().sort_values(ascending=False))

return report_df, per_channel_mean


def diagnose_channel_band_coverage(data_dir, band_defs=DTW_BAND_INDICES, sample_size=30):
    all_files = []
    for root, _, fnames in os.walk(data_dir):
        for fn in fnames:
            if fn.lower().endswith(".xlsx") and not fn.startswith("~$"):
                all_files.append(os.path.join(root, fn))
    all_files = sorted(all_files)
    if not all_files:
        raise FileNotFoundError(
            f"'{data_dir}' contains no .xlsx files (recursive search). Check the path."
        )

    report_df, per_channel_mean = _band_channel_coverage_report(all_files, band_defs, sample_size)

    print(f"\n[channel diagnostic] Sampled from {len(all_files)} total .xlsx file(s) under '{data_dir}'")
    print("\n[channel diagnostic] Fraction of sampled files where each biochemical band has "
          "real (non-zero-variance) signal, per channel:")
    print(report_df.to_string(index=False))
    print("\n[channel diagnostic] Mean real-signal coverage across ALL bands, per channel "
          "(higher = safer choice for --use_channel; 'auto' picks the top row automatically):")
    print(per_channel_mean.to_string())

    best_channel = per_channel_mean.index[0]
    best_score = float(per_channel_mean.iloc[0])

```

```

if best_score < 1.0:

    zero_bands = report_df[(report_df["channel"] == best_channel) &
                           (report_df["fraction_files_with_real_signal"] < 0.5)][["band"].tolist()

    if zero_bands:

        print(f"\n[channel diagnostic] WARNING: even the best channel ('{best_channel}') has "
              f"little/no real signal in: {zero_bands}. Those bands will contribute ~no "
              f"cluster-separation signal even with this channel selected. Consider checking "
              f"the instrument export for those wavelength ranges, or editing DTW_BAND_INDICES "
              f"to drop bands your instrument doesn't actually measure.")

    return report_df, per_channel_mean

```

```

# =====
# 1. Real-data loader (walks DATA_DIR, no synthetic data used by default)
# =====

```

```

class MicroalgaeROIDataset:

    def __init__(self, data_dir, n_components=30, use_synthetic_fallback=False,
                 use_channel="auto", use_derivative=True, use_pigment_indices=True):

        self.n_components = n_components
        self.use_derivative = use_derivative
        self.use_pigment_indices = use_pigment_indices
        self.unresolved_species = []
        self.channel_diagnostic_report = None

    if not os.path.exists(data_dir):

        raise FileNotFoundError(

            f"data_dir does not exist: '{data_dir}'. Set DATA_DIR (or pass --data_dir "
            f"to your real dataset folder."

```

```

)

found_files = self._discover_files(data_dir)

if len(found_files) == 0:
    if not use_synthetic_fallback:
        raise FileNotFoundError(
            f"{data_dir}' exists but contains no usable .xlsx files. "
            f"Check the path and folder layout described at the top of this script."
        )
    logger.warning("No real data found under '%s'. FALLING BACK TO SYNTHETIC DATA.", data_dir)
    self.data_source = "synthetic"
    self.use_channel = "Raw" if use_channel == "auto" else use_channel
    self._build_synthetic()
else:
    self.data_source = "real"

    if use_channel == "auto":
        logger.info("--use_channel='auto': sampling files to pick the channel with the "
            "best real-signal coverage across the biochemical bands...")
        report_df, per_channel_mean = _band_channel_coverage_report(
            found_files, band_defs=DTW_BAND_INDICES, sample_size=min(30, len(found_files)))
        self.channel_diagnostic_report = report_df
        self.use_channel = per_channel_mean.index[0]
        best_score = float(per_channel_mean.iloc[0])
        logger.info(
            "Auto-selected --use_channel='%s' (mean real-signal coverage=%.2f across "
            "biochemical bands). Full per-band table:\n%s",
            self.use_channel, best_score, report_df.to_string(index=False),
        )

```

```

if best_score < 1.0:
    weak_bands = report_df[
        (report_df["channel"] == self.use_channel) &
        (report_df["fraction_files_with_real_signal"] < 0.5)
    ]["band"].tolist()
    if weak_bands:
        logger.warning(
            "Even the auto-selected channel ('%s') has little/no real signal in "
            "bands: %s. Those bands will not meaningfully separate clusters. Run "
            "with --diagnose_channels for the full picture, or edit "
            "DTW_BAND_INDICES to match what your instrument actually measures.",
            self.use_channel, weak_bands,
        )
    else:
        self.use_channel = use_channel

    self._load_real_files(found_files)

    self._preprocess_and_reduce()

def _discover_files(self, data_dir):
    files = []
    for root, _, fnames in os.walk(data_dir):
        for fn in fnames:
            if fn.lower().endswith(".xlsx") and not fn.startswith("~$"):
                files.append(os.path.join(root, fn))
    return sorted(files)

def _load_real_files(self, file_paths):
    all_spectra, labels, days, pigment_rows, groups = [], [], [], [], []

```

```

wavelengths_ref = None

n_ok, n_skip = 0, 0

corrections = {}

unresolved = set()


logger.info("Reading %d hyperspectral excel files from your dataset (channel='%s')...",
            len(file_paths), self.use_channel)
for f_idx, path in enumerate(file_paths):
    parent_species = os.path.basename(os.path.dirname(path))
    fn = os.path.basename(path)
    m = FILENAME_PATTERN.search(fn)
    if m:
        day, species_raw, image_idx = int(m.group(1)), m.group(2).upper(), int(m.group(3))
    else:
        day, species_raw, image_idx = -1, parent_species.upper(), -1

    species, corrected, unresolved_flag = canonicalize_species(
        species_raw, parent_folder_species=parent_species, source_file=fn
    )
    if corrected:
        corrections.setdefault((species_raw, species), 0)
        corrections[(species_raw, species)] += 1
    if unresolved_flag or (not corrected and species_raw.upper().strip() not in CANONICAL_SPECIES):
        unresolved.add(species)

    try:
        xl = pd.ExcelFile(path)
        spec_df = xl.parse("Spectral_Data")
    except Exception as e:
        logger.warning("Skipping %s: could not read 'Spectral_Data' sheet (%s)", path, e)

```

```

    n_skip += 1

    continue

if "Wavelength_nm" not in spec_df.columns:

    logger.warning("Skipping %s: 'Spectral_Data' sheet has no 'Wavelength_nm' column", path)

    n_skip += 1

    continue


wl = spec_df["Wavelength_nm"].values.astype(np.float32)

if wavelengths_ref is None:

    wavelengths_ref = wl


roi_cols = [c for c in spec_df.columns if c.endswith(f'_{self.use_channel}')]

if len(roi_cols) == 0:

    logger.warning("Skipping %s: no columns ending in '_%s' (check --use_channel)",
                    path, self.use_channel)

    n_skip += 1

    continue


meta_map = {}

try:

    meta_df = xl.parse("ROI_Metadata")

    if "ROI_Name" in meta_df.columns:

        for _, row in meta_df.iterrows():

            meta_map[row["ROI_Name"]] = row

    else:

        logger.warning("%s: 'ROI_Metadata' sheet has no 'ROI_Name' column; "
                        "pigment indices for this file will be imputed.", path)

except Exception as e:

    logger.info("%s: no usable 'ROI_Metadata' sheet (%s); pigment indices will be imputed.", path, e)

```

```

for col in roi_cols:

    roi_name = col[: -(len(self.use_channel) + 1)]

    spectrum = spec_df[col].values.astype(np.float32)

    if len(wl) != len(wavelengths_ref) or not np.allclose(wl, wavelengths_ref):

        spectrum = np.interp(wavelengths_ref, wl, spectrum)

    all_spectra.append(spectrum)

    labels.append(species)

    days.append(day)

    groups.append(f_idx)

    if roi_name in meta_map:

        mrow = meta_map[roi_name]

        pigment_rows.append((float(mrow.get("NDVI", np.nan)),

                               float(mrow.get("GNDVI", np.nan)),

                               float(mrow.get("Simple_Ratio", np.nan))))

    else:

        pigment_rows.append((np.nan, np.nan, np.nan))

    n_ok += 1

if len(all_spectra) == 0:

    raise ValueError("No usable ROI spectra were extracted from any file in your data_dir.")

logger.info("Successfully parsed %d files (%d skipped).", n_ok, n_skip)

if corrections:

    logger.info("Species-name correction counts: %s", corrections)

if unresolved:

    logger.warning(

        "%d species label(s) could not be confidently canonicalized and were kept "

        "as-is: %s. Inspect these manually.", len(unresolved), sorted(unresolved),

    )

self.unresolved_species = sorted(unresolved)

```

```

self.wavelengths = wavelengths_ref

self.all_features = np.stack(all_spectra, axis=0)

self.species_classes = sorted(set(labels))

self.class_to_idx = {c: i for i, c in enumerate(self.species_classes)}

self.labels = np.array([self.class_to_idx[l] for l in labels], dtype=np.int64)

self.days = np.array(days, dtype=np.int64)

self.groups = np.array(groups, dtype=np.int64)

self.pigment_indices = np.array(pigment_rows, dtype=np.float32)


def _build_synthetic(self, n_species=20, n_days=5, n_images=2, n_rois=15, n_bands=601, overlap=0.15):
    rng = np.random.RandomState(SEED)

    wl = np.linspace(400, 1000, n_bands).astype(np.float32)

    species_codes = CANONICAL_SPECIES[:n_species]

    feats, labels, days, pigment_rows, groups = [], [], [], [], []

    f_idx = 0

    for sp_i, sp in enumerate(species_codes):
        peak = 450 + sp_i * (200 / max(n_species, 1))

        ndvi_bias = 0.03 * sp_i

        for day in range(1, n_days + 1):
            growth = 1.0 + 0.1 * np.sin(day / n_days * np.pi)

            for img in range(1, n_images + 1):
                for roi in range(n_rois):
                    spec = growth * (0.4 + 0.35 * np.exp(-0.5 * ((wl - peak) / 30) ** 2) +
                                     0.15 * np.exp(-0.5 * ((wl - 680) / 15) ** 2))

                    spec += rng.normal(0, 0.02 * (1.0 + overlap), spec.shape)

                    feats.append(spec.astype(np.float32))

                    labels.append(sp)

                    days.append(day)

                    groups.append(f_idx)

                    pigment_rows.append((0.15 + ndvi_bias + rng.normal(0, 0.015),

```

```

        0.05 + ndvi_bias + rng.normal(0, 0.012),
        1.3 + rng.normal(0, 0.025)))

    f_idx += 1

    self.wavelengths = wl

    self.all_features = np.stack(feats, axis=0)

    self.species_classes = species_codes

    self.class_to_idx = {c: i for i, c in enumerate(self.species_classes)}

    self.labels = np.array([self.class_to_idx[l] for l in labels], dtype=np.int64)

    self.days = np.array(days, dtype=np.int64)

    self.groups = np.array(groups, dtype=np.int64)

    self.pigment_indices = np.array(pigment_rows, dtype=np.float32)

def _preprocess_and_reduce(self):
    X = self.all_features

    mean = X.mean(axis=1, keepdims=True)

    std = X.std(axis=1, keepdims=True) + 1e-8

    X_snv = (X - mean) / std

    if self.use_derivative:
        kernel = np.ones(5) / 5.0

        X_smooth = np.apply_along_axis(lambda r: np.convolve(r, kernel, mode="same"), axis=1, arr=X_snv)

        X_proc = np.gradient(X_smooth, axis=1)
    else:
        X_proc = X_snv

    self.pca = PCA(n_components=min(self.n_components, X_proc.shape[1]), random_state=SEED)

    pca_features = self.pca.fit_transform(X_proc).astype(np.float32)

    if self.use_pigment_indices:
        pig = self.pigment_indices.copy()

```

```

col_median = np.nanmedian(pig, axis=0)
for c in range(pig.shape[1]):
    pig[~np.isfinite(pig[:, c]), c] = col_median[c]
pig_mean, pig_std = pig.mean(axis=0), pig.std(axis=0) + 1e-8
pig_z = (pig - pig_mean) / pig_std
self.reduced_features = np.concatenate([pca_features, pig_z], axis=1).astype(np.float32)
else:
    self.reduced_features = pca_features

```

```

def meta_dataframe(self):
    species_names = [self.species_classes[i] for i in self.labels]
    return pd.DataFrame({"Species": species_names, "Day": self.days, "Source_File_Group": self.groups})

```

```

def band_average(X: np.ndarray, wavelengths: np.ndarray, band_defs: dict) -> pd.DataFrame:
    out = {}
    for name, (lo, hi) in band_defs.items():
        mask = (wavelengths >= lo) & (wavelengths <= hi)
        if not mask.any():
            raise ValueError(f'Band '{name}' ({lo}-{hi}nm) has no matching wavelengths in your data.')
        out[name] = X[:, mask].mean(axis=1)
    return pd.DataFrame(out)

```

```

def build_species_day_trajectories(ds: MicroalgaeROIDataset, band_defs=DTW_BAND_INDICES):
    meta = ds.meta_dataframe()
    band_df = band_average(ds.all_features, ds.wavelengths, band_defs)
    band_df["Species"] = meta["Species"].values
    band_df["Day"] = meta["Day"].values

```

```

agg = band_df.groupby(["Species", "Day"]).mean().reset_index()

species_order = sorted(agg["Species"].unique())
day_order = sorted(agg["Day"].unique())
band_names = list(band_defs.keys())

n_species, n_days, n_feat = len(species_order), len(day_order), len(band_names)
traj_array = np.full((n_species, n_days, n_feat), np.nan)

sp_idx = {s: i for i, s in enumerate(species_order)}
day_idx = {d: i for i, d in enumerate(day_order)}
for _, row in agg.iterrows():
    i, j = sp_idx[row["Species"]], day_idx[row["Day"]]
    traj_array[i, j, :] = row[band_names].values

if np.isnan(traj_array).any():
    for i in range(n_species):
        for k in range(n_feat):
            series = pd.Series(traj_array[i, :, k])
            traj_array[i, :, k] = series.interpolate(limit_direction="both").values

return traj_array, species_order, day_order, band_names

```

```

# =====
# 2. Metrics / reporting utilities
# =====

```

```

def cluster_balance_report(labels):
    labels = np.asarray(labels)

```

```

counts = pd.Series(labels).value_counts()
dominant_frac = counts.max() / len(labels)
return {
    "dominant_cluster_fraction": float(dominant_frac),
    "active_clusters": int(len(counts)),
    "counts": counts.to_dict(),
}

```

```

def clustering_validation_metrics(flat_features, cluster_labels, dtw_dist_matrix=None,
                                method_name="", data_source=""):
    n_unique = len(np.unique(cluster_labels))
    report = {"method_name": method_name, "data_source": data_source, "n_clusters": n_unique}

    if n_unique > 1:
        report["silhouette_flat_euclidean"] = float(silhouette_score(flat_features, cluster_labels))
        report["davies_bouldin_flat_euclidean"] = float(davies_bouldin_score(flat_features, cluster_labels))
        if dtw_dist_matrix is not None:
            report["silhouette_dtw_precomputed"] = float(
                silhouette_score(dtw_dist_matrix, cluster_labels, metric="precomputed"))
        else:
            report["silhouette_dtw_precomputed"] = None
    else:
        report["silhouette_flat_euclidean"] = -1.0
        report["davies_bouldin_flat_euclidean"] = 999.0
        report["silhouette_dtw_precomputed"] = None

    report["cluster_balance"] = cluster_balance_report(cluster_labels)
    return report

```

```

def stability_check_generic(fit_fn, k, n_seeds=5):
    all_labels = [fit_fn(k, s) for s in range(SEED, SEED + n_seeds)]
    aris = []
    for i in range(len(all_labels)):
        for j in range(i + 1, len(all_labels)):
            aris.append(adjusted_rand_score(all_labels[i], all_labels[j]))
    mean_ari = float(np.mean(aris)) if aris else 1.0
    return mean_ari, all_labels

```

```

# =====
# 3. Model 1: Hard-DTW K-means
# =====

```

```

def fit_dtw_kmeans(traj_scaled, n_clusters, seed, n_init=5, max_iter=50, sakoe_chiba_radius=3):
    model = TimeSeriesKMeans(
        n_clusters=n_clusters, metric="dtw",
        metric_params={"global_constraint": "sakoe_chiba", "sakoe_chiba_radius": sakoe_chiba_radius},
        max_iter=max_iter, n_init=n_init, random_state=seed, verbose=False,
    )
    labels = model.fit_predict(traj_scaled)
    return model, labels

```

```

def select_k_via_silhouette_dtw(traj_scaled, k_min=2, k_max=8, seed=SEED, max_dominant_frac=0.60):
    n_species = traj_scaled.shape[0]
    flat = traj_scaled.reshape(n_species, -1)
    rows = []
    for k in range(k_min, min(k_max, n_species - 1) + 1):

```

```

_model, labels = _fit_dtw_kmeans(traj_scaled, k, seed)

if len(np.unique(labels)) < 2:
    continue

sil = silhouette_score(flat, labels)
db = davies_bouldin_score(flat, labels)
balance = cluster_balance_report(labels)
rows.append({"k": k, "silhouette": sil, "davies_bouldin": db,
            "dominant_fraction": balance["dominant_cluster_fraction"],
            "active_clusters": balance["active_clusters"]})

sweep_df = pd.DataFrame(rows)
print("\n[k-selection] DTW K-means cluster count sweep:")
print(sweep_df.to_string(index=False))
if sweep_df.empty:
    logger.warning("k-sweep produced no valid candidates; falling back to k=4.")
    return 4, sweep_df

ok = sweep_df[sweep_df["dominant_fraction"] <= max_dominant_frac]
pool = ok if not ok.empty else sweep_df
if ok.empty:
    logger.warning("No k avoided a dominant cluster > %.0f%%; picking best silhouette anyway.",
                  max_dominant_frac * 100)
best_k = int(pool.loc[pool["silhouette"].idxmax(), "k"])
logger.info("Selected k=%d by best silhouette (%.4f) among balance-acceptable candidates",
            best_k, pool["silhouette"].max())
return best_k, sweep_df


def run_dtw_temporal_clustering(traj_array, species_order, day_order, band_names, ds,
                               n_clusters=4, seed=SEED, auto_k=True, k_min=2, k_max=8,

```

```

        n_stability_seeds=5, make_figures=True, shared_dtw_dist=None):

scaler = TimeSeriesScalerMeanVariance()
traj_scaled = scaler.fit_transform(traj_array)

n_species = traj_scaled.shape[0]
if k_max > n_species - 1:
    logger.warning(
        "Requested k_max=%d but only %d species are available (max usable k=%d); the sweep "
        "will be truncated to that range.", k_max, n_species, n_species - 1,
    )

k_sweep_df = None
if auto_k:
    n_clusters, k_sweep_df = select_k_via_silhouette_dtw(traj_scaled, k_min, k_max, seed)

model, cluster_labels = _fit_dtw_kmeans(traj_scaled, n_clusters, seed)

result_df = pd.DataFrame({"Species": species_order, "DTW_Cluster": cluster_labels})
print(f"\nSpecies -> DTW cluster assignment (k={n_clusters}):")
print(result_df.sort_values("DTW_Cluster").to_string(index=False))

flat = traj_scaled.reshape(n_species, -1)
report = clustering_validation_metrics(flat, cluster_labels, dtw_dist_matrix=shared_dtw_dist,
                                     method_name="DTW_KMeans_Kinetics", data_source=ds.data_source)

def fit_fn(k, s):
    _m, lbl = _fit_dtw_kmeans(traj_scaled, k, s)
    return lbl

mean_ari, _ = stability_check_generic(fit_fn, n_clusters, n_seeds=n_stability_seeds)

```

