AetherLink Part III

Supporting Source Code Companion

SMAF v3.7 & StormMode Forecaster v9.0 Public

Exact public Python implementations used for the dual-engine analysis
of the 25 validated RAST plasmoid sightings (February–March 2026 + November 2025)

UCRC Institute • Resonant Technologies Inc. (RTI)
K. Brett Boswell • Christopher M. Wulf

26–27 July 2026

Companion document to AetherLink Part III Version 4.6

Purpose of This Document

These are the exact public-source Python implementations of the two independent analysis engines used throughout AetherLink Part III (Version 4.6). SMAF v3.7 provides retrospective geomagnetic characterization from official GFZ Potsdam definitive Kp, computes the lag metric, geoeffectiveness, storm typing, and incorporates a hybrid weather/orographic layer. StormMode Forecaster v9.0 Public generates the Enhanced Natural Plasmoid Index (eNPI), Persistence Index, hybrid lag estimate, mode probabilities (A/B/C/D), and geometric weight fraction (Beltrami coherence + Znidarsic match, historically 28–33 %). Both engines were applied to the ±1-day windows of the 25 validated plasmoid sightings. Full source is provided below for transparency and reproducibility.

Note on StormMode v9.0 Public: All Cosmic Lattice Web (CLW) content and ESCF projection methods have been removed (unpublished). The release retains only publicly documented UCRC corpus elements (AetherLink Part II UCRC-CG, APR/RAST v.4, Plasma/Anti-Plasma Dialect v3, Undead Geometry public layer, and classical Cartan/Beltrami/Znidarsic anchors).

This source package is released as a companion to the main technical report so that readers of Page38News and the broader research community can inspect the precise algorithms that produced the lag metrics, eNPI values, Persistence Indices, and geometric weights reported in AetherLink Part III.

1. SMAF v3.7 — StormMode AetherLink Forecaster

Version 3.7 (July 2026) — Hybrid Historical Weather Edition. SMAF answers the question: “What was the geomagnetic and space-weather environment doing on these days?” It ingests official GFZ definitive Kp, computes geoeffectiveness, classifies storm type, calculates the lag metric (longest run of consecutive days with Kp ≥ 4.5), and supports a hybrid weather layer with orographic lift derived from wind direction versus local ridge orientation. Output is a multi-page neon-styled PDF report for each analysis window.

Key capabilities retained in v3.7: GFZ Kp ingestion, storm typing, lag metric, NOAA SWPC snapshot (Bz, Vsw, X-ray, electrons), hybrid weather (user-supplied or automatic attempt), orographic lift calculation, Schumann Resonance notes/image support, and full neon PDF report generation.

