executable script, main.py.

This script orchestrates the entire "Collapse Engine Stack" into a single, functional pipeline. It takes a defined input state, processes it through the manifold folding engine, predicts future collapse dynamics, and logs the complete analysis to a JSON file.

```
# main.py
# Unified Execution Orchestrator for the Collapse Engine Stack

import numpy as np
import json
import datetime
from dataclasses import dataclass, field
from typing import Dict, List, Tuple, Any
import warnings

# Suppress warnings for cleaner output
warnings.filterwarnings('ignore')

# --- 🧩 1. Core Data Structures & Engine Classes ---
# Sourced from: CatastropheManifoldFolderProductionImplementation.txt

@dataclass
class CollapseSignature:
    """Compressed collapse signature with topological persistence"""
    timestamp: datetime.datetime
    morphology: np.ndarray
    invariants: List[float]  # [trace, determinant, spectral_gap]
    influence_radius: float
    folding_strength: float
    [span_0](start_span)domain: str = 'unknown' #[span_0](end_span)
    [span_1](start_span)metadata: Dict[str, Any] =
field(default_factory=dict) #[span_1](end_span)


@dataclass
class ManifoldState:
    """Current state of the folded manifold"""
    curvature_field: np.ndarray
    memory_tensor: np.ndarray
    entanglement_matrix: np.ndarray
    collapse_history: List[CollapseSignature]
    bifurcation_zones: np.ndarray
    intervention_efficacy: np.ndarray


@dataclass
class DomainMetrics:
    """Represents the input metrics from a specific real-world
```

```
domain."""
    institutional_trust: float
    information_integrity: float
    electoral_confidence: float
    alliance_stability: float
    social_cohesion: float
    timestamp: datetime.datetime


class CatastropheManifoldFolder:
    """
    Implements recursive manifold folding where past collapses
    shape the topology of future collapse probability space
    """

    def __init__(self, dimensions: int = 5, memory_depth: int = 20):
        """
        Initialize the catastrophe manifold

        Args:
            [span_2](start_span)dimensions: Number of collapse
dimensions to track[span_2](end_span)
            [span_3](start_span)memory_depth: How many past collapses
to remember[span_3](end_span)
        """
        [span_4](start_span)self.dimensions = dimensions
#[span_4](end_span)
        [span_5](start_span)self.memory_depth = memory_depth
#[span_5](end_span)

        # Initialize manifold components
        [span_6](start_span)self.manifold_memory = []  # Past collapse
tensors[span_6](end_span)
        [span_7](start_span)self.curvature_field = None  # Current
manifold curvature[span_7](end_span)
        [span_8](start_span)self.entanglement_matrix =
np.eye(dimensions)  # Cross-domain coupling[span_8](end_span)
        [span_9](start_span)self.folding_history = []  # Detailed
collapse records[span_9](end_span)

        # Catastrophe detection thresholds
        [span_10](start_span)self.fold_threshold = 0.7
#[span_10](end_span)
        [span_11](start_span)self.cascade_threshold = 0.85
#[span_11](end_span)
        [span_12](start_span)self.intervention_threshold = 0.6
#[span_12](end_span)
```

```
        # Domain mappings for integration
        self.domain_indices = {
            'institutional_trust': 0,
            'information_integrity': 1,
            'electoral_confidence': 2,
            'alliance_stability': 3,
            'social_cohesion': 4
        [span_13](start_span)} #[span_13](end_span)

    def fold_manifold(self, collapse_tensor: np.ndarray,
                     domain: str = 'unknown',
                     coupling_strength: float = 0.15) ->
ManifoldState:
        """
        Each collapse event 'folds' the manifold, creating persistent
        [span_14](start_span)topological features that influence
future dynamics[span_14](end_span)

        Args:
            [span_15](start_span)collapse_tensor: Tensor representing
the collapse event[span_15](end_span)
            [span_16](start_span)domain: Domain of the collapse (for
tracking)[span_16](end_span)
            [span_17](start_span)coupling_strength: Cross-domain
coupling parameter[span_17](end_span)

        Returns:
            [span_18](start_span)Current manifold state after
folding[span_18](end_span)
        """