```

report["stability_mean_pairwise_ARI"] = mean_ari

report["k_used"] = n_clusters


if make_figures:

    plot_cluster_trajectories(traj_array, species_order, day_order, band_names, cluster_labels,
                              title=f"Model 1: DTW K-means Cluster Trajectories (k={n_clusters})",
                              out_stub="model1_dtw")

    plot_silhouette_profile(flat, cluster_labels, title="Model 1: DTW K-means Silhouette Profile",
                             out_stub="model1_dtw")


return {"result_df": result_df, "report": report, "k_sweep_df": k_sweep_df,
        "cluster_labels": cluster_labels, "flat": flat, "species_order": species_order,
        "cluster_col": "DTW_Cluster"}


# =====
# 4. Model 2: Soft-DTW K-means
# =====


def _fit_softdtw_kmeans(traj_scaled, n_clusters, seed, gamma=1.0, n_init=5, max_iter=100):

    model = TimeSeriesKMeans(
        n_clusters=n_clusters, metric="softdtw", metric_params={"gamma": gamma},
        n_init=n_init, max_iter=max_iter, random_state=seed, verbose=False,
    )

    labels = model.fit_predict(traj_scaled)

    return model, labels


def _default_gamma_grid(traj_scaled):

    n = traj_scaled.shape[0]

```

```

flat = traj_scaled.reshape(n, -1)

d2 = [np.sum((flat[i] - flat[j]) ** 2) for i in range(n) for j in range(i + 1, n)]

med = max(float(np.median(d2)) if d2 else 1.0, 1e-3)

return sorted(set([round(med * f, 5) for f in (0.01, 0.03, 0.1, 0.3, 1.0, 3.0)]))

def grid_search_softdtw(traj_scaled, k_min=2, k_max=8, seed=SEED, gamma_grid=None,
                        max_dominant_frac=0.60, n_stability_seeds=5, top_n_for_stability=4):
    n_species = traj_scaled.shape[0]
    flat = traj_scaled.reshape(n_species, -1)
    if gamma_grid is None:
        gamma_grid = _default_gamma_grid(traj_scaled)
    logger.info("Soft-DTW gamma grid: %s", gamma_grid)

    rows = []
    for gamma in gamma_grid:
        for k in range(k_min, min(k_max, n_species - 1) + 1):
            _model, labels = _fit_softdtw_kmeans(traj_scaled, k, seed, gamma=gamma)
            if len(np.unique(labels)) < 2:
                continue
            sil = silhouette_score(flat, labels)
            db = davies_bouldin_score(flat, labels)
            balance = cluster_balance_report(labels)
            rows.append({"gamma": gamma, "k": k, "silhouette": sil, "davies_bouldin": db,
                        "dominant_fraction": balance["dominant_cluster_fraction"],
                        "active_clusters": balance["active_clusters"]})

    sweep_df = pd.DataFrame(rows)

    print("\n[grid-search] Soft-DTW gamma x k sweep (top 15 by silhouette):")
    print(sweep_df.sort_values("silhouette", ascending=False).head(15).to_string(index=False))

```

```

if sweep_df.empty:
    logger.warning("Soft-DTW grid search produced no valid candidates; falling back to gamma=1.0, k=4.")
    return 1.0, 4, sweep_df, float("nan")

candidates = sweep_df[sweep_df["dominant_fraction"] <= max_dominant_frac].copy()
if candidates.empty:
    logger.warning("No (gamma, k) kept every cluster under %.0f%% dominance; relaxing guard.",
                   max_dominant_frac * 100)
    candidates = sweep_df.copy()

candidates = candidates.sort_values("silhouette",
ascending=False).head(top_n_for_stability).reset_index(drop=True)

print(f"\n[grid-search] Running stability check on top {len(candidates)} candidates...")
stabilities = []
for _, row in candidates.iterrows():
    gamma, k = row["gamma"], int(row["k"])

    def fit_fn(kk, s, _gamma=gamma):
        _m, lbl = _fit_softdtw_kmeans(traj_scaled, kk, s, gamma=_gamma)
        return lbl

    mean_ari, _ = stability_check_generic(fit_fn, k, n_seeds=n_stability_seeds)
    stabilities.append(mean_ari)
candidates["stability_mean_pairwise_ARI"] = stabilities

sil_norm = (candidates["silhouette"] - candidates["silhouette"].min()) / (
    candidates["silhouette"].max() - candidates["silhouette"].min() + 1e-9)
stab_norm = (candidates["stability_mean_pairwise_ARI"] - candidates["stability_mean_pairwise_ARI"].min()) / (
    candidates["stability_mean_pairwise_ARI"].max() - candidates["stability_mean_pairwise_ARI"].min() + 1e-9)
candidates["combined_score"] = 0.5 * sil_norm + 0.5 * stab_norm

```

```

print("\n[grid-search] Final candidate ranking (silhouette + stability combined):")
print(candidates.sort_values("combined_score", ascending=False).to_string(index=False))

best_row = candidates.sort_values("combined_score", ascending=False).iloc[0]
best_gamma, best_k = float(best_row["gamma"]), int(best_row["k"])
logger.info("Selected gamma=%.5g, k=%d by combined silhouette+stability score", best_gamma, best_k)
return best_gamma, best_k, sweep_df, float(best_row["stability_mean_pairwise_ARI"])

def run_softdtw_kmeans_clustering(traj_array, species_order, day_order, band_names, ds,
                                  seed=SEED, k_min=2, k_max=8, gamma_grid=None,
                                  max_dominant_frac=0.60, n_stability_seeds=5, make_figures=True,
                                  shared_dtw_dist=None):
    scaler = TimeSeriesScalerMeanVariance()
    traj_scaled = scaler.fit_transform(traj_array)

    best_gamma, best_k, sweep_df, pre_stability = grid_search_softdtw(
        traj_scaled, k_min=k_min, k_max=k_max, seed=seed, gamma_grid=gamma_grid,
        max_dominant_frac=max_dominant_frac, n_stability_seeds=n_stability_seeds)

    model, cluster_labels = _fit_softdtw_kmeans(traj_scaled, best_k, seed=seed, gamma=best_gamma)

    result_df = pd.DataFrame({"Species": species_order, "SoftDTW_Cluster": cluster_labels})
    print(f"\nSpecies -> Soft-DTW K-means cluster assignment (k={best_k}, gamma={best_gamma:.5g}):")
    print(result_df.sort_values("SoftDTW_Cluster").to_string(index=False))

    n_species = traj_scaled.shape[0]
    flat = traj_scaled.reshape(n_species, -1)
    report = clustering_validation_metrics(flat, cluster_labels, dtw_dist_matrix=shared_dtw_dist,

```

```

        method_name="SoftDTW_KMeans_Kinetics", data_source=ds.data_source)

report["stability_mean_pairwise_ARI"] = pre_stability

report["k_used"] = best_k

report["gamma_used"] = best_gamma


if make_figures:

    plot_cluster_trajectories(traj_array, species_order, day_order, band_names, cluster_labels,

                             title=f"Model 2: Soft-DTW K-means Cluster Trajectories (k={best_k},
gamma={best_gamma:.3g})",

                             out_stub="model2_softdtw")

    plot_silhouette_profile(flat, cluster_labels, title="Model 2: Soft-DTW K-means Silhouette Profile",

                             out_stub="model2_softdtw")


return {"result_df": result_df, "report": report, "sweep_df": sweep_df,

        "cluster_labels": cluster_labels, "flat": flat, "species_order": species_order,

        "best_gamma": best_gamma, "best_k": best_k, "cluster_col": "SoftDTW_Cluster"}


# =====

# 5. Model 3: Agglomerative Hierarchical / DTW distance

# =====


def compute_dtw_distance_matrix(traj_scaled: np.ndarray) -> np.ndarray:

    dist = cdist_dtw(traj_scaled)

    dist = (dist + dist.T) / 2.0

    np.fill_diagonal(dist, 0.0)

    return dist


def fit_hierarchical(dist_matrix, n_clusters, linkage="average"):

    model = AgglomerativeClustering(n_clusters=n_clusters, metric="precomputed", linkage=linkage)

```

```
labels = model.fit_predict(dist_matrix)

return model, labels
```

```
def grid_search_hierarchical(dist_matrix, flat_features, k_min=2, k_max=8,
                             linkages=("average", "complete", "single"), max_dominant_frac=0.60):
    n_species = dist_matrix.shape[0]
    rows = []
    for linkage in linkages:
        for k in range(k_min, min(k_max, n_species - 1) + 1):
            _model, labels = _fit_hierarchical(dist_matrix, k, linkage=linkage)
            if len(np.unique(labels)) < 2:
                continue

            sil = silhouette_score(flat_features, labels)
            db = davies_bouldin_score(flat_features, labels)
            balance = cluster_balance_report(labels)

            rows.append({"linkage": linkage, "k": k, "silhouette": sil, "davies_bouldin": db,
                        "dominant_fraction": balance["dominant_cluster_fraction"],
                        "active_clusters": balance["active_clusters"]})

    sweep_df = pd.DataFrame(rows)

    print("\n[grid-search] Hierarchical/DTW linkage x k sweep (top 15 by silhouette):")
    print(sweep_df.sort_values("silhouette", ascending=False).head(15).to_string(index=False))

    if sweep_df.empty:
        logger.warning("Hierarchical grid search produced no valid candidates; falling back to average/k=4.")
        return "average", 4, sweep_df

    candidates = sweep_df[sweep_df["dominant_fraction"] <= max_dominant_frac]
    pool = candidates if not candidates.empty else sweep_df
```

```

if candidates.empty:

    logger.warning("No (linkage, k) avoided a dominant cluster > %.0f%%; using full sweep.",
                   max_dominant_frac * 100)

best_row = pool.loc[pool["silhouette"].idxmax()]
best_linkage, best_k = best_row["linkage"], int(best_row["k"])
logger.info("Selected linkage=%s, k=%d by best silhouette (%.4f)", best_linkage, best_k, best_row["silhouette"])
return best_linkage, best_k, sweep_df


def plot_dendrogram(dist_matrix, species_order, out_stub="model3_hierarchical", linkage="average"):

    apply_plot_style()

    condensed = squareform(dist_matrix, checks=False)
    Z = scipy_linkage(condensed, method=linkage)

    fig, ax = plt.subplots(figsize=(11, 6), dpi=200)
    dendrogram(Z, labels=species_order, ax=ax, color_threshold=0.7 * max(Z[:, 2]))
    ax.set_title(f"Model 3: Hierarchical/DTW Dendrogram (linkage={linkage}, real distances)")
    ax.set_ylabel(f"DTW distance ({linkage} linkage)")
    for tick in ax.get_xticklabels():
        tick.set_rotation(90)
        tick.set_fontsize(8)
    ax.grid(True, axis="y")
    plt.tight_layout()
    plt.savefig(f"{OUTPUT_DIR}/{out_stub}_dendrogram.png", bbox_inches="tight")
    plt.close(fig)
    return Z


def run_hierarchical_dtw_clustering(traj_array, species_order, day_order, band_names, ds,

```

```

        seed=SEED, k_min=2, k_max=8,

        linkages=("average", "complete", "single"),

        max_dominant_frac=0.60, n_stability_seeds=5, make_figures=True,

        shared_dtw_dist=None):

scaler = TimeSeriesScalerMeanVariance()

traj_scaled = scaler.fit_transform(traj_array)

if shared_dtw_dist is not None:

    logger.info("Reusing shared pairwise DTW distance matrix for Model 3 (ensures identical "

                "geometry / metrics across all models).")

    dist_matrix = shared_dtw_dist
else:

    logger.info("Computing pairwise DTW distance matrix over %d species...", traj_scaled.shape[0])

    dist_matrix = compute_dtw_distance_matrix(traj_scaled)

n_species = traj_scaled.shape[0]

flat = traj_scaled.reshape(n_species, -1)

best_linkage, best_k, sweep_df = grid_search_hierarchical(

    dist_matrix, flat, k_min=k_min, k_max=k_max, linkages=linkages, max_dominant_frac=max_dominant_frac)

model, cluster_labels = _fit_hierarchical(dist_matrix, best_k, linkage=best_linkage)

result_df = pd.DataFrame({"Species": species_order, "Hierarchical_DTW_Cluster": cluster_labels})

print(f"\nSpecies -> Hierarchical/DTW cluster assignment (k={best_k}, linkage={best_linkage}):")

print(result_df.sort_values("Hierarchical_DTW_Cluster").to_string(index=False))

report = clustering_validation_metrics(flat, cluster_labels, dtw_dist_matrix=dist_matrix,

                                     method_name="Hierarchical_DTW_Kinetics", data_source=ds.data_source)

```

```

def fit_fn(k, s):

    rng = np.random.RandomState(s)

    noise = rng.normal(0, 0.02, traj_scaled.shape)

    perturbed_dist = compute_dtw_distance_matrix(traj_scaled + noise)

    _m, lbl = _fit_hierarchical(perturbed_dist, k, linkage=best_linkage)

    return lbl


mean_ari, _ = stability_check_generic(fit_fn, best_k, n_seeds=n_stability_seeds)

report["stability_mean_pairwise_ARI"] = mean_ari

report["stability_note"] = "agglomerative is deterministic; stability measured via input noise perturbation"

report["k_used"] = best_k

report["linkage_used"] = best_linkage


if make_figures:

    plot_cluster_trajectories(traj_array, species_order, day_order, band_names, cluster_labels,

                             title=f"Model 3: Hierarchical/DTW Cluster Trajectories (k={best_k},
linkage={best_linkage})",

                             out_stub="model3_hierarchical")

    plot_silhouette_profile(flat, cluster_labels, title="Model 3: Hierarchical/DTW Silhouette Profile",

                             out_stub="model3_hierarchical")

    plot_dendrogram(dist_matrix, species_order, linkage=best_linkage)


return {"result_df": result_df, "report": report, "sweep_df": sweep_df, "dist_matrix": dist_matrix,

        "cluster_labels": cluster_labels, "flat": flat, "species_order": species_order,

        "best_linkage": best_linkage, "best_k": best_k, "cluster_col": "Hierarchical_DTW_Cluster"}


# =====

# 6. Consensus / co-association across all 3 models

# =====

```

```

def build_coassociation_matrix(species_order, *label_sets):

    n = len(species_order)

    sp_idx = {s: i for i, s in enumerate(species_order)}

    coassoc = np.zeros((n, n))

    n_models = 0

    for sp_order_i, labels_i in label_sets:

        aligned = np.full(n, -1)

        for sp, lbl in zip(sp_order_i, labels_i):

            aligned[sp_idx[sp]] = lbl

        coassoc += (aligned[:, None] == aligned[None, :]).astype(float)

        n_models += 1

    coassoc /= n_models

    np.fill_diagonal(coassoc, 1.0)

    return coassoc


def select_consensus_k(coassoc, shared_dtw_dist, k_min=2, k_max=8, linkage="average",
                       max_dominant_frac=0.60):

    n = coassoc.shape[0]

    dist = 1.0 - coassoc

    np.fill_diagonal(dist, 0.0)

    condensed = squareform(dist, checks=False)

    Z = scipy_linkage(condensed, method=linkage)

    rows = []

    for k in range(k_min, min(k_max, n - 1) + 1):

        labels = fcluster(Z, t=k, criterion="maxclust")

        if len(np.unique(labels)) < 2:

            continue

        sil_dtw = float(silhouette_score(shared_dtw_dist, labels, metric="precomputed"))

```

```

balance = cluster_balance_report(labels)

per_species_agreement = []
n_singletons = 0
for i in range(n):
    members = np.where(labels == labels[i])[0]
    mates = members[members != i]
    if len(mates) > 0:
        per_species_agreement.append(float(coassoc[i, mates].mean()))
    else:
        per_species_agreement.append(0.0)
    n_singletons += 1
mean_species_agreement = float(np.mean(per_species_agreement))

rows.append({"k": k, "silhouette_dtw": sil_dtw,
            "mean_species_agreement": mean_species_agreement,
            "n_singletons": n_singletons,
            "dominant_fraction": balance["dominant_cluster_fraction"],
            "active_clusters": balance["active_clusters"]})

sweep_df = pd.DataFrame(rows)
print("\n[consensus k-selection] sweep across k (DTW silhouette + species-level agreement + balance):")
print(sweep_df.to_string(index=False))

if sweep_df.empty:
    logger.warning("Consensus k-sweep produced no valid candidates; falling back to k=4.")
    return 4, sweep_df, Z

ok = sweep_df[sweep_df["dominant_fraction"] <= max_dominant_frac]
pool = ok if not ok.empty else sweep_df

```

```

if ok.empty:

    logger.warning("No consensus k avoided a dominant cluster > %.0f%%; using full sweep.",
                   max_dominant_frac * 100)

sil_range = pool["silhouette_dtw"].max() - pool["silhouette_dtw"].min()
agree_range = pool["mean_species_agreement"].max() - pool["mean_species_agreement"].min()
sil_norm = (pool["silhouette_dtw"] - pool["silhouette_dtw"].min()) / (sil_range + 1e-9)
agree_norm = (pool["mean_species_agreement"] - pool["mean_species_agreement"].min()) / (agree_range + 1e-9)

pool = pool.assign(combined_score=(0.5 * sil_norm + 0.5 * agree_norm))

print("\n[consensus k-selection] ranked candidates (combined_score = "
      "0.5*silhouette_dtw + 0.5*mean_species_agreement, balance-filtered):")
print(pool.sort_values("combined_score", ascending=False).to_string(index=False))

best_k = int(pool.loc[pool["combined_score"].idxmax(), "k"])
logger.info("Selected consensus k=%d objectively (DTW silhouette + species-level "
            "agreement + balance guard). Check n_singletons in the sweep above before "
            "trusting this at face value in a small-n (n=%d species) regime.", best_k, n)
return best_k, sweep_df, Z

```

```

def consensus_cluster(coassoc, species_order, n_clusters=None, linkage="average",
                     distance_threshold_frac=0.5, out_stub="consensus", precomputed_Z=None):
    if precomputed_Z is not None:
        Z = precomputed_Z
    else:
        dist = 1.0 - coassoc
        np.fill_diagonal(dist, 0.0)
        condensed = squareform(dist, checks=False)
        Z = scipy_linkage(condensed, method=linkage)

```

```

if n_clusters is not None:
    labels = fcluster(Z, t=n_clusters, criterion="maxclust")
else:
    labels = fcluster(Z, t=1.0 - distance_threshold_frac, criterion="distance")

result_df = pd.DataFrame({"Species": species_order, "Consensus_Cluster": labels})
print(f"\nSpecies -> CONSENSUS cluster assignment ({len(np.unique(labels))} clusters):")
print(result_df.sort_values("Consensus_Cluster").to_string(index=False))

apply_plot_style()
fig, ax = plt.subplots(figsize=(11, 6), dpi=200)
dendrogram(Z, labels=species_order, ax=ax, color_threshold=0.7 * max(Z[:, 2]) if len(Z) else None)
ax.set_title("Consensus Dendrogram \u2014 agreement across DTW / Soft-DTW / Hierarchical-DTW")
ax.set_ylabel("1 - co-association (cross-model disagreement)")
for tick in ax.get_xticklabels():
    tick.set_rotation(90)
    tick.set_fontsize(8)
ax.grid(True, axis="y")
plt.tight_layout()
plt.savefig(f"{OUTPUT_DIR}/{out_stub}_dendrogram.png", bbox_inches="tight")
plt.close(fig)

return result_df, Z, labels

```

```

def print_consensus_interpretation(coassoc, species_order, result_df):
    n = len(species_order)
    unanimous_pairs, contested_pairs = [], []
    for i in range(n):

```