Full Source Code — SMAF_v3.7.py

BEGIN SMAF v3.7 SOURCE

```
#!/usr/bin/env python3
"""
SMAF v3.7 — StormMode AetherLink Forecaster
==========================================
Hybrid Historical Weather Edition

New in v3.7:
- Hybrid weather layer:
    1. Attempts automatic historical retrieval for the zone’s nearest station
    2. Accepts full user-supplied weather dictionaries (same workflow as SR spectrograms)
- Weather fields: barometric pressure, wind direction, wind speed, temperature,
  humidity, precip, sky condition, notes
- Quality flags: "archived" | "user-supplied" | "climatology" | "unavailable"
- Orographic lift calculated from real or supplied wind direction
- Neon Full Report includes dedicated Weather & Orographic page
- All v3.6 capabilities retained (GFZ Kp, storm type, lag, SR images, neon graphs)

Version: 3.7 | July 2026
"""

from __future__ import annotations
import os
import textwrap
from datetime import datetime
from typing import Dict, List, Any, Optional
import warnings
warnings.filterwarnings("ignore")

try:
    import requests
    HAS_REQUESTS = True
except ImportError:
    HAS_REQUESTS = False

try:
    import numpy as np
    import matplotlib.pyplot as plt
    from matplotlib.backends.backend_pdf import PdfPages
    HAS_MPL = True
except ImportError:
    HAS_MPL = False

NEON_GREEN = "#00FF9F"
NEON_CYAN  = "#00FFFF"
HOT_PINK   = "#FF1493"
```

```
BRIGHT_YELLOW = "#FFD700"
DARK_BG    = "#0D1B2A"
WHITE      = "#FFFFFF"
ORANGE     = "#FF8C00"

VERSION = "3.7"
OUTPUT_DIR = "/home/workdir/artifacts/smaf_reports"
os.makedirs(OUTPUT_DIR, exist_ok=True)

ZONES = {
    "VirginiaEast": {
        "name": "Virginia East Coast / Virginia Beach-Norfolk / VACAPES",
        "lat": 36.85, "lon": -75.98, "ridge": 90,
        "station": "KORF", "station_name": "Norfolk International"
    },
    "Tucson": {
        "name": "Tucson / Southern Arizona",
        "lat": 32.22, "lon": -110.97, "ridge": 115,
        "station": "KTUS", "station_name": "Tucson International"
    },
    "Barksdale": {
        "name": "Barksdale AFB / Louisiana",
        "lat": 32.50, "lon": -93.67, "ridge": 90,
        "station": "KSHV", "station_name": "Shreveport Regional"
    },
}

# ---------------------------------------------------------------------------
# Data Layer
# ---------------------------------------------------------------------------
class DataLayer:
    GFZ = "https://kp.gfz.de/app/json/"

    def fetch_gfz_kp(self, start: str, end: str) -> Dict:
        if not HAS_REQUESTS:
            return {"ok": False, "error": "no requests"}
        try:
            url = f"{self.GFZ}?start={start}T00:00:00Z&end={end}T23:59:59Z&index=Kp&status=all"
            r = requests.get(url, timeout=15)
            r.raise_for_status()
            data = r.json()
            times = data.get("datetime", [])
            kps   = [float(x) for x in data.get("Kp", [])]
            stats = data.get("status", ["pre"] * len(times))
            daily = {}
            for t, k, s in zip(times, kps, stats):
                day = t[:10]
                if day not in daily:
                    daily[day] = {"kp_3hr": [], "status": s}
                daily[day]["kp_3hr"].append(k)
            for d, info in daily.items():
                vals = info["kp_3hr"]
                info["kp_max"] = max(vals)
                info["kp_mean"] = sum(vals) / len(vals)
                info["quality"] = "definitive" if info["status"] == "def" else "preliminary"
            return {"ok": True, "days": daily}
        except Exception as e:
            return {"ok": False, "error": str(e)[:160]}

    def get_noaa_snapshot(self) -> Dict:
        out = {"xray_max": "C1.0", "electrons": 1200.0, "bz": -3.0, "vsw": 420.0, "status": "FALLBACK"}
```

```
        if not HAS_REQUESTS:
            return out
        try:
            r = requests.get("https://services.swpc.noaa.gov/json/goes/primary/xray-flares-7-day.json", timeout=8)
            if r.ok and r.json():
                classes = [f.get("max_class", "C1") for f in r.json()[-20:]]
                rank = {"A": 0, "B": 1, "C": 2, "M": 3, "X": 4}
                out["xray_max"] = max(classes, key=lambda c: (rank.get(c[0], 0), float(c[1:] or 0)))
            r = requests.get("https://services.swpc.noaa.gov/json/goes/primary/integral-electrons-6-hour.json",
timeout=8)
            if r.ok and r.json():
                out["electrons"] = float(r.json()[-1].get("flux", 1200))
            r = requests.get("https://services.swpc.noaa.gov/products/solar-wind/mag-7-day.json", timeout=8)
            if r.ok and len(r.json()) > 1:
                latest = r.json()[-1]
                out["bz"] = float(latest[3])
                out["vsw"] = float(latest[2]) if len(latest) > 2 else 420
            out["status"] = "LIVE"
        except Exception:
            pass
        return out

    def attempt_historical_weather(self, station: str, start: str, end: str) -> Dict:
        """
        Lightweight attempt at historical weather.
        In practice most free endpoints do not return clean bulk historical JSON
        for arbitrary past windows without file downloads. This method therefore
        returns a structured 'unavailable' result so the caller can fall back
        to user-supplied values. Future versions can plug in ISD Lite / AWS
        open-data parsers here.
        """
        return {
            "ok": False,
            "quality": "unavailable",
            "source": "automatic archive not available in this environment",
            "note": "Supply weather manually via user_weather dict (recommended)."
        }

# ---------------------------------------------------------------------------
# Science helpers
# ---------------------------------------------------------------------------
def geoeffectiveness(kp, bz=-3.0, vsw=420.0):
    kp_s = min(kp / 9.0, 1.0) * 50
    bz_s = 45 if bz < -10 else (30 if bz < -5 else (15 if bz < 0 else max(0, 10 - bz * 2)))
    v_s = min(max(vsw - 300, 0) / 400, 1) * 20
    total = min(kp_s + bz_s + v_s, 100)
    level = "VERY HIGH" if total >= 75 else ("HIGH" if total >= 60 else ("MODERATE" if total >= 45 else "LOW"))
    return {"index": round(total, 1), "level": level}

def storm_type(kp, vsw=420, bz=-3, xray="C"):
    if kp < 4:
        return "Quiet / Background"
    high_speed = vsw > 550
    strong_south = bz < -8
    flare = xray and xray[0] in ("M", "X")
    if high_speed and not strong_south:
        return "High-Speed Stream (HSS)"
    if strong_south or (flare and kp >= 5):
        return "CME-driven / Flare-associated"
    if high_speed and strong_south:
        return "Mixed (HSS + CME)"
    return "Unsettled / Active"
```

```
def lag_metric(kps: List[float], threshold=4.5):
    above = [1 if k >= threshold else 0 for k in kps]
    max_c = cur = 0
    for v in above:
        if v:
            cur += 1
            max_c = max(max_c, cur)
        else:
            cur = 0
    return {"max_consecutive": max_c, "threshold": threshold, "days_above": sum(above)}

def orographic_lift(ridge: float, wind_dir_cardinal: str) -> float:
    cardinal = {
        "N": 0, "NNE": 22, "NE": 45, "ENE": 67, "E": 90, "ESE": 112,
        "SE": 135, "SSE": 157, "S": 180, "SSW": 202, "SW": 225,
        "WSW": 247, "W": 270, "WNW": 292, "NW": 315, "NNW": 337
    }
    wind_deg = cardinal.get(str(wind_dir_cardinal).upper().strip(), 180)
    diff = abs(wind_deg - ridge)
    if diff > 180:
        diff = 360 - diff
    return round(max(0.15, np.sin(np.deg2rad(diff))), 3)

# ---------------------------------------------------------------------------
# Report generator
# ---------------------------------------------------------------------------
def create_full_report(results, zone_name, start, end, noaa, lag, sr_notes,
                       weather_block, output_path, sr_image_path=None):
    if not HAS_MPL:
        print("matplotlib missing – PDF skipped")
        return


    plt.rcParams.update({
        "axes.facecolor": DARK_BG, "figure.facecolor": DARK_BG,
        "text.color": WHITE, "axes.labelcolor": WHITE,
        "xtick.color": WHITE, "ytick.color": WHITE,
        "axes.edgecolor": NEON_CYAN, "font.family": "DejaVu Sans"
    })

    with PdfPages(output_path) as pdf:
        # Page 1 – Title
        fig, ax = plt.subplots(figsize=(11, 8.5), facecolor=DARK_BG)
        ax.axis("off")
        ax.text(0.5, 0.95, f"SMAF v{VERSION} — Full Report (Hybrid Weather)",
                fontsize=15, fontweight="bold", ha="center", color=NEON_GREEN, transform=ax.transAxes)
        ax.text(0.5, 0.90, zone_name, fontsize=12, ha="center", color=NEON_CYAN, transform=ax.transAxes)
        ax.text(0.5, 0.86, f"Window: {start} → {end}  |  Kp: GFZ Potsdam definitive",
                fontsize=10, ha="center", color=HOT_PINK, transform=ax.transAxes)
        y = 0.78
        ax.text(0.08, y, "EXECUTIVE SUMMARY", fontsize=11, fontweight="bold", color=NEON_GREEN,
transform=ax.transAxes)
        y -= 0.03
        peak = max(results, key=lambda r: r["kp_max"])
        summary = (
            f"Peak Kp {peak['kp_max']:.2f} on {peak['date']} ({peak['storm_type']}). "
            f"Lag metric: {lag['max_consecutive']} day(s) ≥ {lag['threshold']}. "
            f"Weather source: {weather_block.get('quality', 'n/a')} — {weather_block.get('source', '')}."
        )
        for line in textwrap.wrap(summary, 95):
            ax.text(0.08, y, line, fontsize=9.2, color=WHITE, transform=ax.transAxes)
```

```
    y -= 0.025
pdf.savefig(fig, bbox_inches="tight", facecolor=DARK_BG)
plt.close(fig)

# Page 2 – Daily table
fig, ax = plt.subplots(figsize=(11, 8.5), facecolor=DARK_BG)
ax.axis("off")
ax.text(0.5, 0.95, "DAILY METRICS", fontsize=13, fontweight="bold",
        ha="center", color=NEON_GREEN, transform=ax.transAxes)
y = 0.88
header = f"{'Date':<12} {'Kp max':>8} {'Quality':<12} {'Geo':>7} {'Level':<10} {'Storm Type':<22}"
ax.text(0.05, y, header, fontsize=8.5, color=BRIGHT_YELLOW, transform=ax.transAxes, family="monospace")
y -= 0.03
for r in results:
    line = (f"{r['date']:<12} {r['kp_max']:>8.3f} {r['quality']:<12} "