        # Extract collapse morphology
        [span_19](start_span)eigenvalues, eigenvectors =
np.linalg.eigh(collapse_tensor) #[span_19](end_span)

        # Compute folding operator based on collapse severity
        [span_20](start_span)folding_strength =
np.max(np.abs(eigenvalues)) / (np.trace(np.abs(collapse_tensor)) +
1e-10) #[span_20](end_span)
        [span_21](start_span)fold_operator =
self._construct_fold_operator(eigenvectors, folding_strength)
#[span_21](end_span)

        # Apply recursive compression
        if self.curvature_field is None:
            [span_22](start_span)self.curvature_field = fold_operator
#[span_22](end_span)
        else:
            # Past collapses influence current folding
```

```
            [span_23](start_span)memory_influence =
self._compute_memory_tensor() #[span_23](end_span)
            self.curvature_field = (
                0.7 * fold_operator +
                0.3 * memory_influence @ self.curvature_field
            [span_24](start_span)) #[span_24](end_span)

        # Update entanglement matrix
        self.entanglement_matrix = self._update_entanglement(
            collapse_tensor, self.entanglement_matrix,
coupling_strength
        [span_25](start_span)) #[span_25](end_span)

        # Compress and store collapse signature
        compressed_signature = self._compress_collapse_signature(
            collapse_tensor, self.curvature_field, domain,
folding_strength
        [span_26](start_span)) #[span_26](end_span)

[span_27](start_span)self.manifold_memory.append(compressed_signature)
#[span_27](end_span)

[span_28](start_span)self.folding_history.append(compressed_signature)
#[span_28](end_span)

        # Maintain memory depth
        if len(self.manifold_memory) > self.memory_depth:
            [span_29](start_span)self.manifold_memory.pop(0)
#[span_29](end_span)

        # Compute current manifold state
        state = ManifoldState(
            curvature_field=self.curvature_field.copy(),
            memory_tensor=self._compute_memory_tensor(),
            entanglement_matrix=self.entanglement_matrix.copy(),
            collapse_history=self.folding_history.copy(),
            bifurcation_zones=self._identify_bifurcation_zones(),

intervention_efficacy=self._compute_intervention_efficacy()
        [span_30](start_span)) #[span_30](end_span)

        return state

    def predict_collapse_zones(self, current_state: np.ndarray,
                            horizon: int = 10,
                            trajectories: int = 100) -> Dict[str,
Any]:
        """
```

```
        Identify regions where manifold folding creates
        [span_31](start_span)'gravitational wells' for future
collapses[span_31](end_span)

        Args:
            [span_32](start_span)current_state: Current system state
vector[span_32](end_span)
            [span_33](start_span)horizon: Prediction horizon (time
steps)[span_33](end_span)
            [span_34](start_span)trajectories: Number of Monte Carlo
trajectories[span_34](end_span)

        Returns:
            [span_35](start_span)Dictionary with collapse predictions
and intervention zones[span_35](end_span)
        """
        if self.curvature_field is None:
            return {
                'collapse_probability': np.zeros_like(current_state),
                'intervention_zones': np.zeros_like(current_state),
                'cascade_risk': 0.0
            [span_36](start_span)} #[span_36](end_span)

        [span_37](start_span)projected_state = current_state @
self.curvature_field #[span_37](end_span)
        [span_38](start_span)flow_field =
self._compute_geodesic_flow(projected_state) #[span_38](end_span)
        [span_39](start_span)collapse_map = np.zeros((trajectories,
len(current_state))) #[span_39](end_span)

        for traj in range(trajectories):
            [span_40](start_span)perturbed_state = projected_state +
np.random.normal(0, 0.01, size=projected_state.shape)
#[span_40](end_span)
            for t in range(horizon):
                evolved_state = self._evolve_on_manifold(
                    perturbed_state, flow_field, t
                [span_41](start_span)) #[span_41](end_span)
                [span_42](start_span)collapse_map[traj] +=
self._detect_convergence_zones(evolved_state) #[span_42](end_span)