```

for j in range(i + 1, n):
    v = coassoc[i, j]
    if v >= 0.999:
        unanimous_pairs.append((species_order[i], species_order[j]))
    elif 0.30 < v < 0.70:
        contested_pairs.append((species_order[i], species_order[j], v))

print("\n=== Consensus interpretation ===")
print(f"Species pairs ALL 3 models agree belong together ( {len(unanimous_pairs)} pairs):")
for a, b in unanimous_pairs[:30]:
    print(f"    {a} -- {b}")
if len(unanimous_pairs) > 30:
    print(f"    ... and {len(unanimous_pairs) - 30} more")

print(f"\nContested pairs (co-association 0.3-0.7, {len(contested_pairs)} pairs) -- treat with caution:")
for a, b, v in sorted(contested_pairs, key=lambda t: -t[2])[:20]:
    print(f"    {a} -- {b}: agreement={v:.2f}")

print(f"\nFinal consensus cluster sizes:\n{result_df['Consensus_Cluster'].value_counts().to_string()}")
print(
    "\nInterpretation guide: species pairs with co-association = 1.0 are your strongest "
    "biological-similarity claims (independently reproduced by 3 different clustering "
    "mechanisms on the same kinetics data). Contested-band pairs are NOT reliable evidence "
    "either way -- report the consensus clusters as your headline result, and use the "
    "per-model clusters (Models 1-3) as supporting/ablation evidence. Internal-validity "
    "numbers (silhouette/Davies-Bouldin) tell you the clusters are well-separated; they do "
    "NOT by themselves tell you the clusters are biologically meaningful -- for that, see "
    "the independent taxonomy / pigment / growth-curve validation report below."
)

```

```

# =====
# 6a. Independent biological validation (taxonomy / pigments / growth curves)
# =====

def validate_against_taxonomy(cluster_df: pd.DataFrame, cluster_col: str,
                             taxonomy: dict = SPECIES_TAXONOMY):
    df = cluster_df.copy()
    df["Taxon"] = df["Species"].map(taxonomy)
    missing = df.loc[df["Taxon"].isna(), "Species"].tolist()
    if missing:
        logger.warning("No SPECIES_TAXONOMY entry for: %s -- excluded from taxonomy validation.", missing)
    df = df.dropna(subset=["Taxon"])
    if df["Taxon"].nunique() < 2 or df[cluster_col].nunique() < 2:
        logger.warning("Not enough distinct taxa/clusters to compute a meaningful taxonomy ARI/NMI.")
    return {"ari_vs_taxonomy": None, "nmi_vs_taxonomy": None, "crosstab": None, "n_species": len(df)}

    taxon_encoded = LabelEncoder().fit_transform(df["Taxon"])
    ari = adjusted_rand_score(taxon_encoded, df[cluster_col])
    nmi = normalized_mutual_info_score(taxon_encoded, df[cluster_col])
    crosstab = pd.crosstab(df["Taxon"], df[cluster_col])
    return {"ari_vs_taxonomy": float(ari), "nmi_vs_taxonomy": float(nmi),
            "crosstab": crosstab, "n_species": len(df)}

# FIX #4 (small-sample guard): Kruskal-Wallis with any group smaller than
# this is unreliable at n=20-species scale -- one point can flip the result.
# Below this we still run the test (so nothing is silently hidden) but flag
# it so it isn't read with the same confidence as a well-powered comparison.
MIN_GROUP_SIZE_FOR_RELIABLE_TEST = 3

```

```

def _min_group_size(groups):
    return min((len(g) for g in groups), default=0)

# FIX #3 (multiple comparisons): plain Benjamini-Hochberg FDR correction,
# implemented directly (no statsmodels dependency) so it runs anywhere numpy/
# pandas already run. Returns q-values in the same order as the input p's;
# NaN/None p-values pass through as NaN and are excluded from the correction.
def benjamini_hochberg(p_values):
    p = np.asarray([np.nan if v is None else v for v in p_values], dtype=float)
    n = np.sum(~np.isnan(p))
    q = np.full_like(p, np.nan)
    if n == 0:
        return q
    valid_idx = np.where(~np.isnan(p))[0]
    order = valid_idx[np.argsort(p[valid_idx])]
    ranked_p = p[order]
    ranks = np.arange(1, n + 1)
    raw_q = ranked_p * n / ranks
    # enforce monotonicity (standard BH step-up correction)
    raw_q = np.minimum.accumulate(raw_q[::-1])[::-1]
    raw_q = np.clip(raw_q, 0, 1)
    q[order] = raw_q
    return q

def validate_against_pigments(ds: MicroalgaeROIDataset, cluster_df: pd.DataFrame, cluster_col: str):
    meta = ds.meta_dataframe()

```

```

pig_df = pd.DataFrame(ds.pigment_indices, columns=["NDVI", "GNDVI", "Simple_Ratio"])
pig_df["Species"] = meta["Species"].values
species_pig = pig_df.groupby("Species").mean().reset_index()
merged = species_pig.merge(cluster_df[["Species", "cluster_col"]], on="Species", how="inner")

test_results = {}

for pig_name in ["NDVI", "GNDVI", "Simple_Ratio"]:
    groups = [g[pig_name].dropna().values for _, g in merged.groupby("cluster_col")]
    groups = [g for g in groups if len(g) > 0]
    if len(groups) > 1:
        try:
            stat, p = kruskal(*groups)
        except ValueError:
            stat, p = np.nan, np.nan
    else:
        stat, p = np.nan, np.nan
    min_n = _min_group_size(groups)
    if min_n and min_n < MIN_GROUP_SIZE_FOR_RELIABLE_TEST:
        logger.warning(
            "Pigment test '%s': smallest cluster has only %d species (<%d) -- "
            "treat this p-value as suggestive, not confirmatory.",
            pig_name, min_n, MIN_GROUP_SIZE_FOR_RELIABLE_TEST)
    test_results[pig_name] = {
        "kruskal_H": None if np.isnan(stat) else float(stat),
        "p_value": None if np.isnan(p) else float(p),
        "min_group_n": min_n,
        "low_power": bool(min_n and min_n < MIN_GROUP_SIZE_FOR_RELIABLE_TEST),
    }

summary_table = merged.groupby("cluster_col")[["NDVI", "GNDVI", "Simple_Ratio"]].agg(["mean", "std"])

```

```
return test_results, summary_table, merged
```

```
def diagnose_band_growth_variance(traj_array, band_names):  
    rows = []  
    for b_idx, band in enumerate(band_names):  
        per_species_max = traj_array[:, :, b_idx].max(axis=1)  
        rows.append({  
            "band": band,  
            "across_species_std_of_max": float(np.std(per_species_max)),  
            "across_species_range_of_max": float(np.ptp(per_species_max)),  
            "min_value_seen": float(traj_array[:, :, b_idx].min()),  
            "max_value_seen": float(traj_array[:, :, b_idx].max()),  
        })  
    return pd.DataFrame(rows).sort_values("across_species_std_of_max", ascending=False).reset_index(drop=True)
```

```
def select_growth_band_auto(traj_array, band_names, min_relative_std=1e-9):  
    diag_df = diagnose_band_growth_variance(traj_array, band_names)  
    best = diag_df.iloc[0]  
    if best["across_species_std_of_max"] <= min_relative_std:  
        logger.warning(  
            "GROWTH-CURVE VALIDATION WARNING: every candidate band has ~zero across-species "  
            "variance in its per-species peak value (best band '%s' has std=%.3g of max). "  
            "Growth-curve descriptors will be uninformative (constant across all species) "  
            "regardless of which band is chosen. Full per-band diagnostic:\n%s",  
            best["band"], best["across_species_std_of_max"], diag_df.to_string(index=False),  
        )  
    else:  
        logger.info("Auto-selected growth_band='%s' (highest across-species variance in peak "
```

```

        "value; std_of_max=%.4g). Full per-band diagnostic:\n%s",
        best["band"], best["across_species_std_of_max"], diag_df.to_string(index=False))
    return best["band"], diag_df

```

```

def compute_growth_curve_features(traj_array, species_order, day_order, band_names,
                                growth_band="auto"):
    if growth_band == "auto":
        growth_band, diag_df = _select_growth_band_auto(traj_array, band_names)
    else:
        diag_df = diagnose_band_growth_variance(traj_array, band_names)

    b_idx = band_names.index(growth_band)
    days = np.asarray(day_order, dtype=float)
    mid = len(days) // 2
    rows = []
    for i, sp in enumerate(species_order):
        y = traj_array[i, :, b_idx]
        auc = float(_trapz(y, days))
        max_val = float(y.max())
        day_of_max = float(days[int(np.argmax(y))])
        early_slope = float((y[mid] - y[0]) / max(days[mid] - days[0], 1e-6)) if mid > 0 else np.nan
        late_slope = float((y[-1] - y[mid]) / max(days[-1] - days[mid], 1e-6)) if mid < len(days) - 1 else np.nan
        rows.append({"Species": sp, "AUC": auc, "Max": max_val, "Day_of_Max": day_of_max,
                    "Early_Slope": early_slope, "Late_Slope": late_slope})
    growth_df = pd.DataFrame(rows)
    growth_df.attrs["growth_band_used"] = growth_band
    growth_df.attrs["band_diagnostics"] = diag_df
    return growth_df

```

```

def validate_against_growth_curves(traj_array, species_order, day_order, band_names,
                                   cluster_df: pd.DataFrame, cluster_col: str,
                                   growth_band="auto"):
    growth_df = compute_growth_curve_features(traj_array, species_order, day_order, band_names, growth_band)
    band_used = growth_df.attrs.get("growth_band_used", growth_band)
    merged = growth_df.merge(cluster_df[["Species", cluster_col]], on="Species", how="inner")

    feats = ["AUC", "Max", "Day_of_Max", "Early_Slope", "Late_Slope"]
    test_results = {}
    for feat in feats:
        overall_std = merged[feat].std()
        if not np.isfinite(overall_std) or overall_std < 1e-9:
            logger.warning(
                "Growth feature '%s' (band=%s) is ~constant across ALL species (std=%0.3g) -- "
                "Kruskal-Wallis would not be meaningful here; marking as 'degenerate' instead "
                "of reporting a bare NA.", feat, band_used, overall_std,
            )
            test_results[feat] = {"kruskal_H": None, "p_value": None, "degenerate": True}
            continue

    groups = [g[feat].dropna().values for _, g in merged.groupby(cluster_col)]
    groups = [g for g in groups if len(g) > 0]
    if len(groups) > 1:
        try:
            stat, p = kruskal(*groups)
        except ValueError:
            stat, p = np.nan, np.nan
    else:
        stat, p = np.nan, np.nan

```

```

min_n = _min_group_size(groups)

if min_n and min_n < MIN_GROUP_SIZE_FOR_RELIABLE_TEST:
    logger.warning(
        "Growth-curve test '%s': smallest cluster has only %d species (<%d) -- "
        "treat this p-value as suggestive, not confirmatory.",
        feat, min_n, MIN_GROUP_SIZE_FOR_RELIABLE_TEST)

test_results[feat] = {
    "kruskal_H": None if np.isnan(stat) else float(stat),
    "p_value": None if np.isnan(p) else float(p),
    "degenerate": False,
    "min_group_n": min_n,
    "low_power": bool(min_n and min_n < MIN_GROUP_SIZE_FOR_RELIABLE_TEST),
}

summary_table = merged.groupby(cluster_col)[feats].agg(["mean", "std"])

return test_results, summary_table, merged, growth_df


def biological_validation_suite(cluster_df, cluster_col, ds, traj_array, species_order,
                               day_order, band_names, label=""):
    print(f"\n--- Biological validation: {label} (cluster column '{cluster_col}') ---")

    tax_res = validate_against_taxonomy(cluster_df, cluster_col)

    if tax_res["ari_vs_taxonomy"] is not None:
        print(f" Taxonomy agreement: ARI={tax_res['ari_vs_taxonomy']:.3f} "
              f"NMI={tax_res['nmi_vs_taxonomy']:.3f} (n={tax_res['n_species']} species)")
        print(tax_res["crosstab"].to_string())
    else:
        print(" Taxonomy agreement: not computable (see warning above).")

```

```
pig_res, pig_summary, _ = validate_against_pigments(ds, cluster_df, cluster_col)
```

```
print(" Pigment-index separation across clusters (Kruskal-Wallis):")
```

```
for name, r in pig_res.items():
```

```
    h_str = f"{r['kruskal_H']:.3f}" if r["kruskal_H"] is not None else "NA"
```

```
    p_str = f"{r['p_value']:.4f}" if r["p_value"] is not None else "NA"
```

```
    flag = " [low n, smallest cluster < 3]" if r.get("low_power") else ""
```

```
    print(f"    {name:14s} H={h_str} p={p_str} {flag}")
```

```
print(pig_summary.to_string())
```

```
# FIX #2 (circularity, part B): force the pre-registered held-out band
```

```
# instead of "auto". "auto" would happily re-select a band that WAS used
```

```
# for clustering (e.g. Lipid_CH_930), silently reintroducing the same
```

```
# circularity the held-out split in main() was meant to remove.
```

```
growth_res, growth_summary, _, _ = validate_against_growth_curves(
```

```
    traj_array, species_order, day_order, band_names, cluster_df, cluster_col,
```

```
    growth_band=GROWTH_VALIDATION_BAND)
```

```
print(" Growth-curve descriptor separation across clusters (Kruskal-Wallis):")
```

```
for name, r in growth_res.items():
```

```
    if r.get("degenerate"):
```

```
        print(f"    {name:14s} DEGENERATE -- constant across all species in this band (see warning above)")
```

```
        continue
```

```
    h_str = f"{r['kruskal_H']:.3f}" if r["kruskal_H"] is not None else "NA"
```

```
    p_str = f"{r['p_value']:.4f}" if r["p_value"] is not None else "NA"
```

```
    flag = " [low n, smallest cluster < 3]" if r.get("low_power") else ""
```

```
    print(f"    {name:14s} H={h_str} p={p_str} {flag}")
```

```
print(growth_summary.to_string())
```

```
flat_row = {
```

```
    "method": label,
```

```
    "ari_vs_taxonomy": tax_res["ari_vs_taxonomy"],
```

```

    "nmi_vs_taxonomy": tax_res["nmi_vs_taxonomy"],
}

for name, r in pig_res.items():
    flat_row[f'pigment_{name}_p'] = r["p_value"]

for name, r in growth_res.items():
    flat_row[f'growth_{name}_p'] = r["p_value"]

# FIX #3 (multiple comparisons): this method's biological-validation row
# runs 3 pigment tests + 5 growth tests = 8 hypothesis tests. Reporting 8
# raw p-values with no correction means ~0.4 false positives expected at
# alpha=0.05 even under the null. Add BH-FDR q-values across exactly the
# set of tests being reported together, and print which survive  $q < 0.05$ .
p_names = [k for k in flat_row if k.endswith("_p") and k not in ("method",)]
q_values = benjamini_hochberg([flat_row[k] for k in p_names])

for name, q in zip(p_names, q_values):
    flat_row[name.replace("_p", "_q_fdr")] = None if np.isnan(q) else float(q)

surviving = [n for n, q in zip(p_names, q_values) if not np.isnan(q) and q < 0.05]

print(f" FDR-corrected (Benjamini-Hochberg, {len(p_names)} tests): "
      f"{len(surviving)} of {len(p_names)} survive  $q < 0.05$ "
      + (f" -> {' '.join(s.replace('_', '') for s in surviving)}" if surviving else ""))

return {
    "taxonomy": tax_res, "pigment_tests": pig_res, "pigment_summary": pig_summary,
    "growth_tests": growth_res, "growth_summary": growth_summary, "flat_row": flat_row,
}

# =====

# 7. Plotting

```

```
# =====
```

```
PALETTE = {"olive": "#494623", "ochre": "#c3892b", "sage": "#8e883d",  
           "brown": "#8f7c2f", "cream": "#e8e7d8", "ivory": "#fbfbfa", "white": "#ffffff"}  
EXTRA_COLORS = ["#5b7a8c", "#a35d5d", "#6b5b95", "#5f6b3e", "#b5651d", "#4a6670"]  
OUTPUT_DIR = "outputs"
```

```
def apply_plot_style():  
    plt.rcParams.update({  
        "font.family": "serif",  
        "font.serif": ["Book Antiqua", "Palatino", "Palatino Linotype", "Georgia", "DejaVu Serif"],  
        "axes.edgecolor": PALETTE["brown"], "axes.linewidth": 1.2,  
        "axes.facecolor": PALETTE["ivory"], "figure.facecolor": PALETTE["white"],  
        "grid.color": PALETTE["cream"], "grid.linestyle": "--", "grid.alpha": 0.6,  
        "xtick.color": PALETTE["olive"], "ytick.color": PALETTE["olive"],  
        "xtick.labelsize": 9, "ytick.labelsize": 9,  
        "axes.labelsize": 10, "axes.titlesize": 12, "legend.fontsize": 8,  
    })  
    os.makedirs(OUTPUT_DIR, exist_ok=True)  
  
def plot_cluster_trajectories(traj_array, species_order, day_order, band_names, cluster_labels, title, out_stub):  
    apply_plot_style()  
    cluster_labels = np.asarray(cluster_labels)  
    unique_clusters = sorted(np.unique(cluster_labels))  
    colors = [PALETTE["olive"], PALETTE["ochre"], PALETTE["sage"], PALETTE["brown"]] +  
    EXTRA_COLORS  
  
    n_bands = len(band_names)  
    ncols = (n_bands + 1) // 2
```

```
fig, axes = plt.subplots(2, ncols, figsize=(4.2 * ncols, 7), dpi=200)
```

```
axes = np.atleast_1d(axes).ravel()
```

```
for b_i, band in enumerate(band_names):
```

```
    ax = axes[b_i]
```

```
    for c_i, c in enumerate(unique_clusters):
```

```
        member_idx = np.where(cluster_labels == c)[0]
```

```
        member_traj = traj_array[member_idx, :, b_i]
```

```
        mean_traj, std_traj = member_traj.mean(axis=0), member_traj.std(axis=0)
```

```
        color = colors[c_i % len(colors)]
```

```
        ax.plot(day_order, mean_traj, color=color, linewidth=2.2, label=f"Cluster {c} (n={len(member_idx)})")
```

```
        ax.fill_between(day_order, mean_traj - std_traj, mean_traj + std_traj, color=color, alpha=0.15)
```

```
    ax.set_title(band, fontsize=10)
```

```
    ax.set_xlabel("Day")
```

```
    ax.grid(True)
```

```
    if b_i == 0:
```

```
        ax.legend(fontsize=7, loc="best")
```

```
for j in range(n_bands, len(axes)):
```

```
    axes[j].axis("off")
```

```
fig.suptitle(title, fontsize=13)
```

```
plt.tight_layout()
```

```
plt.savefig(f"{OUTPUT_DIR}/{out_stub}_trajectories.png", bbox_inches="tight")
```