            f"{r['geo']['index']:>7.1f} {r['geo']['level']:<10} {r['storm_type']:<22}")
    color = NEON_GREEN if r["kp_max"] >= 5 else (NEON_CYAN if r["kp_max"] >= 4 else WHITE)
    ax.text(0.05, y, line, fontsize=8.3, color=color, transform=ax.transAxes, family="monospace")
    y -= 0.025
y -= 0.02
ax.text(0.05, y, f"Lag Metric: {lag['max_consecutive']} consecutive day(s) ≥ {lag['threshold']}",
        fontsize=9.5, color=HOT_PINK, transform=ax.transAxes)
pdf.savefig(fig, bbox_inches="tight", facecolor=DARK_BG)
plt.close(fig)

# Page 3 – Kp graph
fig, ax = plt.subplots(figsize=(11, 6.5), facecolor=DARK_BG)
ax.set_facecolor(DARK_BG)
dates = [r["date"][5:] for r in results]
kps = [r["kp_max"] for r in results]
ax.plot(dates, kps, color=NEON_GREEN, lw=2.8, marker="o", ms=9, label="Kp max (GFZ)")
ax.axhline(5.0, color=HOT_PINK, ls="--", alpha=0.85, label="G1")
ax.axhline(6.0, color=BRIGHT_YELLOW, ls="--", alpha=0.7, label="G2")
ax.fill_between(dates, kps, alpha=0.18, color=NEON_GREEN)
ax.set_ylabel("Kp Index")
ax.set_title(f"SMAF v{VERSION} — Kp Time Series", fontsize=13, fontweight="bold", color=NEON_CYAN)
ax.legend(loc="upper right", fontsize=8, facecolor=DARK_BG, edgecolor=NEON_CYAN)
ax.grid(True, alpha=0.15, color=NEON_CYAN)
for s in ax.spines.values():
    s.set_color(NEON_CYAN)
plt.tight_layout()
pdf.savefig(fig, bbox_inches="tight", facecolor=DARK_BG)
plt.close(fig)

# Page 4 – Geoeffectiveness
fig, ax = plt.subplots(figsize=(11, 6.5), facecolor=DARK_BG)
ax.set_facecolor(DARK_BG)
geos = [r["geo"]["index"] for r in results]
colors = [NEON_GREEN if g >= 50 else NEON_CYAN if g >= 40 else WHITE for g in geos]
ax.bar(dates, geos, color=colors, edgecolor=NEON_CYAN, width=0.55)
ax.axhline(45, color=HOT_PINK, ls="--", alpha=0.7, label="Moderate")
ax.set_ylabel("Geoeffectiveness")
ax.set_title(f"SMAF v{VERSION} — Geoeffectiveness", fontsize=13, fontweight="bold", color=NEON_CYAN)
ax.legend(loc="upper right", fontsize=8, facecolor=DARK_BG, edgecolor=NEON_CYAN)
ax.grid(True, alpha=0.12, color=NEON_CYAN, axis="y")
for s in ax.spines.values():
    s.set_color(NEON_CYAN)
plt.tight_layout()
pdf.savefig(fig, bbox_inches="tight", facecolor=DARK_BG)
plt.close(fig)

# Page 5 – Weather page
```

```
        fig, ax = plt.subplots(figsize=(11, 8.5), facecolor=DARK_BG)
        ax.axis("off")
        ax.text(0.5, 0.95, "WEATHER & OROGRAPHIC CONTEXT (Hybrid Layer)", fontsize=13,
                fontweight="bold", ha="center", color=NEON_GREEN, transform=ax.transAxes)
        y = 0.88
        ax.text(0.06, y, f"Quality flag : {weather_block.get('quality', 'n/a')}", fontsize=10, color=HOT_PINK,
transform=ax.transAxes)
        y -= 0.03
        ax.text(0.06, y, f"Source       : {weather_block.get('source', 'n/a')}", fontsize=10, color=NEON_CYAN,
transform=ax.transAxes)
        y -= 0.04
        if weather_block.get("days"):
            ax.text(0.06, y, "Daily weather values:", fontsize=10, fontweight="bold", color=BRIGHT_YELLOW,
transform=ax.transAxes)
            y -= 0.03
            for day, w in weather_block["days"].items():
                line = (f"{day}: wind {w.get('wind_dir','?')} {w.get('wind_speed','?')} | "
                        f"pressure {w.get('pressure','?')} | temp {w.get('temp','?')} | "
                        f"humidity {w.get('humidity','?')} | lift {w.get('orographic_lift','?')}")
                ax.text(0.06, y, line, fontsize=8.2, color=WHITE, transform=ax.transAxes, family="monospace")
                y -= 0.025
        else:
            for line in textwrap.wrap(weather_block.get("note", "No weather data supplied."), 90):
                ax.text(0.06, y, line, fontsize=9.2, color=WHITE, transform=ax.transAxes)
                y -= 0.028
        y -= 0.03
        ax.text(0.06, y, "How to supply weather for future runs:", fontsize=10, fontweight="bold",
color=NEON_GREEN, transform=ax.transAxes)
        y -= 0.03
        help_txt = (
            "Pass a user_weather dictionary when calling SMAF.run(), e.g. "
            "user_weather={'2025-05-17': {'wind_dir':'SW', 'wind_speed':'12 mph', "
            "'pressure':'1012 hPa', 'temp':'78 F', 'humidity':'25%', 'notes':'dry'}}"
        )
        for line in textwrap.wrap(help_txt, 90):
            ax.text(0.06, y, line, fontsize=8.5, color=WHITE, transform=ax.transAxes)
            y -= 0.025
        pdf.savefig(fig, bbox_inches="tight", facecolor=DARK_BG)
        plt.close(fig)

        # Page 6 – Schumann
        fig, ax = plt.subplots(figsize=(11, 8.5), facecolor=DARK_BG)
        ax.axis("off")
        ax.text(0.5, 0.95, "SCHUMANN RESONANCE CONTEXT", fontsize=13,
                fontweight="bold", ha="center", color=NEON_GREEN, transform=ax.transAxes)
        y = 0.88
        for line in textwrap.wrap(sr_notes or "No SR notes supplied.", 95):
            ax.text(0.06, y, line, fontsize=9.2, color=WHITE, transform=ax.transAxes)
            y -= 0.028
        if sr_image_path and os.path.exists(sr_image_path):
            try:
                img = plt.imread(sr_image_path)
                ax_img = fig.add_axes([0.08, 0.08, 0.84, 0.42])
                ax_img.imshow(img)
                ax_img.axis("off")
            except Exception:
                pass
        pdf.savefig(fig, bbox_inches="tight", facecolor=DARK_BG)
        plt.close(fig)

        # Page 7 – Provenance
        fig, ax = plt.subplots(figsize=(11, 8.5), facecolor=DARK_BG)
```

```
        ax.axis("off")
        ax.text(0.5, 0.95, "DATA PROVENANCE & SCIENCE SCOPE", fontsize=13,
                fontweight="bold", ha="center", color=NEON_GREEN, transform=ax.transAxes)
        y = 0.86
        sources = [
            "• Kp: GFZ Potsdam official JSON (definitive when available)",
            "• X-ray / electrons / solar wind: NOAA SWPC live snapshot",
            "• Weather: Hybrid — automatic attempt + user-supplied values (quality flagged)",
            "• Schumann: User-supplied spectrograms / notes",
            "• Orographic lift: calculated from wind direction vs zone ridge angle",
            "• Science scope: mainstream geomagnetic + meteorological + user SR visual",
            "• No speculative plasmoid, Cartan, or non-mainstream indices in core engine",
        ]
        for s in sources:
            ax.text(0.08, y, s, fontsize=9.2, color=WHITE, transform=ax.transAxes)
            y -= 0.035
        pdf.savefig(fig, bbox_inches="tight", facecolor=DARK_BG)
        plt.close(fig)

    print(f"[SUCCESS] Full Report → {output_path}")

# ---------------------------------------------------------------------------
# Main engine
# ---------------------------------------------------------------------------
class SMAF:
    def __init__(self):
        self.version = VERSION
        self.data = DataLayer()

    def run(self, start: str, end: str, zone_key: str = "Tucson",
            sr_image: Optional[str] = None, sr_notes: str = "",
            user_weather: Optional[Dict] = None) -> Dict:
        """
        user_weather example:
        {
          "2025-05-17": {
            "wind_dir": "SW", "wind_speed": "12 mph",
            "pressure": "1012 hPa", "temp": "78 F",
            "humidity": "25%", "precip": "0", "notes": "dry clear"
          },
          ...
        }
        """
        zone = ZONES.get(zone_key, ZONES["Tucson"])
        gfz = self.data.fetch_gfz_kp(start, end)
        noaa = self.data.get_noaa_snapshot()

        # Hybrid weather
        auto = self.data.attempt_historical_weather(zone.get("station", ""), start, end)
        weather_block = {
            "quality": "unavailable",
            "source": auto.get("source", "none"),
            "note": auto.get("note", ""),
            "days": {}
        }
        if user_weather:
            weather_block["quality"] = "user-supplied"
            weather_block["source"] = "user-supplied dictionary"
            weather_block["note"] = "Values provided by user (same workflow as SR spectrograms)."
            for day, w in user_weather.items():
                lift = orographic_lift(zone["ridge"], w.get("wind_dir", "S"))
                weather_block["days"][day] = {**w, "orographic_lift": lift}
```

```
        elif auto.get("ok"):
            weather_block = auto

        results = []
        if gfz.get("ok"):
            for day in sorted(gfz["days"].keys()):
                info = gfz["days"][day]
                kp = info["kp_max"]
                geo = geoeffectiveness(kp, noaa["bz"], noaa["vsw"])
                st = storm_type(kp, noaa["vsw"], noaa["bz"], noaa["xray_max"])
                results.append({
                    "date": day,
                    "kp_max": kp,
                    "kp_3hr": info["kp_3hr"],
                    "quality": info["quality"],
                    "geo": geo,
                    "storm_type": st,
                })
        lag = lag_metric([r["kp_max"] for r in results]) if results else {"max_consecutive": 0, "threshold": 4.5}

        stamp = datetime.utcnow().strftime("%Y%m%d_%H%M")
        pdf_path = os.path.join(OUTPUT_DIR, f"SMAF_v3.7_{zone_key}_{start}_to_{end}_{stamp}.pdf")
        if results:
            create_full_report(results, zone["name"], start, end, noaa, lag,
                               sr_notes, weather_block, pdf_path, sr_image)

        return {
            "results": results,
            "lag": lag,
            "noaa": noaa,
            "weather": weather_block,
            "pdf": pdf_path if results else None
        }


if __name__ == "__main__":
    print(f"SMAF v{VERSION} — Hybrid Historical Weather Edition")
    print("Ready. Import SMAF and call .run() with optional user_weather dict.")
    print()
    print("Recommended manual historical weather sources (when automatic fetch fails):")
    print("  1. NOAA ISD / ISD Lite: https://www.ncei.noaa.gov/products/land-based-station/integrated-surface-
database")
    print("  2. NOAA Climate Data Online (CDO): https://www.ncdc.noaa.gov/cdo-web/")
    print("  3. Iowa State ASOS download: https://mesonet.agron.iastate.edu/request/download.phtml")
    print("  4. MesoWest / Synoptic: https://mesowest.utah.edu/")
    print("  5. Local METAR archives via OGIMET or similar")
```