        [span_43](start_span)collapse_probability =
np.mean(collapse_map, axis=0) #[span_43](end_span)
        [span_44](start_span)historical_weight =
self._compute_historical_proximity(current_state) #[span_44](end_span)
        [span_45](start_span)weighted_prediction =
collapse_probability * historical_weight #[span_45](end_span)
```

```
        intervention_zones = self._identify_intervention_manifolds(
            weighted_prediction, self.curvature_field
        [span_46](start_span)) #[span_46](end_span)

        cascade_risk = self._compute_cascade_risk(
            weighted_prediction, self.entanglement_matrix
        [span_47](start_span)) #[span_47](end_span)

        return {
            'collapse_probability': weighted_prediction,
            'intervention_zones': intervention_zones,
            'cascade_risk': cascade_risk,
            'bifurcation_distance':
self._distance_to_bifurcation(projected_state)
        [span_48](start_span)} #[span_48](end_span)

    def _construct_fold_operator(self, eigenvectors: np.ndarray,
                              strength: float) -> np.ndarray:
        """Build the topological folding operator that warps the
manifold"""
        scaling_matrix = np.diag(
            1 + strength * np.exp(-np.arange(len(eigenvectors)))
        [span_49](start_span)) #[span_49](end_span)
        [span_50](start_span)fold_operator = eigenvectors @
scaling_matrix @ eigenvectors.T #[span_50](end_span)
        nonlinear_term = strength * np.outer(
            eigenvectors[:, 0], eigenvectors[:, 0]
        [span_51](start_span)) ** 2 #[span_51](end_span)

        if len(self.manifold_memory) > 0:
            [span_52](start_span)memory_effect =
np.zeros_like(fold_operator) #[span_52](end_span)
            for past_sig in self.manifold_memory[-5:]:
                [span_53](start_span)memory_effect += 0.1 *
past_sig.morphology #[span_53](end_span)
            [span_54](start_span)fold_operator += memory_effect
#[span_54](end_span)

        return fold_operator + nonlinear_term

    def _update_entanglement(self, collapse_tensor: np.ndarray,
                           current_entanglement: np.ndarray,
                           coupling_strength: float) -> np.ndarray:
        """Update cross-domain entanglement based on collapse
pattern"""
        [span_55](start_span)collapse_correlation =
np.corrcoef(collapse_tensor) #[span_55](end_span)
        [span_56](start_span)momentum = 0.9 #[span_56](end_span)
```

```
        new_entanglement = (
            momentum * current_entanglement +
            (1 - momentum) * coupling_strength * collapse_correlation
        [span_57](start_span)) #[span_57](end_span)
        [span_58](start_span)eigenvals, eigenvecs =
np.linalg.eigh(new_entanglement) #[span_58](end_span)
        [span_59](start_span)eigenvals = np.maximum(eigenvals, 0)
#[span_59](end_span)
        [span_60](start_span)new_entanglement = eigenvecs @
np.diag(eigenvals) @ eigenvecs.T #[span_60](end_span)
        return new_entanglement

    def _compress_collapse_signature(self, tensor: np.ndarray,
                                    curvature: np.ndarray,
                                    domain: str,
                                    folding_strength: float) ->
CollapseSignature:
        """Compress collapse into persistent topological feature"""
        [span_61](start_span)projected = tensor @ curvature
#[span_61](end_span)
        [span_62](start_span)eigenvals = np.linalg.eigvals(projected)
#[span_62](end_span)
        [span_63](start_span)trace_invariant =
np.real(np.trace(projected)) #[span_63](end_span)
        [span_64](start_span)det_invariant =
np.real(np.linalg.det(projected)) #[span_64](end_span)
        [span_65](start_span)spectral_gap = np.real(np.max(eigenvals)
- np.min(eigenvals)) #[span_65](end_span)

        _[span_66](start_span), eigenvecs = np.linalg.eigh(projected)
#[span_66](end_span)
        [span_67](start_span)morphology = eigenvecs[:, -1]
#[span_67](end_span)

        [span_68](start_span)influence_radius =
self._estimate_influence_radius(tensor, curvature)
#[span_68](end_span)

        signature = CollapseSignature(
            timestamp=datetime.datetime.now(),
            morphology=morphology,
            invariants=[trace_invariant, det_invariant, spectral_gap],
            influence_radius=influence_radius,