```
plt.close(fig)
```

```
def plot_silhouette_profile(flat_features, cluster_labels, title, out_stub):
```

```
    apply_plot_style()
```

```
    cluster_labels = np.asarray(cluster_labels)
```

```

unique_clusters = sorted(np.unique(cluster_labels))

if len(unique_clusters) < 2:
    logger.warning("Only one cluster present; skipping silhouette plot for %s", title)
    return

sample_scores = silhouette_samples(flat_features, cluster_labels)

avg_score = sample_scores.mean()

colors = [PALETTE["olive"], PALETTE["ochre"], PALETTE["sage"], PALETTE["brown"]] +
EXTRA_COLORS

fig, ax = plt.subplots(figsize=(7, 5.5), dpi=200)

y_lower = 10

for c_i, c in enumerate(unique_clusters):
    c_scores = np.sort(sample_scores[cluster_labels == c])[:-1]
    y_upper = y_lower + len(c_scores)
    color = colors[c_i % len(colors)]
    ax.fill_betweenx(np.arange(y_lower, y_upper), 0, c_scores, facecolor=color, label=f"Cluster {c}")
    ax.text(-0.05, y_lower + len(c_scores) / 2, f"C {c}", ha="right", va="center",
           color=PALETTE["olive"], weight="bold", fontsize=9)
    y_lower = y_upper + 5

ax.axvline(x=avg_score, color=PALETTE["ochre"], linestyle="--", linewidth=2,
          label=f"Avg Silhouette: {avg_score:.3f}")

ax.set_title(title, pad=15)
ax.set_xlabel("Silhouette Coefficient ($s_i$)")
ax.set_ylabel("Species (real samples)")
ax.set_xlim(min(-0.2, sample_scores.min() - 0.05), 1.0)
ax.set_ylim(0, y_lower)
ax.yaxis.set_visible(False)
ax.grid(True, axis="x")
ax.legend(loc="upper right", fontsize=8)

```

```
plt.tight_layout()

plt.savefig(f'{OUTPUT_DIR}/{out_stub}_silhouette.png', bbox_inches="tight")

plt.close(fig)
```

```
def plot_coassociation_heatmap(coassoc, species_order, out_stub="consensus"):

    apply_plot_style()

    n = len(species_order)

    fig, ax = plt.subplots(figsize=(0.35 * n + 2, 0.35 * n + 2), dpi=200)

    im = ax.imshow(coassoc, cmap="YlOrBr", vmin=0, vmax=1)

    ax.set_xticks(range(n)); ax.set_xticklabels(species_order, rotation=90, fontsize=7)

    ax.set_yticks(range(n)); ax.set_yticklabels(species_order, fontsize=7)

    ax.set_title("Cross-Model Co-Association (fraction of 3 models agreeing)")

    cbar = fig.colorbar(im, ax=ax, fraction=0.046, pad=0.04)

    cbar.set_label("Agreement fraction")

    plt.tight_layout()

    plt.savefig(f'{OUTPUT_DIR}/{out_stub}_coassociation_heatmap.png', bbox_inches="tight")

    plt.close(fig)
```

```
# =====

# 9. Publication-ready interpretation figures (manuscript figure guide)

# =====
```

```
PUB_FONT_STACK = ["Book Antiqua", "Palatino", "Palatino Linotype", "Georgia", "DejaVu Serif"]

PUB_COLORS = {

    "olive": "#494623",

    "ochre": "#c3892b",

    "sage": "#8e883d",

    "brown": "#8f7c2f",
```

```

    "cream": "#e8e7d8",
    "white": "#ffffff",
}
PUB_CYCLE = [PUB_COLORS["olive"], PUB_COLORS["ochre"], PUB_COLORS["sage"],
PUB_COLORS["brown"]]
PUB_HATCHES = [ "", "/", "xx", "..", "\\\\", "++", "oo", "***"]
PUB_DPI = 400

```

```

def _apply_pub_style():
    plt.rcParams.update({
        "font.family": "serif",
        "font.serif": PUB_FONT_STACK,
        "figure.facecolor": PUB_COLORS["white"],
        "axes.facecolor": PUB_COLORS["white"],
        "savefig.facecolor": PUB_COLORS["white"],
        "axes.edgecolor": PUB_COLORS["olive"],
        "axes.labelcolor": PUB_COLORS["olive"],
        "text.color": PUB_COLORS["olive"],
        "xtick.color": PUB_COLORS["olive"],
        "ytick.color": PUB_COLORS["olive"],
        "axes.linewidth": 1.1,
        "grid.color": PUB_COLORS["cream"],
        "grid.linestyle": "--",
        "grid.alpha": 0.9,
        "axes.titlesize": 11.5,
        "axes.labelsize": 10,
        "legend.fontsize": 8.5,
        "xtick.labelsize": 8.5,
        "ytick.labelsize": 8.5,
    })

```

```
os.makedirs(OUTPUT_DIR, exist_ok=True)
```

```
def _save_pub_figure(fig, stub):  
    for ax in fig.get_axes():  
        for spine in ax.spines.values():  
            spine.set_color(PUB_COLORS["olive"])  
    fig.savefig(f'{OUTPUT_DIR}/{stub}.png', dpi=PUB_DPI, bbox_inches="tight",  
                facecolor=PUB_COLORS["white"])  
    fig.savefig(f'{OUTPUT_DIR}/{stub}.pdf', bbox_inches="tight", facecolor=PUB_COLORS["white"])  
    plt.close(fig)  
    logger.info("Saved publication figure: %s.png (%d dpi) / %s.pdf", stub, PUB_DPI, stub)
```

```
def _cluster_color(i):  
    i = int(i)  
    color = PUB_CYCLE[i % len(PUB_CYCLE)]  
    hatch = PUB_HATCHES[(i // len(PUB_CYCLE)) % len(PUB_HATCHES)]  
    return color, hatch
```

```
def fig1_algorithm_mechanics(gamma_used, out_stub="fig1_algorithm_mechanics"):  
    _apply_pub_style()  
    fig, axes = plt.subplots(1, 3, figsize=(13, 4.4), dpi=PUB_DPI)  
  
    x = np.linspace(0, 10, 60)  
    curve_a = np.sin(x / 2) + 0.15 * x  
    curve_b = np.sin((x - 1.4) / 2) + 0.15 * (x - 1.4) + 0.15  
  
    ax = axes[0]
```

```

ax.plot(x, curve_a, color=PUB_COLORS["olive"], linewidth=2.2, label="Species A")
ax.plot(x, curve_b, color=PUB_COLORS["ochre"], linewidth=2.2, label="Species B")
for i in range(0, len(x), 6):
    ax.plot([x[i], x[i]], [curve_a[i], curve_b[i]], color=PUB_COLORS["sage"],
            linewidth=0.9, alpha=0.8)

ax.set_title("Model 1: Hard-DTW\nsingle minimum-cost alignment path")
ax.set_xlabel("Cultivation day (schematic)")
ax.set_ylabel("Band signal (a.u., schematic)")
ax.legend(frameon=False, loc="upper left")
ax.grid(True)

ax = axes[1]
ax.plot(x, curve_a, color=PUB_COLORS["olive"], linewidth=2.2)
ax.plot(x, curve_b, color=PUB_COLORS["ochre"], linewidth=2.2)
for i in range(0, len(x), 5):
    for j in range(max(0, i - 3), min(len(x), i + 4)):
        ax.plot([x[i], x[j]], [curve_a[i], curve_b[j]], color=PUB_COLORS["brown"],
                linewidth=0.4, alpha=0.18)

ax.set_title(f"Model 2: Soft-DTW ( $\gamma=\{\gamma_{used:.3g}\}$ )\nweighted expectation over all paths")
ax.set_xlabel("Cultivation day (schematic)")
ax.grid(True)

ax = axes[2]
ax.plot(x, curve_a, color=PUB_COLORS["olive"], linewidth=2.2)
ax.plot(x, curve_b, color=PUB_COLORS["ochre"], linewidth=2.2)
worst_i = int(np.argmax(np.abs(curve_a - curve_b)))
ax.plot([x[worst_i], x[worst_i]], [curve_a[worst_i], curve_b[worst_i]],
        color=PUB_COLORS["sage"], linewidth=2.6)
ax.annotate("complete-linkage:\nmax pairwise distance",
           xy=(x[worst_i], (curve_a[worst_i] + curve_b[worst_i]) / 2),

```

```

        xytext=(0.42, 0.12), textcoords="axes fraction", fontsize=8,
        arrowprops=dict(arrowstyle="->", color=PUB_COLORS["olive"], lw=1))
ax.set_title("Model 3: Hierarchical/DTW\ncomplete-linkage distance")
ax.set_xlabel("Cultivation day (schematic)")
ax.grid(True)

fig.suptitle("Figure 1 \u2014 Algorithmic mechanics of the three DTW clustering strategies", y=1.04)
plt.tight_layout()
_save_pub_figure(fig, out_stub)

def _sankey_node_layout(labels, pad=0.8):
    counts = pd.Series(labels).value_counts().sort_index()
    layout = {}
    cursor = 0.0
    for cluster_id, n in counts.items():
        layout[int(cluster_id)] = (cursor, cursor + n)
        cursor += n + pad
    total_height = cursor - pad if len(counts) else 0.0
    return layout, dict(counts), total_height

def _sankey_ribbon(ax, x0, x1, y0_top, y0_bot, y1_top, y1_bot, color, alpha=0.65, n=60):
    xs = np.linspace(x0, x1, n)
    t = (xs - x0) / (x1 - x0)
    s = t * t * (3 - 2 * t) # smoothstep easing
    top = y0_top + (y1_top - y0_top) * s
    bot = y0_bot + (y1_bot - y0_bot) * s
    ax.fill_between(xs, top, bot, color=color, alpha=alpha, linewidth=0, zorder=1)

```

```

def fig2_cluster_flow(species_order, res_m1, res_m2, res_m3, consensus_df,
                      out_stub="fig2_cluster_flow"):
    _apply_pub_style()

def _labels_for(res):
    d = dict(zip(res["species_order"], np.asarray(res["cluster_labels"])))
    return [int(d[s]) for s in species_order]

cons_map = dict(zip(consensus_df["Species"], consensus_df["Consensus_Cluster"]))

stages = [
    _labels_for(res_m1),
    _labels_for(res_m2),
    _labels_for(res_m3),
    [int(cons_map[s]) for s in species_order],
]

stage_prefixes = ["M1", "M2", "M3", "CN"]
stage_titles = [f"Model 1: Hard-DTW (k={len(set(stages[0]))})",
                 f"Model 2: Soft-DTW (k={len(set(stages[1]))})",
                 f"Model 3: Hierarchical (k={len(set(stages[2]))})",
                 f"Consensus (k={len(set(stages[3]))})"]

n_species = len(species_order)
node_width = 0.22
gap = 2.6
x_pos = [i * gap for i in range(4)]

layouts, counts_per_stage, heights = [], [], []
for labels in stages:
    layout, counts, h = _sankey_node_layout(labels)

```

```

layouts.append(layout)

counts_per_stage.append(counts)

heights.append(h)

max_height = max(heights) if heights else n_species

fig_h = max(6.5, max_height * 0.42)

fig, ax = plt.subplots(figsize=(15, fig_h), dpi=PUB_DPI)

for s in range(3):

    labels_l, labels_r = stages[s], stages[s + 1]

    layout_l, layout_r = layouts[s], layouts[s + 1]

    trans = {}

    for i in range(n_species):

        key = (labels_l[i], labels_r[i])

        trans[key] = trans.get(key, 0) + 1

    left_cursor = {c: layout_l[c][0] for c in layout_l}

    right_cursor = {c: layout_r[c][0] for c in layout_r}

    for (c_l, c_r) in sorted(trans.keys(), key=lambda k: (k[0], k[1])):

        flow_n = trans[(c_l, c_r)]

        y0_top = left_cursor[c_l]

        y0_bot = y0_top + flow_n

        left_cursor[c_l] = y0_bot

        y1_top = right_cursor[c_r]

        y1_bot = y1_top + flow_n

        right_cursor[c_r] = y1_bot

```

```

color, _hatch = _cluster_color(c_l)

_sankey_ribbon(ax, x_pos[s] + node_width, x_pos[s + 1] - node_width,
               y0_top, y0_bot, y1_top, y1_bot, color=color, alpha=0.55)

for s_idx, (layout, counts) in enumerate(zip(layouts, counts_per_stage)):
    for cluster_id, (y_top, y_bot) in layout.items():
        color, hatch = _cluster_color(cluster_id)
        ax.add_patch(mpatches.FancyBboxPatch(
            (x_pos[s_idx] - node_width, y_top), node_width * 2, (y_bot - y_top),
            boxstyle="square,pad=0.0", linewidth=1.3,
            edgecolor=PUB_COLORS["olive"], facecolor=color, hatch=hatch,
            alpha=0.95, zorder=3))
        n_here = counts[cluster_id]
        label = f'{{stage_prefixes[s_idx]}}-C{{cluster_id}}\n(n={{n_here}})'
        ax.text(x_pos[s_idx], (y_top + y_bot) / 2, label, ha="center", va="center",
                fontsize=7.6, fontweight="bold", color=PUB_COLORS["white"], zorder=4)

for s_idx, title in enumerate(stage_titles):
    ax.text(x_pos[s_idx], -1.4, title, ha="center", fontsize=10,
            fontweight="bold", color=PUB_COLORS["olive"])

ax.set_xlim(x_pos[0] - 1.0, x_pos[-1] + 1.0)
ax.set_ylim(max_height + 0.5, -2.2)
ax.axis("off")

fig.suptitle("Figure 2 \u2014 Sankey diagram of species flow across the 3 models and "
            "consensus (ribbon width = number of species making that exact transition)",
            y=1.01, fontsize=11.5)

plt.tight_layout()

_save_pub_figure(fig, out_stub)

```

```

def fig3_centroid_comparison(traj_array, species_order, day_order, band_names,
                             res_m1, res_m2, res_m3, growth_band,
                             out_stub="fig3_centroid_comparison"):
    _apply_pub_style()
    b_idx = band_names.index(growth_band)
    linestyles = ["-", "--", "-.", ":"]

    fig, axes = plt.subplots(1, 3, figsize=(13, 4.4), dpi=PUB_DPI, sharey=True)
    for ax, res, title in zip(
        axes, (res_m1, res_m2, res_m3),
        ("Model 1: Hard-DTW", "Model 2: Soft-DTW", "Model 3: Hierarchical/DTW")):
        labels = np.asarray(res["cluster_labels"])
        sp_order = res["species_order"]
        sp_idx_local = {s: i for i, s in enumerate(sp_order)}
        for c_i, c in enumerate(sorted(np.unique(labels))):
            member_species = [s for s in sp_order if labels[sp_idx_local[s]] == c]
            idxs = [species_order.index(s) for s in member_species]
            mean_traj = traj_array[idxs, :, b_idx].mean(axis=0)
            color, _hatch = _cluster_color(c_i)
            ls = linestyles[(c_i // len(PUB_CYCLE)) % len(linestyles)]
            ax.plot(day_order, mean_traj, color=color, linewidth=2.2, linestyle=ls,
                    label=f"C {c} (n={len(member_species)})")
        ax.set_title(title, fontsize=10.5)
        ax.set_xlabel("Cultivation day")
        ax.grid(True)
        ax.legend(frameon=False, fontsize=7, loc="best")
    axes[0].set_ylabel(f"{growth_band} (mean band value)")
    fig.suptitle(f"Figure 3 \u2014 Cluster centroid trajectories per model ({growth_band})", y=1.04)
    plt.tight_layout()

```

```
_save_pub_figure(fig, out_stub)
```

```
def fig4_pigment_growth_boxplots(ds, traj_array, species_order, day_order, band_names,  
                                consensus_df, cluster_col="Consensus_Cluster",  
                                out_stub="fig4_pigment_growth_boxplots"):
```

```
    _apply_pub_style()
```

```
    pig_res, _pig_summary, pig_merged = validate_against_pigments(ds, consensus_df, cluster_col)
```

```
    growth_res, _growth_summary, growth_merged, _growth_df = validate_against_growth_curves(  
        traj_array, species_order, day_order, band_names, consensus_df, cluster_col)
```

```
    fig, axes = plt.subplots(1, 3, figsize=(13, 4.6), dpi=PUB_DPI)
```

```
    clusters = sorted(pig_merged[cluster_col].unique())
```

```
def _box(ax, merged, col, title, p_val):
```

```
    data = [merged.loc[merged[cluster_col] == c, col].dropna().values for c in clusters]
```

```
    bp = ax.boxplot(data, patch_artist=True, widths=0.55)
```

```
    for i, box in enumerate(bp["boxes"]):
```

```
        color, hatch = _cluster_color(i)
```

```
        box.set(facecolor=color, edgecolor=PUB_COLORS["olive"], hatch=hatch, alpha=0.92)
```

```
    for element in ("whiskers", "caps"):
```

```
        for line in bp[element]:
```

```
            line.set(color=PUB_COLORS["olive"])
```

```
    for median in bp["medians"]:
```

```
        median.set(color=PUB_COLORS["white"], linewidth=1.8)
```

```
    for flier in bp["fliers"]:
```

```
        flier.set(markeredgecolor=PUB_COLORS["olive"], markerfacecolor=PUB_COLORS["cream"])
```

```
    ax.set_xticks(range(1, len(clusters) + 1))
```

```
    ax.set_xticklabels([f"C{c}" for c in clusters])
```

```
    p_str = f"p={p_val:.4f}" if p_val is not None else "p=NA"
```

```

ax.set_title(f'{title} ({p_str})', fontsize=10.5)

ax.grid(True, axis="y")

_box(axes[0], pig_merged, "NDVI", "NDVI", pig_res["NDVI"]["p_value"])
_box(axes[1], pig_merged, "GNDVI", "GNDVI", pig_res["GNDVI"]["p_value"])
_box(axes[2], growth_merged, "AUC", "AUC (growth accumulation)", growth_res["AUC"]["p_value"])

fig.suptitle("Figure 4 \u2014 Independent biological validation across consensus clusters"
            " (Kruskal\u2013Wallis)", y=1.04)

plt.tight_layout()

_save_pub_figure(fig, out_stub)

def fig5_channel_signal_audit(channel_diagnostic_report, out_stub="fig5_channel_signal_audit"):
    _apply_pub_style()
    if channel_diagnostic_report is None or channel_diagnostic_report.empty:
        logger.warning("No channel diagnostic report available (an explicit --use_channel was "
                        "'set, so 'auto' never ran); skipping Figure 5.")
        return

    df = channel_diagnostic_report
    bands = sorted(df["band"].unique())
    channels = sorted(df["channel"].unique())
    fig, ax = plt.subplots(figsize=(9.5, 5), dpi=PUB_DPI)

    x = np.arange(len(bands))
    width = 0.8 / max(len(channels), 1)
    for c_i, channel in enumerate(channels):
        color, hatch = _cluster_color(c_i)
        vals = [

```