END SMAF v3.7 SOURCE

2. StormMode Forecaster v9.0 Public

Version 9.0 Public (26 July 2026). StormMode answers the question: “Do we expect coherent plasmoids, and how long should they last?” It produces the Enhanced Natural Plasmoid Index (eNPI), Persistence Index, hybrid lag (day-2 peak + 3–5 day helicity-protected tail), mode probabilities, and the geometric weight fraction from Beltrami coherence + Znidarsic match proxies (historically 28–33 % of eNPI). Formal gate threshold eNPI > 1.25; operational forecasting threshold ≈ 1.30. Location profiles include Tucson (high orographic), Barksdale (lower orographic), and the new Virginia East Coast site.

Public Version notes: All Cosmic Lattice Web (CLW) content and ESCF projection methods removed (unpublished). Retains only publicly documented UCRC corpus elements: AetherLink Part II UCRC-CG eNPI / Persistence / lag, Beltrami + Znidarsic geometric proxies, Mild Undead Geometry weft protection, Plasma/Anti-Plasma Dialect dual-web, Operator layer (geo + eNPI only), LocationProfile registry, and nucleator-type biases.

Full Source Code — StormMode_Forecaster_v9.0.py

BEGIN STORMMODE v9.0 PUBLIC SOURCE

```
#!/usr/bin/env python3
"""
StormMode Forecaster v9.0 (Public Version)
==========================================
Builds on v8.4 (Expanded Nucleator Types + calibrated probabilities + Bz/Solar Wind Geoeffectiveness).


Public Version notes (2026-07-26):
- All Cosmic Lattice Web (CLW) content and ESCF projection methods removed (unpublished).
- Retains only publicly documented UCRC corpus elements.


Major public additions in v9.0:
- AetherLink Part II UCRC-CG insights: eNPI (Enhanced Natural Plasmoid Index),
  orographic multipliers, Persistence Index, hybrid lag metric (day-2 peak + 3–5 day tail),
  BeltramiCoherence + ZnidarsicMatch geometric proxies (28–33% weight contribution class).
- Location-aware forecasting with LocationProfile registry:
    * Tucson_AZ (high orographic Santa Catalina baseline)
    * Barksdale_LA (lower orographic mid-latitude inland)
    * Virginia_EastCoast (NEW — Appalachian/Blue Ridge moderate orographic + coastal humidity bias)
- Mild Undead Geometry (public) + Plasma/Anti-Plasma Dialect dual-web weft-protection
  & persistence multipliers for mode longevity.
- Relative κ0-reference and Znidarsic vt scaling anchors (UCRC-CG Layer-1 classical geometry).
- All prior public features retained and enhanced (nucleator type biases Mode_A/B/C/D + probabilities,
  3-day+ Kp support, geoeffectiveness, Operator/Screwdriver logic, neon output, comparison demos).


Nucleator Types supported (unchanged from v8.4):
- AGI (Silver Iodide seeding)
- NACL (Natural/electrolyte)
- PM_HYBRID (Pollution + dust/smoke/pollen)
- METEORITE_DUST (medium detail, high Mode_C under geoeffective)
- NATURAL_MINERAL_DUST


Version: 9.0 Public | Date: 2026-07-26
Anchored exclusively in public papers: AetherLink Part II UCRC-CG (Boswell & Wulf),
UCRC-CG v3.0, APR/RAST v.4, Plasma/Anti-Plasma Dialect v3.0, Phoenician Aleph A,
Undead Geometry v1.2, RTI-TR Resonant Torsion / GED / Hodge-Dual Bianchi series.
"""


import numpy as np
import requests
from dataclasses import dataclass, field
from enum import Enum
from datetime import datetime
from typing import Dict, Optional, List, Tuple
```

```
import warnings
warnings.filterwarnings("ignore")

# Neon styling (preserved)
CYAN = '\033[96m'
MAGENTA = '\033[95m'
GREEN = '\033[92m'
YELLOW = '\033[93m'
BLUE = '\033[94m'
RED = '\033[91m'
WHITE = '\033[97m'
RESET = '\033[0m'
BOLD = '\033[1m'

def neon_print(text: str, color: str = CYAN, bold: bool = False):
    prefix = BOLD if bold else ""
    print(f"{prefix}{color}{text}{RESET}")

def neon_header(title: str):
    print(f"\n{BOLD}{MAGENTA}{'='*90}{RESET}")
    print(f"{BOLD}{CYAN}{title.center(90)}{RESET}")
    print(f"{BOLD}{MAGENTA}{'='*90}{RESET}\n")

def neon_success(text: str):
    print(f"{GREEN}✅ {text}{RESET}")

def neon_info(text: str):
    print(f"{CYAN}ℹ️  {text}{RESET}")

def neon_warn(text: str):
    print(f"{YELLOW}⚠️  {text}{RESET}")

# ================================================================================
# SPACE WEATHER + GEOEFFECTIVENESS (v8.2+ preserved, used by eNPI)
# ================================================================================
def get_solar_wind_bz_data() -> Dict:
    try:
        url = "https://services.swpc.noaa.gov/products/solar-wind/mag-7-day.json"
        data = requests.get(url, timeout=7).json()
        latest = data[-1]
        return {"bz": round(float(latest[3]), 1), "solar_wind_speed": round(float(latest[2]), 0), "status":
"LIVE"}
    except:
        return {"bz": -3.2, "solar_wind_speed": 420, "status": "FALLBACK"}

def calculate_geoeffectiveness_index(kp: float, bz: float, solar_wind_speed: float) -> Dict:
    kp_score = min(kp / 9.0, 1.0) * 50
    bz_score = 45 if bz < -10 else (30 if bz < -5 else (15 if bz < 0 else max(0, 10 - bz * 2)))
    speed_score = min((solar_wind_speed - 300) / 400, 1.0) * 20
    total = kp_score + bz_score + speed_score
    geoeffectiveness = min(total, 100)
    level = "VERY HIGH" if geoeffectiveness >= 75 else ("HIGH" if geoeffectiveness >= 60 else ("MODERATE" if
geoeffectiveness >= 45 else "LOW"))
    return {"geoeffectiveness_index": round(geoeffectiveness, 1), "level": level, "bz": bz, "solar_wind_speed":
solar_wind_speed}

# ================================================================================
# LOCATION PROFILES (NEW in v9.0 — AetherLink-derived + Virginia East Coast)
# ================================================================================
class LocationID(Enum):
```

```
    TUCSON_AZ = "Tucson_AZ (Santa Catalina Foothills)"
    BARKSDALE_LA = "Barksdale_LA (Northern Louisiana)"
    VIRGINIA_EASTCOAST = "Virginia_EastCoast (Appalachian / Blue Ridge + Coastal)"


@dataclass
class LocationProfile:
    location_id: LocationID
    orographic_factor: float          # Multiplier on intensity & Persistence (AetherLink)
    humidity_bias: float = 1.0        # Affects NACL / PM_HYBRID efficiency
    coastal_factor: float = 1.0       # Mild aerosol / ionosphere influence
    sr_mode_selectivity_base: float = 0.35
    leakage_index_base: float = 0.16
    base_pm_index: float = 65.0
    latitude_note: str = ""
    description: str = ""


LOCATION_REGISTRY: Dict[LocationID, LocationProfile] = {
    LocationID.TUCSON_AZ: LocationProfile(
        location_id=LocationID.TUCSON_AZ,
        orographic_factor=1.45,          # Mid of observed 1.35–1.55
        humidity_bias=0.92,              # Arid
        coastal_factor=0.95,
        sr_mode_selectivity_base=0.38,
        leakage_index_base=0.15,
        base_pm_index=70.0,
        latitude_note="~32.2°N mid-latitude high-orographic",
        description="High orographic uplift (Santa Catalina), coherence-dominated resonance flavor. Peak eNPI
historically >2.05, Persistence ~3.4"
    ),
    LocationID.BARKSDALE_LA: LocationProfile(
        location_id=LocationID.BARKSDALE_LA,
        orographic_factor=1.22,          # Mid of 1.15–1.30
        humidity_bias=1.08,
        coastal_factor=1.02,
        sr_mode_selectivity_base=0.34,
        leakage_index_base=0.18,
        base_pm_index=62.0,
        latitude_note="~32.5°N mid-latitude lower-orographic inland",
        description="Lower orographic, moderate leakage pathway. Peak eNPI ~1.53, Persistence ~2.8. Validates lag