            folding_strength=folding_strength,
            domain=domain,
            metadata={
                'eigenvalues': eigenvals.tolist(),
                'tensor_norm': np.linalg.norm(tensor)
```

```
            }
        [span_69](start_span)) #[span_69](end_span)
        return signature

    def _compute_memory_tensor(self) -> np.ndarray:
        """Compute memory tensor from past collapses"""
        if len(self.manifold_memory) == 0:
            [span_70](start_span)return np.eye(self.dimensions)
#[span_70](end_span)

        [span_71](start_span)memory_tensor =
np.zeros((self.dimensions, self.dimensions)) #[span_71](end_span)
        for i, sig in enumerate(self.manifold_memory):
            [span_72](start_span)weight = np.exp(-0.1 *
(len(self.manifold_memory) - i)) #[span_72](end_span)
            [span_73](start_span)memory_tensor += weight *
np.outer(sig.morphology, sig.morphology) #[span_73](end_span)

        [span_74](start_span)memory_tensor /= (np.trace(memory_tensor)
+ 1e-10) #[span_74](end_span)
        return memory_tensor

    def _compute_geodesic_flow(self, state: np.ndarray) -> np.ndarray:
        """Compute geodesic flow field on the folded manifold"""
        [span_75](start_span)flow = -np.gradient(self.curvature_field,
axis=0) #[span_75](end_span)
        [span_76](start_span)flow_field = flow @ state[:, np.newaxis]
#[span_76](end_span)
        [span_77](start_span)return flow_field.squeeze()
#[span_77](end_span)

    def _evolve_on_manifold(self, state: np.ndarray,
                            flow_field: np.ndarray,
                            time_step: int) -> np.ndarray:
        """Evolve state along geodesic on folded manifold"""
        [span_78](start_span)dt = 0.1 #[span_78](end_span)
        [span_79](start_span)k1 = dt * flow_field #[span_79](end_span)
        [span_80](start_span)k2 = dt * self._flow_at_state(state + 0.5
* k1) #[span_80](end_span)
        [span_81](start_span)k3 = dt * self._flow_at_state(state + 0.5
* k2) #[span_81](end_span)
        [span_82](start_span)k4 = dt * self._flow_at_state(state + k3)
#[span_82](end_span)
        [span_83](start_span)evolved_state = state + (k1 + 2*k2 + 2*k3
+ k4) / 6 #[span_83](end_span)
        [span_84](start_span)evolved_state =
self._project_to_manifold(evolved_state) #[span_84](end_span)
        return evolved_state
```

```
    def _flow_at_state(self, state: np.ndarray) -> np.ndarray:
        """Compute flow field at a given state"""
        [span_85](start_span)return -self.curvature_field @ state
#[span_85](end_span)

    def _project_to_manifold(self, state: np.ndarray) -> np.ndarray:
        """Project state back onto the constraint manifold"""
        [span_86](start_span)norm = np.linalg.norm(state)
#[span_86](end_span)
        if norm > 0:
            [span_87](start_span)state = state / norm
#[span_87](end_span)
        return state

    def _detect_convergence_zones(self, state: np.ndarray) ->
np.ndarray:
        """Detect zones where trajectories converge (collapse
attractors)"""
        [span_88](start_span)epsilon = 1e-6 #[span_88](end_span)
        [span_89](start_span)divergence = np.zeros_like(state)
#[span_89](end_span)
        for i in range(len(state)):
            [span_90](start_span)perturbed_plus = state.copy()
#[span_90](end_span)
            [span_91](start_span)perturbed_minus = state.copy()
#[span_91](end_span)
            [span_92](start_span)perturbed_plus[i] += epsilon
#[span_92](end_span)
            [span_93](start_span)perturbed_minus[i] -= epsilon
#[span_93](end_span)
            [span_94](start_span)flow_plus =
self._flow_at_state(perturbed_plus) #[span_94](end_span)
            [span_95](start_span)flow_minus =
self._flow_at_state(perturbed_minus) #[span_95](end_span)
            [span_96](start_span)divergence[i] = (flow_plus[i] -
flow_minus[i]) / (2 * epsilon) #[span_96](end_span)

        [span_97](start_span)convergence_indicator = np.maximum(0,
-divergence) #[span_97](end_span)
        [span_98](start_span)convergence_zones =
(convergence_indicator > self.fold_threshold).astype(float)
#[span_98](end_span)
        return convergence_zones

    def _compute_historical_proximity(self, current_state: np.ndarray)
-> np.ndarray:
        """Weight predictions by proximity to historical collapse