```

df[(df["band"] == b) & (df["channel"] == channel)][["fraction_files_with_real_signal"]]
    .values[0] for b in bands
]
ax.bar(x + c_i * width, vals, width=width, color=color, edgecolor=PUB_COLORS["olive"],
       hatch=hatch, label=channel)

ax.set_xticks(x + width * (len(channels) - 1) / 2)
ax.set_xticklabels(bands, rotation=30, ha="right")
ax.set_ylabel("Fraction of sampled files with real signal")
ax.set_ylim(0, 1.08)
ax.legend(frameon=False, title="Channel")
ax.grid(True, axis="y")
fig.suptitle("Figure 5 \u2014 Channel signal-coverage audit across biochemical bands", y=1.04)
plt.tight_layout()
_save_pub_figure(fig, out_stub)

```

```

def fig6_taxonomy_vs_consensus(consensus_df, cluster_col="Consensus_Cluster",
                               taxonomy=SPECIES_TAXONOMY, out_stub="fig6_taxonomy_vs_consensus"):
    _apply_pub_style()
    tax_res = validate_against_taxonomy(consensus_df, cluster_col, taxonomy)
    crosstab = tax_res["crosstab"]
    if crosstab is None:
        logger.warning("Not enough distinct taxa/clusters for Figure 6; skipping.")
        return

    fig, ax = plt.subplots(figsize=(10, 5.5), dpi=PUB_DPI)
    clusters = crosstab.columns.tolist()
    taxa = crosstab.index.tolist()
    x = np.arange(len(clusters))

```

```

bottom = np.zeros(len(clusters))

for t_i, taxon in enumerate(taxa):
    color, hatch = _cluster_color(t_i)

    vals = crosstab.loc[taxon, clusters].values.astype(float)

    ax.bar(x, vals, bottom=bottom, color=color, edgecolor=PUB_COLORS["olive"],
           hatch=hatch, label=taxon, width=0.62)

    bottom += vals

ax.set_xticks(x)
ax.set_xticklabels([f'C{c}' for c in clusters])
ax.set_ylabel("Number of species")
ax.set_xlabel("Consensus cluster")

ari, nmi = tax_res["ari_vs_taxonomy"], tax_res["nmi_vs_taxonomy"]
ax.legend(frameon=False, fontsize=7.5, title="Taxon", loc="upper right")
ax.grid(True, axis="y")

fig.suptitle(f'Figure 6 \u2014 Taxon composition per consensus cluster '
            f'(ARI={ari:.3f}, NMI={nmi:.3f})", y=1.04)

plt.tight_layout()

_save_pub_figure(fig, out_stub)

```

```

def fig7_dtw_vs_euclidean(out_stub="fig7_dtw_vs_euclidean"):
    _apply_pub_style()

    x = np.linspace(0, 10, 80)

    curve_a = np.sin(x / 2 - 0.3) + 0.1 * x
    curve_b = np.sin((x - 1.6) / 2 - 0.3) + 0.1 * (x - 1.6) + 0.2

    fig, axes = plt.subplots(1, 2, figsize=(10.5, 4.6), dpi=PUB_DPI, sharey=True)

    ax = axes[0]

```

```

ax.plot(x, curve_a, color=PUB_COLORS["olive"], linewidth=2.2, label="Species A")
ax.plot(x, curve_b, color=PUB_COLORS["ochre"], linewidth=2.2, label="Species B")
for i in range(0, len(x), 4):
    ax.plot([x[i], x[i]], [curve_a[i], curve_b[i]], color=PUB_COLORS["sage"], lw=0.8, alpha=0.8)
ax.set_title("Rigid Euclidean matching\n(fixed $t_i \rightarrow t_i$)")
ax.set_xlabel("Cultivation day (schematic)")
ax.set_ylabel("Band signal (a.u., schematic)")
ax.legend(frameon=False)
ax.grid(True)

```

```

ax = axes[1]
ax.plot(x, curve_a, color=PUB_COLORS["olive"], linewidth=2.2)
ax.plot(x, curve_b, color=PUB_COLORS["ochre"], linewidth=2.2)
shift = 8
for i in range(0, len(x) - shift, 4):
    ax.plot([x[i], x[i + shift]], [curve_a[i], curve_b[i + shift]],
            color=PUB_COLORS["brown"], lw=0.8, alpha=0.8)
ax.set_title("Elastic DTW alignment\n(warped path)")
ax.set_xlabel("Cultivation day (schematic)")
ax.grid(True)

```

```

fig.suptitle("Figure 7 \u2014 Rigid Euclidean matching vs. elastic DTW alignment", y=1.04)
plt.tight_layout()
_save_pub_figure(fig, out_stub)

```

```

def generate_all_publication_figures(res_m1, res_m2, res_m3, consensus_df, ds,
                                    traj_array, species_order, day_order, band_names):
    growth_band = compute_growth_curve_features(
        traj_array, species_order, day_order, band_names, growth_band="auto"
    )

```

```

).attrs["growth_band_used"]

logger.info("Generating publication figures (Book Antiqua / palette-only / white "
           "background, PNG @ %d dpi + PDF)...", PUB_DPI)
fig1_algorithm_mechanics(res_m2["best_gamma"])
fig2_cluster_flow(species_order, res_m1, res_m2, res_m3, consensus_df)
fig3_centroid_comparison(traj_array, species_order, day_order, band_names,
                          res_m1, res_m2, res_m3, growth_band)
fig4_pigment_growth_boxplots(ds, traj_array, species_order, day_order, band_names, consensus_df)
fig5_channel_signal_audit(getattr(ds, "channel_diagnostic_report", None))
fig6_taxonomy_vs_consensus(consensus_df, "Consensus_Cluster")
fig7_dtw_vs_euclidean()
logger.info("All publication figures saved to '%s' (7 figures x .png + .pdf).", OUTPUT_DIR)

# =====
# 9b. NEW: individual per-model labeled trajectory figures + PCA latent plots
# =====

# This section answers "give me one clean figure per model, not one crowded
# combined plot" -- each of the 4 clusterings (Model 1 / 2 / 3 / Consensus)
# gets its own trajectory figure with an auto-generated descriptive cluster
# name (same spirit as your reference image's "Rapid Biomass Accumulators"
# etc., but derived from THIS run's actual growth-curve numbers), plus its
# own PCA "latent space" scatter of the DTW-scaled trajectories.
# =====

def _individual_species_growth_stats(traj_array, species_order, day_order, band_names, growth_band):
    b_idx = band_names.index(growth_band)
    days = np.asarray(day_order, dtype=float)
    mid = len(days) // 2

```

```

rows = []
for i, sp in enumerate(species_order):
    y = traj_array[i, :, b_idx]
    rows.append({
        "Species": sp,
        "AUC": float(_trapz(y, days)),
        "Max": float(y.max()),
        "Day_of_Max": float(days[int(np.argmax(y))]),
        "Late_Slope": float((y[-1] - y[mid]) / max(days[-1] - days[mid], 1e-6)),
    })
return pd.DataFrame(rows)

```

```

def _individual_auto_label(auc_pct, late_slope_pct):
    """Purely descriptive of the ranked numbers for THIS clustering -- not a
    mechanistic claim. Rename after inspecting pigment data if you like."""
    if auc_pct >= 0.75:
        return "Rapid Biomass Accumulators"
    if late_slope_pct >= 0.75 and auc_pct < 0.75:
        return "Late-Stage / Stress-Induced Accumulators"
    if auc_pct <= 0.25:
        return "Slow / Vegetative Basal Producers"
    return "Moderate Growth Group"

```

```

def label_clusters_individual(traj_array, species_order, day_order, band_names, cluster_labels,
                             growth_band="Chlorophyll_680"):
    """Returns (label_map, summary_df). label_map: {cluster_id: (name, [members])}."""
    growth_df = _individual_species_growth_stats(traj_array, species_order, day_order, band_names, growth_band)
    sp_to_cluster = dict(zip(species_order, cluster_labels))

```

```

growth_df["Cluster"] = growth_df["Species"].map(sp_to_cluster)

summary = growth_df.groupby("Cluster")[["AUC", "Max", "Day_of_Max", "Late_Slope"]].mean()

```

```

auc_rank = summary["AUC"].rank(pct=True)

late_rank = summary["Late_Slope"].rank(pct=True)

```

```

label_map = {}

for c in summary.index:

    name = _individual_auto_label(auc_rank[c], late_rank[c])

    members = [s for s in species_order if sp_to_cluster[s] == c]

    label_map[c] = (name, members)

return label_map, summary

```

```

def _align_labels_individual(global_species_order, res_species_order, res_labels):

```

```

    """Re-orders a model's cluster_labels to match the global species_order,
    matching by species name (guards against any ordering mismatch)."""

    sp_to_label = dict(zip(res_species_order, res_labels))

    return np.array([sp_to_label[s] for s in global_species_order])

```

```

def plot_model_trajectories_individual(traj_array, species_order, day_order, band_names,
                                     cluster_labels, model_name, out_stub,
                                     plot_band="Chlorophyll_680"):

```

```

    """One clean trajectory figure for ONE model, with auto-labeled clusters
    listing their descriptive name and member species directly in the legend."""

    apply_plot_style()

    b_idx = band_names.index(plot_band)

    cluster_labels = np.asarray(cluster_labels)

    unique_clusters = sorted(np.unique(cluster_labels))

```

```
colors = [PALETTE["olive"], PALETTE["ochre"], PALETTE["sage"], PALETTE["brown"]] +  
EXTRA_COLORS
```

```
label_map, summary = label_clusters_individual(  
    traj_array, species_order, day_order, band_names, cluster_labels, growth_band=plot_band)
```

```
fig, ax = plt.subplots(figsize=(11, 7), dpi=300)
```

```
for c_i, c in enumerate(unique_clusters):
```

```
    idx = np.where(cluster_labels == c)[0]
```

```
    member_traj = traj_array[idx, :, b_idx]
```

```
    mean_t, std_t = member_traj.mean(axis=0), member_traj.std(axis=0)
```

```
    color = colors[c_i % len(colors)]
```

```
    desc, members = label_map[c]
```

```
    legend_txt = f'Cluster {c}: {desc}\n( {', '.join(members)} )'
```

```
    ax.plot(day_order, mean_t, color=color, linewidth=2.6, marker="o",
```

```
            markersize=3.5, label=legend_txt)
```

```
    ax.fill_between(day_order, mean_t - std_t, mean_t + std_t, color=color, alpha=0.15)
```

```
ax.set_title(f'{model_name}\nMean Kinetic Trajectories by Cluster ( {plot_band} )')
```

```
ax.set_xlabel("Cultivation Time (Days)")
```

```
ax.set_ylabel(f'Mean Band Response ( {plot_band} )')
```

```
ax.grid(True, alpha=0.6)
```

```
ax.legend(loc="upper left", frameon=True, fontsize=8.5)
```

```
plt.tight_layout()
```

```
fig.savefig(f'{OUTPUT_DIR}/{out_stub}_trajectories_labeled.png', dpi=300, bbox_inches="tight")
```

```
plt.close(fig)
```

```
return summary
```

```
def plot_model_pca_projection_individual(traj_array, species_order, cluster_labels, model_name, out_stub):
```

```
    """PCA scatter of the DTW-scaled trajectories for ONE model, i.e. a 2-D
```

```

'latent space' view of how that model's clusters separate species.''''

apply_plot_style()

scaler = TimeSeriesScalerMeanVariance()

traj_scaled = scaler.fit_transform(traj_array)

flat = traj_scaled.reshape(traj_scaled.shape[0], -1)


pca = PCA(n_components=2, random_state=SEED)

coords = pca.fit_transform(flat)

var_exp = pca.explained_variance_ratio_ * 100


cluster_labels = np.asarray(cluster_labels)

unique_clusters = sorted(np.unique(cluster_labels))

colors = [PALETTE["olive"], PALETTE["ochre"], PALETTE["sage"], PALETTE["brown"]] +
EXTRA_COLORS


fig, ax = plt.subplots(figsize=(8.5, 7), dpi=300)

for c_i, c in enumerate(unique_clusters):

    idx = np.where(cluster_labels == c)[0]

    color = colors[c_i % len(colors)]

    ax.scatter(coords[idx, 0], coords[idx, 1], s=120, color=color,

               edgecolor=PALETTE["olive"], linewidth=1.0, label=f"Cluster {c}", zorder=3)

    for i in idx:

        ax.annotate(species_order[i], (coords[i, 0], coords[i, 1]), fontsize=8,

                    xytext=(4, 4), textcoords="offset points", color=PALETTE["olive"])


ax.set_xlabel(f"PC1 ( {var_exp[0]:.1f}% variance)")
ax.set_ylabel(f"PC2 ( {var_exp[1]:.1f}% variance)")
ax.set_title(f"{model_name}\nPCA Latent-Space Projection of Species Trajectories")
ax.grid(True, alpha=0.5)
ax.legend(loc="best", frameon=True, fontsize=9)

plt.tight_layout()

```

```
fig.savefig(f'{OUTPUT_DIR}/{out_stub}_pca_latent.png', dpi=300, bbox_inches="tight")

plt.close(fig)
```

```
def generate_all_individual_reports(res_m1, res_m2, res_m3, consensus_df,
                                   traj_array, species_order, day_order, band_names,
                                   plot_band="Chlorophyll_680"):
    """Driver: produces 2 figures (trajectory + PCA latent) for EACH of the
    4 clusterings (Model 1, Model 2, Model 3, Consensus) -- 8 files total in
    OUTPUT_DIR, replacing the one crowded combined chart."""
    consensus_map = dict(zip(consensus_df["Species"], consensus_df["Consensus_Cluster"]))
    consensus_labels_global = np.array([consensus_map[s] for s in species_order])

    models = [
        (_align_labels_individual(species_order, res_m1["species_order"], res_m1["cluster_labels"]),
         "Model 1: Hard-DTW K-Means", "model1_hard_dtw"),
        (_align_labels_individual(species_order, res_m2["species_order"], res_m2["cluster_labels"]),
         "Model 2: Soft-DTW K-Means", "model2_soft_dtw"),
        (_align_labels_individual(species_order, res_m3["species_order"], res_m3["cluster_labels"]),
         "Model 3: Hierarchical / DTW", "model3_hierarchical"),
        (consensus_labels_global, "Consensus (3-model agreement)", "model4_consensus"),
    ]

    summaries = {}

    for labels, name, stub in models:
        print(f"\nGenerating individual figures for: {name}")
        summary = plot_model_trajectories_individual(
            traj_array, species_order, day_order, band_names, labels, name, stub,
            plot_band=plot_band)
        plot_model_pca_projection_individual(traj_array, species_order, labels, name, stub)
```

```

        summaries[name] = summary

        print(summary.round(3).to_string())

    return summaries

# =====
# 10. Entry point
# =====

def build_arg_parser():

    parser = argparse.ArgumentParser(description="3-model DTW consensus clustering on your own dataset")

    parser.add_argument("--data_dir", type=str, default=DATA_DIR,
                        help="Path to your dataset folder (see header comment for expected layout).")

    parser.add_argument("--use_channel", type=str, default="auto",
                        help="Column suffix to read per ROI, e.g. 'Raw' or 'Reflectance'. Default "
                             "'auto' samples your files and picks whichever channel has the most "
                             "'real (non-zero-filled) signal in the biochemical bands this pipeline "
                             "clusters on -- run with --diagnose_channels first to inspect this "
                             "choice before a full run.")

    parser.add_argument("--diagnose_channels", action="store_true",
                        help="Run the channel/band signal-coverage diagnostic and exit, without "
                             "running the (slow) DTW clustering. Use this first on a new dataset.")

    parser.add_argument("--allow_synthetic_fallback", action="store_true",
                        help="If set, falls back to fabricated data when no real files are found "
                             "(useful only for a smoke test -- do NOT use this for real results).")

    parser.add_argument("--k_min", type=int, default=2)

    parser.add_argument("--k_max", type=int, default=8)

    parser.add_argument("--n_stability_seeds", type=int, default=5)

    parser.add_argument("--consensus_k", type=int, default=None,

```

```

        help="Fix the number of CONSENSUS clusters manually. If omitted "
        "(default), k is chosen objectively by sweeping k_min..k_max and "
        "maximizing DTW-silhouette + mean within-cluster co-association, "
        "subject to the same dominant-cluster-fraction balance guard used "
        "throughout the rest of the pipeline. See select_consensus_k().")

    return parser

def main(argv=None):
    parser = build_arg_parser()
    args, _ = parser.parse_known_args(args=argv)
    set_seed(SEED)

    logger.info("data_dir=%s use_channel=%s allow_synthetic_fallback=%s",
                args.data_dir, args.use_channel, args.allow_synthetic_fallback)

    if args.diagnose_channels:
        diagnose_channel_band_coverage(args.data_dir)
        logger.info("Channel diagnostic complete (--diagnose_channels was set); exiting without "
                    "running the clustering pipeline.")
        return None

    ds = MicroalgaeROIDataset(args.data_dir, use_synthetic_fallback=args.allow_synthetic_fallback,
                              use_channel=args.use_channel)
    logger.info("Loaded: all_features=%s, species=%s, data_source=%s, channel_used=%s",
                ds.all_features.shape, ds.species_classes, ds.data_source, ds.use_channel)

    traj_array, species_order, day_order, band_names = build_species_day_trajectories(ds)
    logger.info("Trajectory tensor shape (n_species, n_days, n_features): %s", traj_array.shape)

```

```

# FIX #2 (circularity): growth-curve "validation" used to be computed from
# a band auto-selected out of the SAME six bands fed into clustering
# (e.g. it kept picking Water_NIR_970 -- one of the exact features the
# models were fit on). Testing whether clusters explain variation in a
# feature the clusters were built from is not independent validation.
#
# Fix: hold ONE band out of the clustering feature set entirely and use
# it ONLY downstream for growth-curve validation. The model never sees
# it, so a significant Kruskal-Wallis result on it is now genuine
# out-of-feature-set evidence, not a tautology.
#
# Water_NIR_970 is kept as the held-out band because it was the band
# earlier runs kept auto-selecting anyway (highest across-species
# variance) -- so this is the same signal, now used honestly.
if GROWTH_VALIDATION_BAND not in band_names:
    raise ValueError(
        f'GROWTH_VALIDATION_BAND='{GROWTH_VALIDATION_BAND}' not found in
band_names={band_names}; "
        "update the constant to match DTW_BAND_INDICES.")
clustering_band_names = [b for b in band_names if b != GROWTH_VALIDATION_BAND]
clustering_band_idx = [band_names.index(b) for b in clustering_band_names]
traj_array_cluster = traj_array[:, :, clustering_band_idx]
logger.info(
    "Excluding '%s' from the clustering feature set (%d bands remain: %s); it is reserved "
    "for out-of-sample growth-curve validation only.",
    GROWTH_VALIDATION_BAND, len(clustering_band_names), clustering_band_names,
)

shared_scaler = TimeSeriesScalerMeanVariance()
shared_traj_scaled = shared_scaler.fit_transform(traj_array_cluster)
logger.info("Computing shared pairwise DTW distance matrix (reused by all 3 models for scoring)...")

```