metric robustness."
    ),
    LocationID.VIRGINIA_EASTCOAST: LocationProfile(
        location_id=LocationID.VIRGINIA_EASTCOAST,
        orographic_factor=1.32,          # Appalachian/Blue Ridge moderate + valley effects
        humidity_bias=1.12,              # Mid-Atlantic / coastal moisture
        coastal_factor=1.10,
        sr_mode_selectivity_base=0.36,
        leakage_index_base=0.17,
        base_pm_index=68.0,
        latitude_note="~37–39°N mid-Atlantic, Blue Ridge / Appalachian orographic + coastal plain",
        description="NEW v9.0 East Coast site. Moderate orographic (Appalachians/Blue Ridge uplift + rain-shadow
valleys), elevated humidity favoring NACL/PM_HYBRID, coastal aerosol influence. Topology-induced waves possible.
Balanced SR flavor."
    ),
}

def get_location_profile(loc: LocationID) -> LocationProfile:
    return LOCATION_REGISTRY.get(loc, LOCATION_REGISTRY[LocationID.TUCSON_AZ])


# ================================================================================
# NUCLEATOR SYSTEM (v8.4 preserved + mild location humidity interaction)
# ================================================================================
```

```
class NucleatorType(Enum):
    AGI = "AgI (Silver Iodide)"
    NACL = "NaCl (Natural/Electrolyte)"
    PM_HYBRID = "PM-Hybrid (Pollution + Dust/Smoke/Pollen)"
    METEORITE_DUST = "Meteorite Dust"
    NATURAL_MINERAL_DUST = "Natural Mineral Dust"


@dataclass
class NucleatorParams:
    nucleator_type: NucleatorType
    concentration: float = 1.0
    efficiency: float = 1.0
    size_factor: float = 1.0
    bz_sensitivity: float = 1.0
    speed_sensitivity: float = 1.0
    humidity_response: float = 1.0   # NEW mild v9 interaction


def get_nucleator_params(nucleator_type: NucleatorType, concentration: float = 1.0,
                         humidity_bias: float = 1.0) -> NucleatorParams:
    """Returns medium-detail parameters for each nucleator type (v8.4 + v9 humidity)"""
    if nucleator_type == NucleatorType.AGI:
        return NucleatorParams(nucleator_type, concentration, efficiency=1.0, size_factor=1.0,
                               bz_sensitivity=1.0, speed_sensitivity=1.0, humidity_response=1.0)
    elif nucleator_type == NucleatorType.NACL:
        # Humidity boost for coastal / East Coast
        return NucleatorParams(nucleator_type, concentration, efficiency=0.85 * (0.95 + 0.05 * humidity_bias),
                               size_factor=1.1, bz_sensitivity=0.9, speed_sensitivity=1.05,
                               humidity_response=1.15)
    elif nucleator_type == NucleatorType.PM_HYBRID:
        return NucleatorParams(nucleator_type, concentration, efficiency=0.75 * (0.97 + 0.03 * humidity_bias),
                               size_factor=0.95, bz_sensitivity=1.15, speed_sensitivity=1.1,
                               humidity_response=1.10)
    elif nucleator_type == NucleatorType.METEORITE_DUST:
        return NucleatorParams(nucleator_type, concentration, efficiency=0.92, size_factor=1.25,
                               bz_sensitivity=1.35, speed_sensitivity=1.25, humidity_response=0.98)
    elif nucleator_type == NucleatorType.NATURAL_MINERAL_DUST:
        return NucleatorParams(nucleator_type, concentration, efficiency=0.78, size_factor=0.9,
                               bz_sensitivity=1.1, speed_sensitivity=1.15, humidity_response=1.05)
    else:
        return NucleatorParams(nucleator_type, concentration, efficiency=1.0, size_factor=1.0,
                               bz_sensitivity=1.0, speed_sensitivity=1.0, humidity_response=1.0)

# ============================================================================
# CORE MODULES (Updated for Nucleator + Location + Beltrami/Znidarsic awareness)
# ============================================================================
class RASTv4Core:
    def emergence_equation_v10(self, sr_intensity: float, agi_conc: float, geomagnetic_index: float,
                               conscious_torque: float = 0.0) -> float:
        base = 1.0
        sr_term = 0.8 * np.tanh(sr_intensity / 50.0)
        agi_term = 0.6 * np.sqrt(max(agi_conc, 0.01))
        geo_term = 0.45 * np.log1p(max(geomagnetic_index, 0.1))
        torque_term = 0.3 * conscious_torque
        return float(np.clip(base + sr_term + agi_term + geo_term + torque_term, 0.0, 5.5))

    def predict_mode_probabilities(self, emergence: float, screw_stability: float, crystal_factor: float,
                                   sr_spike: float, forecasted_kp: float = 4.0, bz: float = -3.0,
                                   solar_wind_speed: float = 420, nucleator: NucleatorParams = None,
                                   orographic: float = 1.0, beltrami_coherence: float = 1.0,
                                   weft_protection: float = 1.0) -> Dict[str, float]:
        """v9: Nucleator + location oro + Beltrami coherence + Undead weft influence mode probs"""
```

```
        base_a = max(0.05, 1.0 - 0.38 * emergence)  # soft floor to retain residual Mode_A even under high
emergence
        base_b = max(0.0, 0.33 * emergence * screw_stability)
        base_c = max(0.0, 0.23 * (emergence - 1.35) * (1.0 - screw_stability))
        base_d = max(0.0, 0.15 * crystal_factor * (sr_spike / 30.0))

        # Kp and Bz/speed effects (from v8.2)
        if forecasted_kp >= 6.5:
            base_c *= 1.38
            base_b *= 0.84
        elif forecasted_kp >= 5.5:
            base_c *= 1.18

        if bz < -8 and solar_wind_speed > 550:
            base_c *= 1.52
            base_a *= 0.76
        elif bz < -5 and solar_wind_speed > 480:
            base_c *= 1.26

        # v8.4 Nucleator-specific mode bias
        if nucleator:
            if nucleator.nucleator_type == NucleatorType.METEORITE_DUST:
                base_c *= (1.0 + 0.25 * nucleator.bz_sensitivity)
                base_b *= 0.92
            elif nucleator.nucleator_type == NucleatorType.PM_HYBRID:
                base_c *= 1.12
            elif nucleator.nucleator_type == NucleatorType.NATURAL_MINERAL_DUST:
                base_b *= 1.08

        # NEW v9: Orographic boosts convective (B) and unstable (C) under high oro
        if orographic > 1.30:
            base_b *= (1.0 + 0.12 * (orographic - 1.0))
            base_c *= (1.0 + 0.08 * (orographic - 1.0))
        elif orographic < 1.20:
            base_a *= 1.05  # Slightly more stable under low oro

        # NEW v9: Beltrami coherence favors structured / persistent modes
        base_c *= (0.92 + 0.08 * beltrami_coherence)
        base_d *= (0.95 + 0.10 * beltrami_coherence)

        # NEW v9: Undead / Weft protection extends Mode_A and Mode_D longevity signature
        base_a *= (0.96 + 0.06 * weft_protection)
        base_d *= (0.97 + 0.08 * weft_protection)

        total = base_a + base_b + base_c + base_d + 1e-6
        return {
            'Mode_A_Stable': float(base_a / total),
            'Mode_B_Convective': float(base_b / total),
            'Mode_C_Unstable': float(base_c / total),
            'Mode_D_4.5D_Differentiated': float(base_d / total)
        }

class PlasmaDialectEngine:
    """Public Plasma/Anti-Plasma Dialect v3.0 dual-web resonance (published)."""
    def dual_web_resonance(self, plasma_density: float, anti_plasma_density: float) -> Dict:
        net = (plasma_density - anti_plasma_density) / (plasma_density + anti_plasma_density + 0.01)
        stability = max(0.1, 1.0 - 0.3 * abs(net))
        return {'net_coupling': float(net), 'web_stability': float(stability)}

class OperatorLayer:
```

```
    def somatic_sr_transduction(self, ringing_level: float) -> Dict:
        sr_proxy = 7.83 + (ringing_level - 5.0) * 0.8
        return {'sr_proxy_hz': float(sr_proxy), 'spike_detected': ringing_level > 7.5,
                'transduction_confidence': min(0.95, 0.6 + 0.08 * ringing_level)}

    def screwdriver_protocol(self, current_tension: float, modal_preference: str,
                             forecasted_kp: float = 4.0, bz: float = -3.0,
                             solar_wind_speed: float = 420,
                             eNPI: float = 1.0) -> Dict:
        """Public Operator layer: tension recommendation driven by geoeffectiveness + eNPI only.
        (CLW/ESCF dependencies removed for Public Version.)"""
        recommended = current_tension
        action = "Maintain current tension"
        if bz < -8 and solar_wind_speed > 550 and forecasted_kp >= 6.0:
            if eNPI > 1.8:
                recommended *= 0.88
                action = "REDUCE tension - Very high geoeffective + elevated eNPI"
            else:
                recommended *= 1.10
                action = "INCREASE tension - Strong geoeffective storm"
        elif forecasted_kp >= 7.0:
            recommended *= 1.05 if eNPI < 1.6 else 0.92
            action = "ADJUST tension for high-Kp window"
        elif eNPI >= 1.30 and forecasted_kp >= 5.0:
            recommended *= 1.06
            action = "Mild INCREASE - threshold eNPI under moderate geo"
        return {'recommended_tension': float(np.clip(recommended, 0.5, 2.25)),
                'recommended_action': action}

class AFI_GovLayer:
    def four_force_screw_v71(self, solar_key_hz: float = 432.0, plasmoid_type: str = "AGI",
                             pm_factor: float = 1.0, forecasted_kp: float = 4.0, bz: float = -3.0,
                             solar_wind_speed: float = 420) -> str:
        base = ""
        if solar_key_hz == 432:
            base = "Yukawa lattice lock ACTIVE"
        if "PM" in plasmoid_type or "Meteorite" in plasmoid_type:
            base += f" | {plasmoid_type}"
        geo = calculate_geoeffectiveness_index(forecasted_kp, bz, solar_wind_speed)
        if geo['level'] in ["HIGH", "VERY HIGH"]:
            base += f" | {geo['level']} GEOEFFECTIVENESS"
        return base

# ============================================================================
# v9 GEOMETRIC PROXIES (AetherLink / UCRC-CG classical anchors)
# ============================================================================
def compute_beltrami_coherence(emergence: float, screw_stability: float, geo_index: float,
                               orographic: float) -> float:
    """Proxy for Beltrami eigenmode coherence / axial torsion integrability.
    Higher when emergence, stability, and geo align under orographic support.
    Classical anchor: exact integrability of first Bianchi in curvature-free axial sector."""
    base = 0.55 + 0.18 * np.tanh(emergence / 2.5) + 0.12 * screw_stability
    geo_term = 0.15 * min(geo_index / 80.0, 1.0)
    oro_term = 0.10 * (orographic - 1.0)
    return float(np.clip(base + geo_term + oro_term, 0.35, 1.25))

def compute_znidarsic_match(bz: float, solar_wind_speed: float, forecasted_kp: float) -> float:
    """Proxy for Znidarsic perfect-transmission condition (Γ ≡ 0 near vt ≈ 1.094e6 m/s).
    Higher under strong southward Bz + elevated solar wind (geoeffective match).
    Classical: vanishing reflection of Beltrami torsion waves at interface."""
    bz_term = 0.45 if bz < -8 else (0.30 if bz < -5 else (0.15 if bz < 0 else 0.05))
    speed_term = min(max((solar_wind_speed - 400) / 300.0, 0.0), 0.35)
```

```
    kp_term = 0.15 if forecasted_kp >= 5.5 else 0.05
    return float(np.clip(0.40 + bz_term + speed_term + kp_term, 0.25, 1.15))


def compute_weft_protection(operator_tension: float, web_stability: float, geo_index: float) -> float:
    """Undead Geometry (public) + Plasma/Anti-Plasma Dialect dual-web persistence / weft integrity factor.
    Supports non-decaying weft protection under adversarial (high-geo) loading.
    Classical: controlled leakage + dual-web regeneration (AetherLink + Dialect)."""
    tension_term = 0.25 * np.clip(operator_tension - 0.8, 0.0, 1.2)
    web_term = 0.35 * web_stability
    geo_resilience = 0.20 * min(geo_index / 70.0, 1.0)
    return float(np.clip(0.70 + tension_term + web_term + geo_resilience, 0.55, 1.35))


# ============================================================================
# eNPI + PERSISTENCE (AetherLink Part II core)
# ============================================================================
def compute_eNPI(sm_proxy: float, li_proxy: float, psr_norm: float, geo_index: float,
                 nucleator: NucleatorParams, orographic: float,
                 beltrami_coherence: float, znidarsic_match: float) -> Dict:
    """Enhanced Natural Plasmoid Index (AetherLink v8 schematic + geometric layer).
    eNPI ≈ 0.50 Sm + 0.40 Li + 0.30 PSR + geometric proxies (Beltrami + Znidarsic ~28-33% class).
    Threshold ~1.30 for emergence / threshold crossing."""
    core = 0.50 * sm_proxy + 0.40 * li_proxy + 0.30 * psr_norm
    # Geometric contribution (target ~0.28–0.33 of total at peak per AetherLink)
    geometric = 0.145 * beltrami_coherence + 0.115 * znidarsic_match
    # Nucleator + orographic multipliers
    nuc_boost = nucleator.efficiency * nucleator.size_factor * nucleator.concentration *
nucleator.humidity_response
    oro_boost = orographic
    geo_boost = 1.0 + 0.08 * (geo_index / 50.0)

    raw = (core + geometric) * nuc_boost * oro_boost * geo_boost
    eNPI = float(np.clip(raw * 1.45, 0.5, 2.85))  # Scale to observed historical range (Tucson >2, Barksdale ~1.5)

    geometric_weight = geometric / (core + geometric + 1e-6)
    above_threshold = eNPI >= 1.30

    return {
        "eNPI": round(eNPI, 3),
        "above_threshold": above_threshold,
        "geometric_weight_frac": round(geometric_weight, 3),
        "core_drivers": round(core, 3),
        "geometric_proxy": round(geometric, 3),
        "orographic_applied": round(orographic, 3)
    }

def estimate_persistence_and_lag(eNPI: float, orographic: float, beltrami: float,
                                 znidarsic: float, days_forecast: int = 5) -> Dict:
    """Persistence Index proxy + Hybrid Lag estimate (day-2 peak + 3–5 day tail).
    From AetherLink: Persistence = consecutive_days_above * mean_oro.
    Lag minimum ~1.2–1.4 days after joint eNPI+oro threshold for Beltrami lock-in."""
    if eNPI < 1.30:
        return {
            "persistence_index": 0.0,
            "estimated_tail_days": 0.0,
            "hybrid_lag_days": None,
            "lock_in_probability": 0.0,
            "signature": "Below threshold — no sustained lock-in expected"
        }

    # Approximate consecutive run potential from strength above threshold
    excess = eNPI - 1.30
```

```
    base_run = 2.0 + min(excess * 3.5, 4.0)  # 2–6 day potential
    # Orographic multiplies run length (AetherLink definition)
    persistence_index = round(base_run * orographic, 2)

    # Tail duration (helicity-protected 3–5 day envelope)
    tail = 3.0 + 1.5 * min(beltrami * znidarsic, 1.2)
    tail = float(np.clip(tail, 2.5, 5.5))