```

shared_dtw_dist = compute_dtw_distance_matrix(shared_traj_scaled)

print("\n" + "=" * 80 + "\nRUNNING MODEL 1: Hard-DTW K-Means\n" + "=" * 80)

res_m1 = run_dtw_temporal_clustering(traj_array_cluster, species_order, day_order, clustering_band_names, ds,
                                    k_min=args.k_min, k_max=args.k_max,
                                    n_stability_seeds=args.n_stability_seeds,
                                    shared_dtw_dist=shared_dtw_dist)

print("\n" + "=" * 80 + "\nRUNNING MODEL 2: Soft-DTW K-Means\n" + "=" * 80)

res_m2 = run_softdtw_kmeans_clustering(traj_array_cluster, species_order, day_order, clustering_band_names,
ds,
                                    k_min=args.k_min, k_max=args.k_max,
                                    n_stability_seeds=args.n_stability_seeds,
                                    shared_dtw_dist=shared_dtw_dist)

print("\n" + "=" * 80 + "\nRUNNING MODEL 3: Hierarchical DTW Clustering\n" + "=" * 80)

res_m3 = run_hierarchical_dtw_clustering(traj_array_cluster, species_order, day_order, clustering_band_names,
ds,
                                    k_min=args.k_min, k_max=args.k_max,
                                    n_stability_seeds=args.n_stability_seeds,
                                    shared_dtw_dist=shared_dtw_dist)

internal_rows = []
for res in (res_m1, res_m2, res_m3):
    r = res["report"]
    internal_rows.append({
        "method": r["method_name"], "k_used": r.get("k_used"),
        "n_clusters": r["n_clusters"],
        "silhouette_dtw_precomputed": r.get("silhouette_dtw_precomputed"),
        "silhouette_flat_euclidean": r.get("silhouette_flat_euclidean"),
        "davies_bouldin_flat_euclidean": r.get("davies_bouldin_flat_euclidean"),
    })

```

```

        "dominant_cluster_fraction": r["cluster_balance"]["dominant_cluster_fraction"],
        "stability_mean_pairwise_ARI": r.get("stability_mean_pairwise_ARI"),
    })

internal_comparison_df = pd.DataFrame(internal_rows)

print("\n" + "=" * 80 + "\nINTERNAL-VALIDITY COMPARISON ACROSS MODELS (shared DTW distance
matrix)\n" + "=" * 80)

print(internal_comparison_df.to_string(index=False))

apply_plot_style()

internal_comparison_df.to_csv(f"{OUTPUT_DIR}/internal_validity_comparison.csv", index=False)


print("\n" + "=" * 80 + "\nCOMPUTING ENSEMBLE CONSENSUS CLUSTERING\n" + "=" * 80)

coassoc = build_coassociation_matrix(
    species_order,
    (res_m1["species_order"], res_m1["cluster_labels"]),
    (res_m2["species_order"], res_m2["cluster_labels"]),
    (res_m3["species_order"], res_m3["cluster_labels"]),
)

plot_coassociation_heatmap(coassoc, species_order, out_stub="consensus")


print("\n" + "=" * 80 + "\nOBJECTIVE CONSENSUS-K SELECTION\n" + "=" * 80)

if args.consensus_k is not None and args.consensus_k > 0:
    logger.info("Using user-specified consensus_k=%d (objective auto-selection skipped).",
               args.consensus_k)

    best_consensus_k = args.consensus_k

    _dist = 1.0 - coassoc

    np.fill_diagonal(_dist, 0.0)

    Z_consensus_precomputed = scipy_linkage(squareform(_dist, checks=False), method="average")
else:
    best_consensus_k, consensus_k_sweep_df, Z_consensus_precomputed = select_consensus_k(
        coassoc, shared_dtw_dist, k_min=args.k_min, k_max=args.k_max)

    consensus_k_sweep_df.to_csv(f"{OUTPUT_DIR}/consensus_k_selection_sweep.csv", index=False)

```

```

consensus_df, Z_consensus, consensus_labels = consensus_cluster(
    coassoc, species_order, n_clusters=best_consensus_k, out_stub="consensus",
    precomputed_Z=Z_consensus_precomputed)
print_consensus_interpretation(coassoc, species_order, consensus_df)

# FIX #5 (transparency): consensus-k selection has, in practice, ended up
# choosing a k/partition that is IDENTICAL to one individual model's own
# partition (checked here via ARI=1.0 against each model). When that
# happens, "consensus" isn't adding independent multi-model agreement --
# it's just relabeling one model's result. This isn't fixed automatically
# (there may be legitimate cases where 3 models genuinely agree exactly),
# but it must be visible instead of silently reported as "consensus".
for res, label in ((res_m1, "Model 1: Hard-DTW"), (res_m2, "Model 2: Soft-DTW"),
                  (res_m3, "Model 3: Hierarchical-DTW")):
    model_labels = res["result_df"].set_index("Species").iloc[:, 0].reindex(species_order)
    common = model_labels.notna()
    if common.sum() >= 2:
        ari_vs_model = adjusted_rand_score(
            consensus_labels[common.values], model_labels[common].values)
        if ari_vs_model >= 0.999:
            logger.warning(
                "Consensus partition is IDENTICAL (ARI=%0.3f) to %s's own partition. "
                "The 'consensus' result below is not independent multi-model agreement in "
                "this run -- it has converged onto a single model. Report this explicitly "
                "rather than presenting Consensus as if it synthesizes all three.",
                ari_vs_model, label)

    print("\n" + "=" * 80 + "\nINDEPENDENT BIOLOGICAL VALIDATION (taxonomy / pigments / growth
curves)\n" + "=" * 80)

    bio_results = {}

```

```

for res, label in ((res_m1, "Model 1: Hard-DTW"), (res_m2, "Model 2: Soft-DTW"),
                  (res_m3, "Model 3: Hierarchical-DTW")):
    bio_results[label] = biological_validation_suite(
        res["result_df"], res["cluster_col"], ds, traj_array, species_order, day_order,
        band_names, label=label)

bio_results["Consensus"] = biological_validation_suite(
    consensus_df, "Consensus_Cluster", ds, traj_array, species_order, day_order,
    band_names, label="Consensus")

bio_comparison_df = pd.DataFrame([bio_results[k]["flat_row"] for k in bio_results])

print("\n" + "=" * 80 + "\nBIOLOGICAL-VALIDATION COMPARISON ACROSS MODELS +
CONSENSUS\n" + "=" * 80)

print(bio_comparison_df.to_string(index=False))

print(
    "\nReading guide: 'ari_vs_taxonomy' / 'nmi_vs_taxonomy' close to 0 mean the clusters do "
    "NOT track coarse phylum/class groupings (this can be a positive finding -- it would mean "
    "the spectral-kinetic signature captures something beyond taxonomy, e.g. shared pigment "
    "strategy or growth strategy across distantly related taxa -- but it should be reported, "
    "not hidden). 'pigment_*_p' and 'growth_*_p' below 0.05 indicate that cluster membership "
    "significantly explains variation in that pigment index / growth descriptor, which is the "
    "strongest independent evidence for biologically meaningful spectral-kinetic phenotyping. "
    "Report the consensus row as your headline validation result; use the per-model rows to "
    "show the finding is not an artifact of one particular clustering algorithm."
)

bio_comparison_df.to_csv(f"{OUTPUT_DIR}/biological_validation_comparison.csv", index=False)

print("\n" + "=" * 80 + "\nGENERATING MANUSCRIPT PUBLICATION FIGURES\n" + "=" * 80)

generate_all_publication_figures(res_m1, res_m2, res_m3, consensus_df, ds,
                                traj_array, species_order, day_order, band_names)

```

```

print("\n" + "=" * 80 + "\nGENERATING INDIVIDUAL PER-MODEL LABELED FIGURES (trajectories +
PCA latent)\n" + "=" * 80)

individual_summaries = generate_all_individual_reports(
    res_m1, res_m2, res_m3, consensus_df, traj_array, species_order, day_order, band_names,
    plot_band="Chlorophyll_680")

logger.info("Pipeline execution complete! Figures and CSV summaries saved in '%s' directory.", OUTPUT_DIR)

return {"res_m1": res_m1, "res_m2": res_m2, "res_m3": res_m3,
        "coassoc": coassoc, "consensus_df": consensus_df,
        "internal_comparison_df": internal_comparison_df,
        "bio_results": bio_results, "bio_comparison_df": bio_comparison_df,
        "individual_summaries": individual_summaries}

# =====

# SCRIPT DRIVER -- training + validation only (no figures)

# =====

def run_training_and_validation():
    os.makedirs(OUTPUT_DIR, exist_ok=True)
    print("OUTPUT_DIR =", OUTPUT_DIR)

    # -----
    # 1. Configuration -- edit these directly (equivalents of the
    # original argparse flags).
    # -----

    data_dir          = DATA_DIR # <-- point this at your real dataset folder
    use_channel        = "auto"    # 'auto' | 'Raw' | 'Reflectance' | ...
    allow_synthetic_fallback = False # True only for a smoke test, never for real results
    k_min, k_max       = 2, 8
    n_stability_seeds   = 5

```

```

consensus_k          = None    # None = auto-select objectively; or an int to force it

set_seed(SEED)

logger.info("data_dir=%s use_channel=%s allow_synthetic_fallback=%s",
           data_dir, use_channel, allow_synthetic_fallback)

# -----
# 2. Load data & build trajectories
# -----

ds = MicroalgaeROIDataset(data_dir, use_synthetic_fallback=allow_synthetic_fallback,
                          use_channel=use_channel)

logger.info("Loaded: all_features=%s, species=%s, data_source=%s, channel_used=%s",
           ds.all_features.shape, ds.species_classes, ds.data_source, ds.use_channel)

traj_array, species_order, day_order, band_names = build_species_day_trajectories(ds)

logger.info("Trajectory tensor shape (n_species, n_days, n_features): %s", traj_array.shape)

if GROWTH_VALIDATION_BAND not in band_names:
    raise ValueError(
        f"GROWTH_VALIDATION_BAND='{GROWTH_VALIDATION_BAND}' not found in
band_names={band_names}")

clustering_band_names = [b for b in band_names if b != GROWTH_VALIDATION_BAND]
clustering_band_idx   = [band_names.index(b) for b in clustering_band_names]
traj_array_cluster    = traj_array[:, :, clustering_band_idx]

logger.info("Excluding '%s' from clustering features (%d bands remain: %s)",
           GROWTH_VALIDATION_BAND, len(clustering_band_names), clustering_band_names)

shared_scaler        = TimeSeriesScalerMeanVariance()
shared_traj_scaled   = shared_scaler.fit_transform(traj_array_cluster)

logger.info("Computing shared pairwise DTW distance matrix (reused by all 3 models)...")

```

```

shared_dtw_dist = compute_dtw_distance_matrix(shared_traj_scaled)

# -----

# 3-5. Model 1 / 2 / 3

# -----

print("\n" + "=" * 80 + "\nRUNNING MODEL 1: Hard-DTW K-Means\n" + "=" * 80)

res_m1 = run_dtw_temporal_clustering(
    traj_array_cluster, species_order, day_order, clustering_band_names, ds,
    k_min=k_min, k_max=k_max, n_stability_seeds=n_stability_seeds,
    shared_dtw_dist=shared_dtw_dist)

print("\n" + "=" * 80 + "\nRUNNING MODEL 2: Soft-DTW K-Means\n" + "=" * 80)

res_m2 = run_softdtw_kmeans_clustering(
    traj_array_cluster, species_order, day_order, clustering_band_names, ds,
    k_min=k_min, k_max=k_max, n_stability_seeds=n_stability_seeds,
    shared_dtw_dist=shared_dtw_dist)

print("\n" + "=" * 80 + "\nRUNNING MODEL 3: Hierarchical DTW Clustering\n" + "=" * 80)

res_m3 = run_hierarchical_dtw_clustering(
    traj_array_cluster, species_order, day_order, clustering_band_names, ds,
    k_min=k_min, k_max=k_max, n_stability_seeds=n_stability_seeds,
    shared_dtw_dist=shared_dtw_dist)

# -----

# 6. Internal-validity comparison across models

# -----

internal_rows = []

for res in (res_m1, res_m2, res_m3):
    r = res["report"]
    internal_rows.append({

```

```

        "method": r["method_name"], "k_used": r.get("k_used"),
        "n_clusters": r["n_clusters"],
        "silhouette_dtw_precomputed": r.get("silhouette_dtw_precomputed"),
        "silhouette_flat_euclidean": r.get("silhouette_flat_euclidean"),
        "davies_bouldin_flat_euclidean": r.get("davies_bouldin_flat_euclidean"),
        "dominant_cluster_fraction": r["cluster_balance"]["dominant_cluster_fraction"],
        "stability_mean_pairwise_ARI": r.get("stability_mean_pairwise_ARI"),
    })

internal_comparison_df = pd.DataFrame(internal_rows)

print("\n" + "=" * 80 + "\nINTERNAL-VALIDITY COMPARISON ACROSS MODELS\n" + "=" * 80)

print(internal_comparison_df.to_string(index=False))

apply_plot_style()

internal_comparison_df.to_csv(f"{OUTPUT_DIR}/internal_validity_comparison.csv", index=False)

# -----
# 7. Ensemble consensus clustering
# -----

print("\n" + "=" * 80 + "\nCOMPUTING ENSEMBLE CONSENSUS CLUSTERING\n" + "=" * 80)

coassoc = build_coassociation_matrix(
    species_order,
    (res_m1["species_order"], res_m1["cluster_labels"]),
    (res_m2["species_order"], res_m2["cluster_labels"]),
    (res_m3["species_order"], res_m3["cluster_labels"]),
)

plot_coassociation_heatmap(coassoc, species_order, out_stub="consensus")

print("\n" + "=" * 80 + "\nOBJECTIVE CONSENSUS-K SELECTION\n" + "=" * 80)

if consensus_k is not None and consensus_k > 0:
    logger.info("Using user-specified consensus_k=%d (auto-selection skipped).", consensus_k)
    best_consensus_k = consensus_k

```

```

_dist = 1.0 - coassoc
np.fill_diagonal(_dist, 0.0)

Z_consensus_precomputed = scipy_linkage(squareform(_dist, checks=False), method="average")
else:

    best_consensus_k, consensus_k_sweep_df, Z_consensus_precomputed = select_consensus_k(
        coassoc, shared_dtw_dist, k_min=k_min, k_max=k_max)

    consensus_k_sweep_df.to_csv(f'{OUTPUT_DIR}/consensus_k_selection_sweep.csv', index=False)

consensus_df, Z_consensus, consensus_labels = consensus_cluster(
    coassoc, species_order, n_clusters=best_consensus_k, out_stub="consensus",
    precomputed_Z=Z_consensus_precomputed)
print_consensus_interpretation(coassoc, species_order, consensus_df)

# -----
# 7a. Transparency check -- did consensus collapse onto one model?
# -----

for res, label in ((res_m1, "Model 1: Hard-DTW"), (res_m2, "Model 2: Soft-DTW"),
                  (res_m3, "Model 3: Hierarchical-DTW")):

    model_labels = res["result_df"].set_index("Species").iloc[:, 0].reindex(species_order)
    common = model_labels.notna()

    if common.sum() >= 2:

        ari_vs_model = adjusted_rand_score(
            consensus_labels[common.values], model_labels[common].values)

        if ari_vs_model >= 0.999:

            logger.warning(
                "Consensus partition is IDENTICAL (ARI=%0.3f) to %s's own partition.",
                ari_vs_model, label)

        else:

            print(f'{label}: ARI vs consensus = {ari_vs_model:.3f}')

```

```

# -----

# 8. Independent biological validation

# -----

print("\n" + "=" * 80 + "\nINDEPENDENT BIOLOGICAL VALIDATION\n" + "=" * 80)

bio_results = {}

for res, label in ((res_m1, "Model 1: Hard-DTW"), (res_m2, "Model 2: Soft-DTW"),
                  (res_m3, "Model 3: Hierarchical-DTW")):

    bio_results[label] = biological_validation_suite(
        res["result_df"], res["cluster_col"], ds, traj_array, species_order, day_order,
        band_names, label=label)

bio_results["Consensus"] = biological_validation_suite(
    consensus_df, "Consensus_Cluster", ds, traj_array, species_order, day_order,
    band_names, label="Consensus")


bio_comparison_df = pd.DataFrame([bio_results[k]["flat_row"] for k in bio_results])

print("\n" + "=" * 80 + "\nBIOLOGICAL-VALIDATION COMPARISON ACROSS MODELS +
CONSENSUS\n" + "=" * 80)

print(bio_comparison_df.to_string(index=False))

bio_comparison_df.to_csv(f"{OUTPUT_DIR}/biological_validation_comparison.csv", index=False)


# -----

# 9. Save everything the figures script needs

# -----

results_bundle = {
    "ds": ds,
    "traj_array": traj_array,
    "band_names": band_names,
    "day_order": day_order,
    "species_order": species_order,
    "res_m1": res_m1, "res_m2": res_m2, "res_m3": res_m3,
    "consensus_df": consensus_df,

```

```
"coassoc": coassoc,  
"internal_comparison_df": internal_comparison_df,  
"bio_results": bio_results,  
"bio_comparison_df": bio_comparison_df,  
}
```

```
pkl_path = os.path.join(OUTPUT_DIR, "model_results.pkl")  
with open(pkl_path, "wb") as f:  
    pickle.dump(results_bundle, f)
```

```
logger.info("Training + validation complete. Saved -> %s", pkl_path)  
print("Saved:", pkl_path)  
return results_bundle
```

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

Step 2 — Figure setup (imports, style, shared helpers)

```
import os  
import warnings  
import textwrap  
  
import numpy as np  
import pandas as pd  
import matplotlib  
import matplotlib.pyplot as plt  
import matplotlib.patches as mpatches  
from matplotlib import font_manager  
  
from scipy.cluster.hierarchy import linkage as scipy_linkage, fcluster
```

```
from scipy.spatial.distance import squareform
```

```
from scipy.stats import kruskal, pearsonr
```

```
from sklearn.decomposition import PCA
```

```
from sklearn.cluster import KMeans
```

```
from sklearn.metrics import silhouette_samples, adjusted_rand_score
```

```
from tslearn.metrics import cdist_dtw, dtw_path
```

```
from tslearn.clustering import TimeSeriesKMeans
```

```
from tslearn.preprocessing import TimeSeriesScalerMeanVariance
```

```
warnings.filterwarnings("ignore", message="findfont.*")
```

```
_trapz = getattr(np, "trapezoid", None) or np.trapz
```

```
SEED = 42
```

```
OUTPUT_DIR = "outputs"
```

```
# =====
```

```
# 1. House style -- Times New Roman, palette-only colors, high-res dual export
```

```
# =====
```

```
PUB_COLORS = {
```

```
    "olive": "#494623",
```

```
    "ochre": "#c3892b",
```

```
    "sage": "#8e883d",
```

```
    "brown": "#8f7c2f",
```

```
    "cream": "#e8e7d8",
```

```
    "white": "#ffffff",
```

```
}
```

```
# Cycle through the 4 "ink" colors for successive clusters/series; once a
```

```

# color repeats, a hatch pattern is layered on top to keep it distinguishable

# -- no color outside the 6 approved swatches is ever introduced.

PUB_CYCLE = [PUB_COLORS["olive"], PUB_COLORS["ochre"], PUB_COLORS["sage"],
PUB_COLORS["brown"]]

PUB_HATCHES = [ "", "/", "xx", "..", "\\\\", "++", "oo", "***"]

PUB_DPI = 600


_TIMES_CANDIDATES = ["Times New Roman", "Times", "Nimbus Roman", "Liberation Serif",
                    "Tinos", "DejaVu Serif"]


def _resolve_times_font():
    """Pick the first Times-like font actually installed, so matplotlib
    doesn't silently substitute something else without us knowing."""
    available = {f.name for f in font_manager.fontManager.ttflist}
    for name in _TIMES_CANDIDATES:
        if name in available:
            return name
    return "DejaVu Serif"