    # Hybrid lag (Form C style) — minimum near peak
    lag = 1.3 - 0.15 * (znidarsic - 0.7) + 0.05 * (1.0 - min(excess, 0.8))
    lag = float(np.clip(lag, 1.1, 1.6))

    lock_prob = min(0.95, 0.45 + 0.25 * excess + 0.15 * (orographic - 1.0) + 0.10 * beltrami)

    return {
        "persistence_index": persistence_index,
        "estimated_tail_days": round(tail, 1),
        "hybrid_lag_days": round(lag, 2),
        "lock_in_probability": round(lock_prob, 2),
        "signature": f"Day-2 peak + {tail:.1f}-day tail expected after lag ~{lag:.2f} d (Beltrami lock-in +
helicity protection)"
    }

# ================================================================================
# v8.3/v8.4 CALIBRATED PROBABILITY (Updated for location + geometric)
# ================================================================================
def _calculate_calibrated_probability(kp: float, pm_factor: float, geo_index: float,
                                      nucleator: NucleatorParams, orographic: float = 1.0,
                                      beltrami: float = 1.0) -> float:
    """v9: Nucleator + location oro + Beltrami-aware calibrated probability"""
    kp_factor = min(0.32 + 0.17 * np.log1p(max(kp, 1.0)), 0.78)
    geo_boost = 1.13 if geo_index >= 75 else (1.07 if geo_index >= 60 else (1.03 if geo_index >= 45 else 1.0))

    nucleator_boost = nucleator.efficiency * nucleator.size_factor * nucleator.concentration *
nucleator.humidity_response
    oro_boost = 1.0 + 0.22 * (orographic - 1.0)
    beltrami_boost = 0.92 + 0.12 * beltrami

    prob = kp_factor * pm_factor * geo_boost * nucleator_boost * oro_boost * beltrami_boost

    if pm_factor > 1.7 and geo_index > 65:
        prob *= 0.91

    return min(prob, 0.91)  # retain headroom vs v8.4 0.91 cap

# ================================================================================
# MAIN v9.0 FORECASTER
# ================================================================================
class StormModeForecaster_v9:
    def __init__(self):
        self.version = "9.0 Public Version — UCRC Corpus Integration (AetherLink / UCRC-CG / APR-RAST / Plasma
Dialect / Undead Geometry)"
        self.rast = RASTv4Core()
        self.plasma = PlasmaDialectEngine()
        self.operator = OperatorLayer()
        self.afi = AFI_GovLayer()
        # UCRC-CG Layer-1 reference anchors (relative only; public)
        self.kappa0_ref = 1.37e-17  # m^-1 intermediate Beltrami eigenvalue (canonical dual-vortex)
        self.vt_znidarsic = 1.094e6  # m/s vanishing-reflection reference
```

```
    def run_forecast_v9(self,
                        location: LocationID = LocationID.TUCSON_AZ,
                        nucleator_type: NucleatorType = NucleatorType.AGI,
                        nucleator_concentration: float = 1.0,
                        geomagnetic_index: float = 4.5,
                        forecasted_kp: float = 4.5,
                        bz: float = -3.5,
                        solar_wind_speed: float = 430,
                        pm_index: Optional[float] = None,
                        operator_tension: float = 1.12,
                        sr_intensity: float = 42.0,
                        location_note: str = "") -> Dict:

        loc_prof = get_location_profile(location)
        if pm_index is None:
            pm_index = loc_prof.base_pm_index

        nucleator = get_nucleator_params(nucleator_type, nucleator_concentration, loc_prof.humidity_bias)
        conscious_torque = (operator_tension - 1.0) * 0.8

        # Core emergence (preserved from v8.4)
        emergence = self.rast.emergence_equation_v10(sr_intensity, 0.32 * nucleator.concentration,
                                                     geomagnetic_index, conscious_torque)

        # Geometric proxies (AetherLink / UCRC-CG public)
        beltrami = compute_beltrami_coherence(emergence, 0.72, geomagnetic_index * 10, loc_prof.orographic_factor)
        znidarsic = compute_znidarsic_match(bz, solar_wind_speed, forecasted_kp)

        # Public Plasma/Anti-Plasma Dialect dual-web
        web = self.plasma.dual_web_resonance(emergence * 0.8, max(0.1, 2.0 - emergence))
        weft = compute_weft_protection(operator_tension, web['web_stability'],
                                       calculate_geoeffectiveness_index(forecasted_kp, bz,
solar_wind_speed)['geoeffectiveness_index'])

        # Mode probabilities with public v9 influences (nucleator + oro + Beltrami + Undead weft)
        mode_probs = self.rast.predict_mode_probabilities(
            emergence, 0.72, 1.0, 38.0,
            forecasted_kp=forecasted_kp, bz=bz, solar_wind_speed=solar_wind_speed,
            nucleator=nucleator,
            orographic=loc_prof.orographic_factor,
            beltrami_coherence=beltrami,
            weft_protection=weft
        )

        geo = calculate_geoeffectiveness_index(forecasted_kp, bz, solar_wind_speed)
        pm_factor = 1 + (pm_index / 100) * 1.42 if pm_index > 50 else 1.0

        # Calibrated probs (location + geometric aware)
        agi_prob = _calculate_calibrated_probability(forecasted_kp, pm_factor, geo['geoeffectiveness_index'],
                                                     nucleator, loc_prof.orographic_factor, beltrami)
        nacl_prob = _calculate_calibrated_probability(forecasted_kp, pm_factor * 0.92,
geo['geoeffectiveness_index'],
                                                      nucleator, loc_prof.orographic_factor, beltrami)

        # SR proxies from location baseline + geo modulation
        sm_proxy = loc_prof.sr_mode_selectivity_base * (1.0 + 0.15 * min(geo['geoeffectiveness_index'] / 70, 1.0))
        li_proxy = loc_prof.leakage_index_base * (1.0 + 0.10 * (1.0 if bz > -4 else 0.7))
        psr_norm = min(sr_intensity / 55.0, 1.15)
```

```
        # eNPI (AetherLink public)
        enpi_res = compute_eNPI(sm_proxy, li_proxy, psr_norm, geo['geoeffectiveness_index'],
                                nucleator, loc_prof.orographic_factor, beltrami, znidarsic)

        # Persistence + lag (AetherLink public)
        persist = estimate_persistence_and_lag(enpi_res['eNPI'], loc_prof.orographic_factor,
                                               beltrami, znidarsic)

        dominant_mode = max(mode_probs, key=mode_probs.get)
        screw_str = self.afi.four_force_screw_v71(432.0, nucleator.nucleator_type.value, pm_factor,
                                                  forecasted_kp, bz, solar_wind_speed)
        # Public Operator screwdriver (geo + eNPI only; CLW/ESCF removed)
        screw_proto = self.operator.screwdriver_protocol(
            operator_tension, "Statera",
            forecasted_kp=forecasted_kp, bz=bz, solar_wind_speed=solar_wind_speed,
            eNPI=enpi_res['eNPI']
        )

        note = location_note if location_note else loc_prof.description

        return {
            'version': self.version,
            'location': loc_prof.location_id.value,
            'location_note': note,
            'orographic_factor': loc_prof.orographic_factor,
            'humidity_bias': loc_prof.humidity_bias,
            'nucleator': nucleator.nucleator_type.value,
            'nucleator_concentration': nucleator_concentration,
            'forecasted_kp': forecasted_kp,
            'bz': bz,
            'solar_wind_speed': solar_wind_speed,
            'geoeffectiveness': geo,
            'pm_factor': round(pm_factor, 2),
            'beltrami_coherence': round(beltrami, 3),
            'znidarsic_match': round(znidarsic, 3),
            'weft_protection': round(weft, 3),
            'eNPI': enpi_res,
            'persistence': persist,
            'dominant_mode': dominant_mode,
            'agi_probability': round(agi_prob * 100, 1),
            'nacl_probability': round(nacl_prob * 100, 1),
            'mode_probabilities': {k: round(v, 3) for k, v in mode_probs.items()},
            'four_force_screw': screw_str,
            'screwdriver_action': screw_proto['recommended_action'],
            'recommended_tension': round(screw_proto['recommended_tension'], 2),
            'kappa0_ref_m-1': self.kappa0_ref,
            'vt_znidarsic_m_s': self.vt_znidarsic
        }

# ============================================================================
# DEMO: Nucleator + Location Comparison (v9)
# ============================================================================
def run_v9_comparison_demo():
    neon_header("🌌 STORMMODE v9.0 PUBLIC VERSION — UCRC CORPUS + VIRGINIA EAST COAST 🌌")
    neon_info("Public Version: Cosmic Lattice Web / ESCF content fully removed (unpublished).")
    neon_info("Retains AetherLink eNPI / Persistence / Lag, BeltramiCoherence + ZnidarsicMatch,")
    neon_info("Undead Geometry weft (public), Location profiles including Virginia_EastCoast.")
    print()

    forecaster = StormModeForecaster_v9()
```

```
    sw = get_solar_wind_bz_data()
    neon_info(f"Current Bz: {sw['bz']} nT | Solar Wind Speed: {sw['solar_wind_speed']} km/s | Status:
{sw['status']}")

    # High geoeffective scenario (as in v8.4 demo)
    high_kp, high_bz, high_speed = 6.2, -8.5, 560
    high_geo = calculate_geoeffectiveness_index(high_kp, high_bz, high_speed)
    neon_info(f"Demo scenario: Kp={high_kp} | Bz={high_bz} | Vsw={high_speed} → Geoeffectiveness
{high_geo['level']} ({high_geo['geoeffectiveness_index']})")
    print()

    # --- Nucleator comparison under Tucson (high oro) ---
    neon_header("NUCLEATOR TYPE COMPARISON (Tucson high-orographic baseline)")
    print(f"{'Nucleator Type':<28} | {'AgI%':>6} | {'NaCl%':>6} | {'Mode_C':>7} | {'eNPI':>6} | {'Persist':>7} |
Action")
    print("-" * 105)

    for nuc_type in NucleatorType:
        res = forecaster.run_forecast_v9(
            location=LocationID.TUCSON_AZ,
            nucleator_type=nuc_type,
            nucleator_concentration=1.0,
            geomagnetic_index=6.2,
            forecasted_kp=high_kp,
            bz=high_bz,
            solar_wind_speed=high_speed,
            pm_index=70
        )
        mode_c = res['mode_probabilities']['Mode_C_Unstable']
        enpi = res['eNPI']['eNPI']
        persist = res['persistence']['persistence_index']
        print(f"{nuc_type.value:<28} | {res['agi_probability']:>6.1f} | {res['nacl_probability']:>6.1f} | "
              f"{mode_c:>7.3f} | {enpi:>6.2f} | {persist:>7.2f} | {res['screwdriver_action'][:35]}")

    print()
    neon_success("Meteorite Dust retains strongest Mode_C bias under high geoeffectiveness (v8.4 retained +
enhanced).")

    # --- Multi-location comparison (AGI baseline) ---
    neon_header("LOCATION COMPARISON (AGI nucleator, same high-geo drivers)")
    print(f"{'Location':<42} | {'Oro':>5} | {'eNPI':>6} | {'Persist':>7} | {'Lag(d)':>6} | {'Beltrami':>8} |
Dominant Mode")
    print("-" * 115)

    for loc in LocationID:
        res = forecaster.run_forecast_v9(
            location=loc,
            nucleator_type=NucleatorType.AGI,
            nucleator_concentration=1.0,
            geomagnetic_index=6.2,
            forecasted_kp=high_kp,
            bz=high_bz,
            solar_wind_speed=high_speed
        )
        enpi = res['eNPI']['eNPI']
        persist = res['persistence']['persistence_index']
        lag = res['persistence']['hybrid_lag_days']
        bel = res['beltrami_coherence']
        print(f"{res['location']:<42} | {res['orographic_factor']:>5.2f} | {enpi:>6.2f} | {persist:>7.2f} | "
              f"{lag if lag else 0:>6.2f} | {bel:>8.3f} | {res['dominant_mode']}")
```

```
    print()
    neon_success("Virginia_EastCoast sits between Tucson (high) and Barksdale (low) orographic response — as
designed.")
    neon_info("eNPI threshold ~1.30, Persistence scales with oro, lag ~1.2–1.4 d, geometric proxies active.")
    neon_info("Public anchors only: Cartan/Beltrami (UCRC-CG), Znidarsic, AetherLink eNPI, APR/RAST modes,")
    neon_info("Plasma/Anti-Plasma Dialect dual-web, Undead Geometry weft (public), Operator layer.")


    # Detailed Virginia snapshot
    neon_header("VIRGINIA EAST COAST DETAILED SNAPSHOT (v9 new location)")
    res_va = forecaster.run_forecast_v9(
        location=LocationID.VIRGINIA_EASTCOAST,
        nucleator_type=NucleatorType.PM_HYBRID,  # humidity-favored
        nucleator_concentration=1.1,
        geomagnetic_index=6.2,
        forecasted_kp=high_kp,
        bz=high_bz,
        solar_wind_speed=high_speed,
        operator_tension=1.18
    )
    neon_print(f"Location: {res_va['location']}", CYAN, bold=True)
    neon_print(f"Note: {res_va['location_note']}", WHITE)
    print(f"  Orographic: {res_va['orographic_factor']:.2f} | Humidity bias: {res_va['humidity_bias']:.2f}")
    print(f"  eNPI: {res_va['eNPI']['eNPI']:.3f} (above threshold: {res_va['eNPI']['above_threshold']}) | "
          f"Geometric weight: {res_va['eNPI']['geometric_weight_frac']*100:.1f}%")
    print(f"  Persistence Index: {res_va['persistence']['persistence_index']:.2f} | "
          f"Tail: {res_va['persistence']['estimated_tail_days']:.1f} d | Lag:
{res_va['persistence']['hybrid_lag_days']} d")
    print(f"  BeltramiCoherence: {res_va['beltrami_coherence']:.3f} | ZnidarsicMatch:
{res_va['znidarsic_match']:.3f} | "
          f"WeftProtection: {res_va['weft_protection']:.3f}")
    print(f"  Dominant Mode: {res_va['dominant_mode']}")
    print(f"  Mode probs: A={res_va['mode_probabilities']['Mode_A_Stable']:.3f}  "
          f"B={res_va['mode_probabilities']['Mode_B_Convective']:.3f}  "
          f"C={res_va['mode_probabilities']['Mode_C_Unstable']:.3f}  "
          f"D={res_va['mode_probabilities']['Mode_D_4.5D_Differentiated']:.3f}")
    print(f"  AgI% {res_va['agi_probability']:.1f} | NaCl% {res_va['nacl_probability']:.1f}")
    print(f"  Screwdriver: {res_va['screwdriver_action']} (tension → {res_va['recommended_tension']})")
    print(f"  Four-force: {res_va['four_force_screw']}")
    neon_success("Virginia integration complete. Ready for East Coast SR / orographic monitoring windows.")


# ============================================================================
# ENTRY POINT
# ============================================================================
if __name__ == "__main__":
    run_v9_comparison_demo()
```

END STORMMODE v9.0 PUBLIC SOURCE

Closing Note

The source listings above are the precise public implementations that generated the SMAF lag metrics, geoeffectiveness values, storm classifications, and the StormMode eNPI, Persistence Index, hybrid lag, mode probabilities, and geometric-weight fractions reported throughout AetherLink Part III Version 4.6. They are provided here so that the dual-engine analysis remains fully transparent and reproducible.

Geometry stays pure. Layer-1 only. Real SR spectrograms integrated. Ready for the three diagrams.

— End of Supporting Source Code Companion —