_TIMES_FONT = _resolve_times_font()


def _apply_pub_style():
    plt.rcParams.update({
        "font.family": "serif",
        "font.serif": [_TIMES_FONT] + _TIMES_CANDIDATES,
        "mathtext.fontset": "stix",
        "figure.facecolor": PUB_COLORS["white"],
        "axes.facecolor": PUB_COLORS["white"],

```

```

"savefig.facecolor": PUB_COLORS["white"],
"axes.edgecolor": PUB_COLORS["olive"],
"axes.labelcolor": PUB_COLORS["olive"],
"text.color": PUB_COLORS["olive"],
"xtick.color": PUB_COLORS["olive"],
"ytick.color": PUB_COLORS["olive"],
"axes.linewidth": 1.1,
"grid.color": PUB_COLORS["cream"],
"grid.linestyle": "--",
"grid.alpha": 0.9,
"axes.titlesize": 11.5,
"axes.labelsize": 10,
"legend.fontsize": 8.5,
"xtick.labelsize": 8.5,
"ytick.labelsize": 8.5,
"figure.autolayout": False,
})
os.makedirs(OUTPUT_DIR, exist_ok=True)

```

```

def _cluster_color(i):
    i = int(i)
    color = PUB_CYCLE[i % len(PUB_CYCLE)]
    hatch = PUB_HATCHES[(i // len(PUB_CYCLE)) % len(PUB_HATCHES)]
    return color, hatch

```

```

def _wrap(title, width=78):
    """Prevents long titles from merging into the plot area / other text."""
    return "\n".join(textwrap.wrap(title, width=width))

```

```

def _save_pub_figure(fig, stub, out_dir=OUTPUT_DIR, dpi=PUB_DPI):
    os.makedirs(out_dir, exist_ok=True)
    for ax in fig.get_axes():
        for spine in ax.spines.values():
            spine.set_color(PUB_COLORS["olive"])
    png_path = f"{out_dir}/{stub}.png"
    pdf_path = f"{out_dir}/{stub}.pdf"
    fig.savefig(png_path, dpi=dpi, bbox_inches="tight", facecolor=PUB_COLORS["white"])
    fig.savefig(pdf_path, bbox_inches="tight", facecolor=PUB_COLORS["white"])
    plt.close(fig)
    print(f" saved: {png_path} (+ .pdf)")

```

```

# =====

```

```

# 2. Small shared helpers (self-contained -- no import of the training file)

```

```

# =====

```

```

def _diagnose_band_growth_variance(traj_array, band_names):
    rows = []
    for b_idx, band in enumerate(band_names):
        per_species_max = traj_array[:, :, b_idx].max(axis=1)
        rows.append({"band": band, "std_of_max": float(np.std(per_species_max))})
    return pd.DataFrame(rows).sort_values("std_of_max", ascending=False).reset_index(drop=True)

```

```

def compute_growth_curve_features(traj_array, species_order, day_order, band_names,
                                  growth_band="auto"):
    if growth_band == "auto":

```

```

diag_df = _diagnose_band_growth_variance(traj_array, band_names)

growth_band = diag_df.iloc[0]["band"]

b_idx = band_names.index(growth_band)

days = np.asarray(day_order, dtype=float)

mid = len(days) // 2

rows = []

for i, sp in enumerate(species_order):

    y = traj_array[i, :, b_idx]

    auc = float(_trapz(y, days))

    max_val = float(y.max())

    day_of_max = float(days[int(np.argmax(y))])

    early_slope = float((y[mid] - y[0]) / max(days[mid] - days[0], 1e-6)) if mid > 0 else np.nan

    late_slope = float((y[-1] - y[mid]) / max(days[-1] - days[mid], 1e-6)) if mid < len(days) - 1 else np.nan

    rows.append({"Species": sp, "AUC": auc, "Max": max_val, "Day_of_Max": day_of_max,

                "Early_Slope": early_slope, "Late_Slope": late_slope})

growth_df = pd.DataFrame(rows)

growth_df.attrs["growth_band_used"] = growth_band

return growth_df

```

```

def _species_to_consensus(consensus_df, cluster_col="Consensus_Cluster"):

    return dict(zip(consensus_df["Species"], consensus_df[cluster_col]))

```

Step 3 — Load model results

```
import pickle
```

```
with open("outputs/model_results.pkl", "rb") as f:
```

```
    R = pickle.load(f)
```

```
    traj_array, band_names = R["traj_array"], R["band_names"]
```

```
    day_order, species_order = R["day_order"], R["species_order"]

```

```
res_m1, res_m2, res_m3 = R["res_m1"], R["res_m2"], R["res_m3"]
consensus_df, ds = R["consensus_df"], R["ds"]
```

Fig. 2 — Algorithm mechanics (Hard-DTW / Soft-DTW / Hierarchical-DTW)

```
def fig1_algorithm_mechanics(gamma_used, out_stub="fig1_algorithm_mechanics", out_dir=OUTPUT_DIR):
    _apply_pub_style()

    fig, axes = plt.subplots(1, 3, figsize=(13.5, 4.6), dpi=PUB_DPI, constrained_layout=True)

    x = np.linspace(0, 10, 60)
    curve_a = np.sin(x / 2) + 0.15 * x
    curve_b = np.sin((x - 1.4) / 2) + 0.15 * (x - 1.4) + 0.15

    ax = axes[0]
    ax.plot(x, curve_a, color=PUB_COLORS["olive"], linewidth=2.2, label="Species A")
    ax.plot(x, curve_b, color=PUB_COLORS["ochre"], linewidth=2.2, label="Species B")
    for i in range(0, len(x), 6):
        ax.plot([x[i], x[i]], [curve_a[i], curve_b[i]], color=PUB_COLORS["sage"],
                linewidth=0.9, alpha=0.8)
    ax.set_title(_wrap("Model 1: Hard-DTW \u2014 single minimum-cost alignment path", 42))
    ax.set_xlabel("Cultivation day (schematic)")
    ax.set_ylabel("Band signal (a.u., schematic)")
    ax.legend(frameon=False, loc="upper left")
    ax.grid(True)

    ax = axes[1]
    ax.plot(x, curve_a, color=PUB_COLORS["olive"], linewidth=2.2)
    ax.plot(x, curve_b, color=PUB_COLORS["ochre"], linewidth=2.2)
    for i in range(0, len(x), 5):
        for j in range(max(0, i - 3), min(len(x), i + 4)):
            ax.plot([x[i], x[j]], [curve_a[i], curve_b[j]], color=PUB_COLORS["brown"],
                    linewidth=0.4, alpha=0.18)
```

```

ax.set_title(_wrap(f"Model 2: Soft-DTW (gamma={gamma_used:.3g}) \u2014 weighted "
                  f"expectation over all paths", 42))
ax.set_xlabel("Cultivation day (schematic)")
ax.grid(True)

ax = axes[2]
ax.plot(x, curve_a, color=PUB_COLORS["olive"], linewidth=2.2)
ax.plot(x, curve_b, color=PUB_COLORS["ochre"], linewidth=2.2)
worst_i = int(np.argmax(np.abs(curve_a - curve_b)))
ax.plot([x[worst_i], x[worst_i]], [curve_a[worst_i], curve_b[worst_i]],
        color=PUB_COLORS["sage"], linewidth=2.6)
ax.annotate("complete-linkage:\nmax pairwise distance",
            xy=(x[worst_i], (curve_a[worst_i] + curve_b[worst_i]) / 2),
            xytext=(0.4, 0.1), textcoords="axes fraction", fontsize=8,
            arrowprops=dict(arrowstyle="->", color=PUB_COLORS["olive"], lw=1))
ax.set_title(_wrap("Model 3: Hierarchical/DTW \u2014 complete-linkage distance", 42))
ax.set_xlabel("Cultivation day (schematic)")
ax.grid(True)

fig.suptitle("Figure 1 \u2014 Algorithmic mechanics of the three DTW clustering strategies",
            fontsize=12)
_save_pub_figure(fig, out_stub, out_dir=out_dir)

```

```
fig1_algorithm_mechanics(res_m2["best_gamma"])
```

Fig. 3 — DTW-based silhouette profiles

```

def fig11_silhouette_profiles(traj_array, species_order, res_m1, res_m2, res_m3,
                              out_stub="fig11_silhouette_profiles", out_dir=OUTPUT_DIR):
    _apply_pub_style()

    fig, axes = plt.subplots(1, 3, figsize=(14, 5), dpi=PUB_DPI, constrained_layout=True)

```

```

for ax, res, title in zip(axes, (res_m1, res_m2, res_m3),
                           ("Model 1: Hard-DTW", "Model 2: Soft-DTW", "Model 3: Hierarchical/DTW")):
    labels = np.asarray(res["cluster_labels"])
    flat = res["flat"]
    unique_clusters = sorted(np.unique(labels))
    if len(unique_clusters) < 2:
        ax.text(0.5, 0.5, "single cluster --\nsilhouette undefined", ha="center", va="center")
        ax.set_title(title); continue
    sample_scores = silhouette_samples(flat, labels)
    avg_score = sample_scores.mean()
    y_lower = 10
    for c_i, c in enumerate(unique_clusters):
        c_scores = np.sort(sample_scores[labels == c])[::-1]
        y_upper = y_lower + len(c_scores)
        color, hatch = _cluster_color(c_i)
        ax.fill_betweenx(np.arange(y_lower, y_upper), 0, c_scores, facecolor=color,
                          hatch=hatch, label=f"C {c}")
        y_lower = y_upper + 5
    ax.axvline(avg_score, color=PUB_COLORS["brown"], linestyle="--", linewidth=2,
               label=f"avg={avg_score:.3f}")
    ax.set_title(title, fontsize=10.5)
    ax.set_xlabel("Silhouette coefficient")
    ax.set_xlim(min(-0.2, sample_scores.min() - 0.05), 1.0)
    ax.yaxis.set_visible(False)
    ax.grid(True, axis="x")
    ax.legend(loc="upper right", fontsize=7.2)
axes[0].set_ylabel("Species (DTW-scaled)")
fig.suptitle(_wrap("Figure 2 \u2014 DTW-based silhouette profile per model "
                  "(performance comparison)", 90), fontsize=11.5)
_save_pub_figure(fig, out_stub, out_dir=out_dir)

```

```
fig11_silhouette_profiles(traj_array, species_order, res_m1, res_m2, res_m3)
```

Fig. 4 — Sankey diagram of cluster flow

```
def _sankey_node_layout(labels, pad=0.8):  
    counts = pd.Series(labels).value_counts().sort_index()  
  
    layout, cursor = {}, 0.0  
    for cluster_id, n in counts.items():  
        layout[int(cluster_id)] = (cursor, cursor + n)  
        cursor += n + pad  
    return layout, dict(counts), (cursor - pad if len(counts) else 0.0)  
  
def _sankey_ribbon(ax, x0, x1, y0_top, y0_bot, y1_top, y1_bot, color, alpha=0.55, n=60):  
    xs = np.linspace(x0, x1, n)  
    t = (xs - x0) / (x1 - x0)  
    s = t * t * (3 - 2 * t)  
    top = y0_top + (y1_top - y0_top) * s  
    bot = y0_bot + (y1_bot - y0_bot) * s  
    ax.fill_between(xs, top, bot, color=color, alpha=alpha, linewidth=0, zorder=1)  
  
def fig2_cluster_flow(species_order, res_m1, res_m2, res_m3, consensus_df,  
                      out_stub="fig2_cluster_flow", out_dir=OUTPUT_DIR):  
    _apply_pub_style()  
  
    def _labels_for(res):  
        d = dict(zip(res["species_order"], np.asarray(res["cluster_labels"])))  
        return [int(d[s]) for s in species_order]
```

```

cons_map = _species_to_consensus(consensus_df)

stages = [_labels_for(res_m1), _labels_for(res_m2), _labels_for(res_m3),
          [int(cons_map[s]) for s in species_order]]

stage_prefixes = ["M1", "M2", "M3", "CN"]

stage_titles = [f"Model 1: Hard-DTW (k={len(set(stages[0]))})",
                 f"Model 2: Soft-DTW (k={len(set(stages[1]))})",
                 f"Model 3: Hierarchical (k={len(set(stages[2]))})",
                 f"Consensus (k={len(set(stages[3]))})"]

n_species = len(species_order)

node_width, gap = 0.22, 2.6

x_pos = [i * gap for i in range(4)]

layouts, counts_per_stage, heights = [], [], []

for labels in stages:
    layout, counts, h = _sankey_node_layout(labels)

    layouts.append(layout); counts_per_stage.append(counts); heights.append(h)

max_height = max(heights) if heights else n_species

fig_h = max(6.8, max_height * 0.42)

fig, ax = plt.subplots(figsize=(15.5, fig_h), dpi=PUB_DPI, constrained_layout=True)

for s in range(3):
    labels_l, labels_r = stages[s], stages[s + 1]

    layout_l, layout_r = layouts[s], layouts[s + 1]

    trans = {}

    for i in range(n_species):
        key = (labels_l[i], labels_r[i])

        trans[key] = trans.get(key, 0) + 1

    left_cursor = {c: layout_l[c][0] for c in layout_l}

```

```

right_cursor = {c: layout_r[c][0] for c in layout_r}

for (c_l, c_r) in sorted(trans.keys()):
    flow_n = trans[(c_l, c_r)]

    y0_top = left_cursor[c_l]; y0_bot = y0_top + flow_n; left_cursor[c_l] = y0_bot
    y1_top = right_cursor[c_r]; y1_bot = y1_top + flow_n; right_cursor[c_r] = y1_bot

    color, _h = _cluster_color(c_l)

    _sankey_ribbon(ax, x_pos[s] + node_width, x_pos[s + 1] - node_width,
                    y0_top, y0_bot, y1_top, y1_bot, color=color)

for s_idx, (layout, counts) in enumerate(zip(layouts, counts_per_stage)):
    for cluster_id, (y_top, y_bot) in layout.items():
        color, hatch = _cluster_color(cluster_id)
        ax.add_patch(mpatches.FancyBboxPatch(
            (x_pos[s_idx] - node_width, y_top), node_width * 2, (y_bot - y_top),
            boxstyle="square,pad=0.0", linewidth=1.3, edgecolor=PUB_COLORS["olive"],
            facecolor=color, hatch=hatch, alpha=0.95, zorder=3))
        n_here = counts[cluster_id]
        ax.text(x_pos[s_idx], (y_top + y_bot) / 2, f"{stage_prefixes[s_idx]}-C{cluster_id}\n(n={n_here})",
                ha="center", va="center", fontsize=7.4, fontweight="bold",
                color=PUB_COLORS["white"], zorder=4)

for s_idx, title in enumerate(stage_titles):
    ax.text(x_pos[s_idx], -1.6, title, ha="center", fontsize=9.5,
            fontweight="bold", color=PUB_COLORS["olive"])

ax.set_xlim(x_pos[0] - 1.0, x_pos[-1] + 1.0)
ax.set_ylim(max_height + 0.5, -2.6)
ax.axis("off")

fig.suptitle(_wrap("Figure 2 \u2014 Sankey diagram of species flow across the 3 models "
                    "and consensus (ribbon width = species making that transition)", 100),

```

```

        fontsize=11.5)

_save_pub_figure(fig, out_stub, out_dir=out_dir)

```

```
fig2_cluster_flow(species_order, res_m1, res_m2, res_m3, consensus_df)
```

Fig. 5 — Radar fingerprints of consensus clusters

```

def _save_pub_figure(fig, stub, out_dir, dpi=PUB_DPI, show_inline=True):
    """Creates out_dir if missing, saves high-DPI .png + vector .pdf,
    optionally displays inline in Jupyter, then closes the figure."""
    os.makedirs(out_dir, exist_ok=True)

    for ax in fig.get_axes():
        for spine in ax.spines.values():
            spine.set_color(PUB_COLORS["olive"])

    png_path = os.path.join(out_dir, f"{stub}.png")
    pdf_path = os.path.join(out_dir, f"{stub}.pdf")
    fig.savefig(png_path, dpi=dpi, bbox_inches="tight", facecolor=PUB_COLORS["white"])
    fig.savefig(pdf_path, bbox_inches="tight", facecolor=PUB_COLORS["white"])
    print(f" saved: {png_path}")
    print(f" saved: {pdf_path}")

    if show_inline:
        try:
            from IPython.display import display

            display(fig) # actually shows it in the notebook output
        except ImportError:
            pass # not running in Jupyter/IPython -- fine, just skip

    plt.close(fig)

```

```

def fig9_radar_fingerprints(traj_array, species_order, day_order, band_names, consensus_df,
                           cluster_col="Consensus_Cluster",
                           out_stub="fig9_radar_fingerprints",
                           out_dir="figures_fixed", # <- new folder, created automatically
                           dpi=PUB_DPI,
                           show_inline=True):
    _apply_pub_style()
    sp_to_cl = _species_to_consensus(consensus_df, cluster_col)
    clusters = sorted(consensus_df[cluster_col].unique())

    n_bands = len(band_names)
    descriptor = np.zeros((len(clusters), n_bands))
    days = np.asarray(day_order, dtype=float)
    for b_i in range(n_bands):
        cluster_means = []
        for c in clusters:
            idx = [i for i, s in enumerate(species_order) if sp_to_cl[s] == c]
            auc_vals = [_trapz(traj_array[i, :, b_i], days) for i in idx]
            cluster_means.append(np.mean(auc_vals))
        cluster_means = np.array(cluster_means, dtype=float)
        rng = cluster_means.max() - cluster_means.min()
        descriptor[:, b_i] = (cluster_means - cluster_means.min()) / rng if rng > 1e-9 else 0.5

    angles = np.linspace(0, 2 * np.pi, n_bands, endpoint=False).tolist()
    angles += angles[:1]

    fig, ax = plt.subplots(figsize=(8.5, 8.5), dpi=dpi, subplot_kw=dict(polar=True),
                             constrained_layout=True)

    for c_i, c in enumerate(clusters):

```

```

vals = descriptor[c_i].tolist(); vals += vals[:1]

color, hatch = _cluster_color(c_i)

linestyle = "-" if (c_i // len(PUB_CYCLE)) == 0 else "--"

ax.plot(angles, vals, color=color, linewidth=2.2, linestyle=linestyle,
        label=f"Cluster {c}")

ax.fill(angles, vals, color=color, alpha=0.15, hatch=hatch)

ax.set_xticks(angles[:-1])

ax.set_xticklabels([_wrap(b.replace("_", " "), 14) for b in band_names], fontsize=8.5)

ax.set_yticklabels([])

ax.set_ylim(0, 1)

ax.legend(loc="upper right", bbox_to_anchor=(1.35, 1.1), frameon=True, fontsize=8.5)

fig.suptitle(_wrap("Figure 9 \u2014 Radar fingerprint of normalized kinetic descriptors "
                  "(band AUC per consensus cluster", 80), fontsize=11.5)

_save_pub_figure(fig, out_stub, out_dir=out_dir, dpi=dpi, show_inline=show_inline)

```

```
fig9_radar_fingerprints(traj_array, species_order, day_order, band_names, consensus_df)
```

Fig. 6 — Taxon composition of consensus clusters

```

def fig6_taxonomy_vs_consensus(consensus_df, cluster_col="Consensus_Cluster",
                               taxonomy=None, out_stub="fig6_taxonomy_vs_consensus",
                               out_dir=OUTPUT_DIR):

    _apply_pub_style()

    if taxonomy is None:

        raise ValueError("fig6_taxonomy_vs_consensus needs a species->taxon dict "
                          "(pass taxonomy=SPECIES_TAXONOMY from your training script).")

    df = consensus_df.copy()

    df["Taxon"] = df["Species"].map(taxonomy)

```

```

df = df.dropna(subset=["Taxon"])

if df["Taxon"].nunique() < 2 or df[cluster_col].nunique() < 2:
    print(" [fig6] not enough distinct taxa/clusters -- skipped.")
    return

from sklearn.metrics import adjusted_rand_score as _ari, normalized_mutual_info_score as _nmi
le_map = {t: i for i, t in enumerate(sorted(df["Taxon"].unique()))}
ari = _ari(df[cluster_col].values, df["Taxon"].map(le_map).values)
nmi = _nmi(df[cluster_col].values, df["Taxon"].map(le_map).values)

crosstab = pd.crosstab(df["Taxon"], df[cluster_col])

fig, ax = plt.subplots(figsize=(10.5, 6), dpi=PUB_DPI, constrained_layout=True)
clusters = crosstab.columns.tolist()
taxa = crosstab.index.tolist()
x = np.arange(len(clusters))
bottom = np.zeros(len(clusters))
for t_i, taxon in enumerate(taxa):
    color, hatch = _cluster_color(t_i)
    vals = crosstab.loc[taxon, clusters].values.astype(float)
    ax.bar(x, vals, bottom=bottom, color=color, edgecolor=PUB_COLORS["olive"],
           hatch=hatch, label=_wrap(taxon, 26), width=0.6)
    bottom += vals

ax.set_xticks(x); ax.set_xticklabels([f"C{c}" for c in clusters])
ax.set_ylabel("Number of species")
ax.set_xlabel("Consensus cluster")
ax.legend(frameon=False, fontsize=7.3, title="Taxon", loc="upper left",
         bbox_to_anchor=(1.01, 1.0))
ax.grid(True, axis="y")
fig.suptitle(_wrap(f"Figure 6 \u2014 Taxon composition per consensus cluster "))

```

```

f'(ARI={ari:.3f}, NMI={nmi:.3f})", 90), fontsize=11.5)
_save_pub_figure(fig, out_stub, out_dir=out_dir)

```

```
fig6_taxonomy_vs_consensus(consensus_df, "Consensus_Cluster", taxonomy=SPECIES_TAXONOMY)
```

Fig. 7 — NDVI, GNDVI and AUC boxplots (Kruskal-Wallis)

```

def fig4_pigment_growth_boxplots(ds, traj_array, species_order, day_order, band_names,
                                consensus_df, cluster_col="Consensus_Cluster",
                                out_stub="fig4_pigment_growth_boxplots", out_dir=OUTPUT_DIR):
    _apply_pub_style()
    meta = ds.meta_dataframe()
    pig_df = pd.DataFrame(ds.pigment_indices, columns=["NDVI", "GNDVI", "Simple_Ratio"])
    pig_df["Species"] = meta["Species"].values
    species_pig = pig_df.groupby("Species").mean().reset_index()
    pig_merged = species_pig.merge(consensus_df[["Species", cluster_col]], on="Species", how="inner")

    growth_df = compute_growth_curve_features(traj_array, species_order, day_order, band_names,
                                              growth_band="auto")
    growth_merged = growth_df.merge(consensus_df[["Species", cluster_col]], on="Species", how="inner")

    def _kw_p(merged, col):
        groups = [g[col].dropna().values for _, g in merged.groupby(cluster_col)]
        groups = [g for g in groups if len(g) > 0]
        if len(groups) > 1:
            try:
                _stat, p = kruskal(*groups)
                return float(p)
            except ValueError:
                return None
        return None

```

```

fig, axes = plt.subplots(1, 3, figsize=(13.5, 4.8), dpi=PUB_DPI, constrained_layout=True)
clusters = sorted(pig_merged[cluster_col].unique())

def _box(ax, merged, col, title):
    data = [merged.loc[merged[cluster_col] == c, col].dropna().values for c in clusters]
    bp = ax.boxplot(data, patch_artist=True, widths=0.55)
    for i, box in enumerate(bp["boxes"]):
        color, hatch = _cluster_color(i)
        box.set(facecolor=color, edgecolor=PUB_COLORS["olive"], hatch=hatch, alpha=0.92)
    for element in ("whiskers", "caps"):
        for line in bp[element]:
            line.set(color=PUB_COLORS["olive"])
    for median in bp["medians"]:
        median.set(color=PUB_COLORS["white"], linewidth=1.8)
    for flier in bp["fliers"]:
        flier.set(markeredgecolor=PUB_COLORS["olive"], markerfacecolor=PUB_COLORS["cream"])
    ax.set_xticks(range(1, len(clusters) + 1))
    ax.set_xticklabels([f"C{c}" for c in clusters])
    p_val = _kw_p(merged, col)
    p_str = f"p={p_val:.4f}" if p_val is not None else "p=NA"
    ax.set_title(f"{title} ({p_str})", fontsize=10.5)
    ax.grid(True, axis="y")

_box(axes[0], pig_merged, "NDVI", "NDVI")
_box(axes[1], pig_merged, "GNDVI", "GNDVI")
_box(axes[2], growth_merged, "AUC", "AUC (growth accumulation)")

fig.suptitle(_wrap("Figure 7 \u2014 Independent biological validation across consensus "
                  "clusters (Kruskal-Wallis)", 90), fontsize=11.5)

```

```
_save_pub_figure(fig, out_stub, out_dir=out_dir)
```

```
fig4_pigment_growth_boxplots(ds, traj_array, species_order, day_order, band_names, consensus_df)
```

Fig. 8 — Lag / μ -max / stationary-phase timing per strain

```
def fig17_kinetic_inflection_analysis(traj_array, species_order, day_order, band_names, consensus_df,
                                     cluster_col="Consensus_Cluster",
                                     out_stub="fig17_kinetic_inflection_analysis", out_dir=OUTPUT_DIR):
    _apply_pub_style()
    growth_df = compute_growth_curve_features(traj_array, species_order, day_order, band_names,
                                              growth_band="auto")
    band_used = growth_df.attrs["growth_band_used"]
    b_idx = band_names.index(band_used)
    days = np.asarray(day_order, dtype=float)
    sp_to_cl = _species_to_consensus(consensus_df, cluster_col)

    rows = []
    for i, sp in enumerate(species_order):
        y = traj_array[i, :, b_idx]
        dy = np.gradient(y, days)
        t_mumax = days[int(np.argmax(dy))]
        thresh = y.min() + 0.1 * (y.max() - y.min())
        above = np.where(y >= thresh)[0]
        t_lag = days[above[0]] if len(above) else days[0]
        t_stat = days[int(np.argmax(y))]
        rows.append({"Species": sp, "Cluster": sp_to_cl.get(sp), "T_lag": t_lag,
                    "T_mu_max": t_mumax, "T_stat": t_stat})
    timing_df = pd.DataFrame(rows).dropna(subset=["Cluster"])
    timing_df = timing_df.sort_values(["Cluster", "Species"])
```

```

fig, ax = plt.subplots(figsize=(11, 0.42 * len(timing_df) + 2.5), dpi=PUB_DPI,
                        constrained_layout=True)
y_pos = np.arange(len(timing_df))
clusters = sorted(timing_df["Cluster"].unique())
cluster_color = {c: _cluster_color(i)[0] for i, c in enumerate(clusters)}

ax.barh(y_pos, timing_df["T_stat"] - timing_df["T_lag"], left=timing_df["T_lag"],
        color=[cluster_color[c] for c in timing_df["Cluster"]], alpha=0.35, height=0.6,
        edgecolor=PUB_COLORS["olive"])
ax.scatter(timing_df["T_lag"], y_pos, marker="|", s=120, color=PUB_COLORS["olive"], label="T_lag")
ax.scatter(timing_df["T_mu_max"], y_pos, marker="o", s=45, color=PUB_COLORS["ochre"],
        edgecolor=PUB_COLORS["olive"], label="T_mu-max", zorder=3)
ax.scatter(timing_df["T_stat"], y_pos, marker="s", s=45, color=PUB_COLORS["brown"],
        edgecolor=PUB_COLORS["olive"], label="T_stat", zorder=3)

ax.set_yticks(y_pos)
ax.set_yticklabels([f"{{s}} (C{{c}})" for s, c in zip(timing_df["Species"], timing_df["Cluster"])],
                    fontsize=7.5)
ax.set_xlabel(f"Cultivation day (timing derived from '{{band_used}}')")
ax.legend(frameon=True, fontsize=8.5, loc="upper right")
ax.grid(True, axis="x")
fig.suptitle(_wrap("Figure 17 (S9 support) \u2014 Lag / mu-max / stationary-phase timing "
                  "per species, grouped by consensus cluster", 85), fontsize=11)
_save_pub_figure(fig, out_stub, out_dir=out_dir)

```

```
fig17_kinetic_inflection_analysis(traj_array, species_order, day_order, band_names, consensus_df)
```

Fig. S1 — Co-association matrix

```

def _build_coassociation(species_order, *label_sets):
    n = len(species_order)

```

```

coassoc = np.zeros((n, n))

idx = {s: i for i, s in enumerate(species_order)}

for sp_order, labels in label_sets:
    d = dict(zip(sp_order, np.asarray(labels)))
    for a in species_order:
        for b in species_order:
            if d.get(a) is not None and d.get(a) == d.get(b):
                coassoc[idx[a], idx[b]] += 1

return coassoc / len(label_sets)

```

```

def fig10_coassociation_heatmap(species_order, res_m1, res_m2, res_m3, consensus_df,
                                cluster_col="Consensus_Cluster",
                                out_stub="fig10_coassociation_heatmap", out_dir=OUTPUT_DIR):
    _apply_pub_style()
    coassoc = _build_coassociation(
        species_order,
        (res_m1["species_order"], res_m1["cluster_labels"]),
        (res_m2["species_order"], res_m2["cluster_labels"]),
        (res_m3["species_order"], res_m3["cluster_labels"]),
    )
    sp_to_cl = _species_to_consensus(consensus_df, cluster_col)
    order = sorted(range(len(species_order)), key=lambda i: (sp_to_cl[species_order[i]], species_order[i]))
    sorted_species = [species_order[i] for i in order]
    sorted_coassoc = coassoc[np.ix_(order, order)]

    n = len(species_order)
    fig, ax = plt.subplots(figsize=(0.4 * n + 2.5, 0.4 * n + 2.5), dpi=PUB_DPI,
                            constrained_layout=True)
    cmap = matplotlib.colors.LinearSegmentedColormap.from_list(

```

```

    "pub_seq", [PUB_COLORS["white"], PUB_COLORS["cream"], PUB_COLORS["ochre"],
PUB_COLORS["olive"]])

im = ax.imshow(sorted_coassoc, cmap=cmap, vmin=0, vmax=1)

ax.set_xticks(range(n)); ax.set_xticklabels(sorted_species, rotation=90, fontsize=7)

ax.set_yticks(range(n)); ax.set_yticklabels(sorted_species, fontsize=7)

cbar = fig.colorbar(im, ax=ax, fraction=0.046, pad=0.04)

cbar.set_label("Cross-model agreement fraction")

fig.suptitle(_wrap("Figure 10 (S4) \u2014 Cross-model co-association matrix, species "
    "sorted by consensus cluster", 80), fontsize=11)

_save_pub_figure(fig, out_stub, out_dir=out_dir)

```

fig10_coassociation_heatmap(species_order, res_m1, res_m2, res_m3, consensus_df)

Fig. S2 — Elastic-distance metrics vs. consensus (DTW, LCSS, EDR, TWED)

```
def compute_pairwise_dtw_matrix(traj_array, species_order):
```

```

    scaler = TimeSeriesScalerMeanVariance()

    traj_scaled = scaler.fit_transform(traj_array)

    D = cdist_dtw(traj_scaled)

    return D

```

```
def _lcss_dist(a, b, eps):
```

```

    n, m = len(a), len(b)

    L = np.zeros((n + 1, m + 1))

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if abs(a[i - 1] - b[j - 1]) <= eps:
                L[i, j] = L[i - 1, j - 1] + 1
            else:
                L[i, j] = max(L[i - 1, j], L[i, j - 1])

    lcss_len = L[n, m]

```

```
return 1.0 - lcss_len / min(n, m)
```

```
def _edr_dist(a, b, eps):  
    n, m = len(a), len(b)  
    E = np.zeros((n + 1, m + 1))  
    E[:, 0] = np.arange(n + 1); E[0, :] = np.arange(m + 1)  
    for i in range(1, n + 1):  
        for j in range(1, m + 1):  
            cost = 0 if abs(a[i - 1] - b[j - 1]) <= eps else 1  
            E[i, j] = min(E[i - 1, j - 1] + cost, E[i - 1, j] + 1, E[i, j - 1] + 1)  
    return E[n, m] / max(n, m)
```

```
def _twed_dist(a, b, nu=0.001, lam=1.0):  
    n, m = len(a), len(b)  
    ta, tb = np.arange(n), np.arange(m)  
    T = np.full((n + 1, m + 1), np.inf)  
    T[0, 0] = 0  
    for i in range(1, n + 1):  
        T[i, 0] = T[i - 1, 0] + abs(a[i - 1] - (a[i - 2] if i > 1 else a[0])) + nu + lam  
    for j in range(1, m + 1):  
        T[0, j] = T[0, j - 1] + abs(b[j - 1] - (b[j - 2] if j > 1 else b[0])) + nu + lam  
    for i in range(1, n + 1):  
        for j in range(1, m + 1):  
            d_match = abs(a[i - 1] - b[j - 1]) + abs((a[i - 2] if i > 1 else a[0]) - (b[j - 2] if j > 1 else b[0]))  
            d_del_a = abs(a[i - 1] - (a[i - 2] if i > 1 else a[0])) + nu + lam  
            d_del_b = abs(b[j - 1] - (b[j - 2] if j > 1 else b[0])) + nu + lam  
            T[i, j] = min(T[i - 1, j - 1] + d_match, T[i - 1, j] + d_del_a, T[i, j - 1] + d_del_b)  
    return T[n, m]
```

```

def fig14_elastic_metric_benchmark(traj_array, species_order, day_order, consensus_df,
                                   cluster_col="Consensus_Cluster",
                                   out_stub="fig14_elastic_metric_benchmark", out_dir=OUTPUT_DIR):
    _apply_pub_style()
    scaler = TimeSeriesScalerMeanVariance()
    traj_scaled = scaler.fit_transform(traj_array)
    b_idx = 0
    series = [traj_scaled[i, :, b_idx] for i in range(len(species_order))]
    n = len(species_order)
    eps = 0.5

    D_dtw = compute_pairwise_dtw_matrix(traj_array, species_order)
    D_lcss = np.zeros((n, n)); D_edr = np.zeros((n, n)); D_twed = np.zeros((n, n))
    for i in range(n):
        for j in range(i + 1, n):
            D_lcss[i, j] = D_lcss[j, i] = _lcss_dist(series[i], series[j], eps)
            D_edr[i, j] = D_edr[j, i] = _edr_dist(series[i], series[j], eps)
            D_twed[i, j] = D_twed[j, i] = _twed_dist(series[i], series[j])

    sp_to_cl = _species_to_consensus(consensus_df, cluster_col)
    consensus_labels = np.array([sp_to_cl[s] for s in species_order])
    n_clusters = len(np.unique(consensus_labels))

    aris = {}
    for name, Dm in [("DTW", D_dtw), ("LCSS", D_lcss), ("EDR", D_edr), ("TWED", D_twed)]:
        dist = Dm.copy(); np.fill_diagonal(dist, 0)
        Z = scipy_linkage(squareform(dist, checks=False), method="average")
        labels = fcluster(Z, t=n_clusters, criterion="maxclust")

```

```

    aris[name] = adjusted_rand_score(consensus_labels, labels)

fig, ax = plt.subplots(figsize=(8, 5.4), dpi=PUB_DPI, constrained_layout=True)
names = list(aris.keys())
for i, name in enumerate(names):
    color, hatch = _cluster_color(i)
    ax.bar(name, aris[name], color=color, edgecolor=PUB_COLORS["olive"], hatch=hatch, width=0.6)
ax.set_ylabel("Adjusted Rand Index vs. consensus clustering")
ax.set_ylim(min(0, min(aris.values()) - 0.05), 1.05)
ax.axhline(0, color=PUB_COLORS["olive"], linewidth=0.8)
ax.grid(True, axis="y")
fig.suptitle(_wrap("Figure 14 (S7) \u2014 Agreement of alternative elastic distance "
    "metrics with the consensus partition", 80), fontsize=11.5)
_save_pub_figure(fig, out_stub, out_dir=out_dir)

```

```
fig14_elastic_metric_benchmark(traj_array, species_order, day_order, consensus_df)
```

Fig. S3 — PCA projection per model and consensus

```

def plot_model_pca_projection_individual(traj_array, species_order, cluster_labels, model_name,
    out_stub, out_dir=OUTPUT_DIR):
    _apply_pub_style()
    scaler = TimeSeriesScalerMeanVariance()
    traj_scaled = scaler.fit_transform(traj_array)
    flat = traj_scaled.reshape(traj_scaled.shape[0], -1)
    pca = PCA(n_components=2, random_state=SEED)
    coords = pca.fit_transform(flat)
    var_exp = pca.explained_variance_ratio_ * 100

    cluster_labels = np.asarray(cluster_labels)
    unique_clusters = sorted(np.unique(cluster_labels))

```

```

fig, ax = plt.subplots(figsize=(8.8, 7.2), dpi=PUB_DPI, constrained_layout=True)

for c_i, c in enumerate(unique_clusters):
    idx = np.where(cluster_labels == c)[0]

    color, hatch = _cluster_color(c_i)

    ax.scatter(coords[idx, 0], coords[idx, 1], s=120, color=color,

               edgecolor=PUB_COLORS["olive"], linewidth=1.0, label=f"Cluster {c}", zorder=3)

    for i in idx:
        ax.annotate(species_order[i], (coords[i, 0], coords[i, 1]), fontsize=7.5,

                   xytext=(4, 4), textcoords="offset points", color=PUB_COLORS["olive"])

ax.set_xlabel(f"PC1 ( {var_exp[0]:.1f}% variance)")
ax.set_ylabel(f"PC2 ( {var_exp[1]:.1f}% variance)")
ax.set_title(_wrap(f"{model_name} \u2014 PCA latent-space projection of species trajectories", 60))
ax.grid(True, alpha=0.5)
ax.legend(loc="best", frameon=True, fontsize=8.5)

_save_pub_figure(fig, f"{out_stub}_pca_latent", out_dir=out_dir)

```

```

methods = [
    ("Model 1: Hard-DTW K-Means", res_m1["result_df"], res_m1["cluster_col"]),
    ("Model 2: Soft-DTW K-Means", res_m2["result_df"], res_m2["cluster_col"]),
    ("Model 3: Hierarchical-DTW", res_m3["result_df"], res_m3["cluster_col"]),
    ("Consensus (3-model agreement)", consensus_df, "Consensus_Cluster"),
]

```

```

for label, result_df, cluster_col in methods:
    sp_to_cl = dict(zip(result_df["Species"], result_df[cluster_col]))
    labels = [sp_to_cl[s] for s in species_order]

    plot_model_pca_projection_individual(traj_array, species_order, labels,

                                       label, label.split(":")[0].replace(" ", "_"))

```