```

```
patterns"""
        if len(self.manifold_memory) == 0:
            [span_99](start_span)return np.ones_like(current_state)
#[span_99](end_span)
        [span_100](start_span)proximity = np.zeros_like(current_state)
#[span_100](end_span)
        for sig in self.manifold_memory:
            [span_101](start_span)distance =
np.linalg.norm(current_state - sig.morphology) #[span_101](end_span)
            [span_102](start_span)weight = np.exp(-distance**2 / (2 *
sig.influence_radius**2)) #[span_102](end_span)
            [span_103](start_span)proximity += weight
#[span_103](end_span)
        [span_104](start_span)proximity = proximity /
(len(self.manifold_memory) + 1e-10) #[span_104](end_span)
        [span_105](start_span)return 1 + proximity
#[span_105](end_span)

    def _identify_intervention_manifolds(self, collapse_probability:
np.ndarray,
                                         curvature: np.ndarray) ->
np.ndarray:
        """Identify regions where minimal intervention can prevent
collapse"""
        [span_106](start_span)sensitivity =
np.zeros_like(collapse_probability) #[span_106](end_span)
        for i in range(len(collapse_probability)):
            [span_107](start_span)if collapse_probability[i] >
self.intervention_threshold: #[span_107](end_span)
                [span_108](start_span)jacobian =
np.gradient(curvature[i, :]) #[span_108](end_span)
                [span_109](start_span)sensitivity[i] = 1.0 /
(np.linalg.norm(jacobian) + 1e-10) #[span_109](end_span)

        if np.max(sensitivity) > 0:
            [span_110](start_span)sensitivity = sensitivity /
np.max(sensitivity) #[span_110](end_span)
        return sensitivity

    def _compute_cascade_risk(self, collapse_probability: np.ndarray,
                              entanglement: np.ndarray) -> float:
        """Compute risk of cascade failure across domains"""
        [span_111](start_span)high_risk = collapse_probability >
self.cascade_threshold #[span_111](end_span)
        if np.sum(high_risk) == 0:
            [span_112](start_span)return 0.0 #[span_112](end_span)

        [span_113](start_span)cascade_matrix = entanglement *
```

```
np.outer(high_risk, high_risk) #[span_113](end_span)
        [span_114](start_span)eigenvals =
np.linalg.eigvals(cascade_matrix) #[span_114](end_span)
        [span_115](start_span)cascade_risk =
np.real(np.max(eigenvals)) #[span_115](end_span)
        [span_116](start_span)return min(1.0, cascade_risk)
#[span_116](end_span)

    def _distance_to_bifurcation(self, state: np.ndarray) -> float:
        """Estimate distance to nearest bifurcation point"""
        [span_117](start_span)jacobian = self.curvature_field -
np.eye(self.dimensions) #[span_117](end_span)
        [span_118](start_span)eigenvals = np.linalg.eigvals(jacobian)
#[span_118](end_span)
        [span_119](start_span)real_parts = np.real(eigenvals)
#[span_119](end_span)
        [span_120](start_span)distance = np.min(np.abs(real_parts))
#[span_120](end_span)
        return distance

    def _identify_bifurcation_zones(self) -> np.ndarray:
        """Identify regions of parameter space near bifurcations"""
        if self.curvature_field is None:
            [span_121](start_span)return np.zeros((self.dimensions,
self.dimensions)) #[span_121](end_span)

        [span_122](start_span)bifurcation_indicator =
np.zeros((self.dimensions, self.dimensions)) #[span_122](end_span)
        for i in range(self.dimensions):
            for j in range(self.dimensions):
                [span_123](start_span)h = 1e-6 #[span_123](end_span)
                [span_124](start_span)f_pp = self.curvature_field[i,
j] #[span_124](end_span)
                if i > 0 and j > 0 and i < self.dimensions-1 and j <
self.dimensions-1:
                    [span_125](start_span)f_px =
self.curvature_field[i+1, j] #[span_125](end_span)
                    [span_126](start_span)f_mx =
self.curvature_field[i-1, j] #[span_126](end_span)
                    [span_127](start_span)f_py =
self.curvature_field[i, j+1] #[span_127](end_span)
                    [span_128](start_span)f_my =
self.curvature_field[i, j-1] #[span_128](end_span)
                    [span_129](start_span)hessian_trace = ((f_px -
2*f_pp + f_mx) + (f_py - 2*f_pp + f_my)) / h**2 #[span_129](end_span)
                    [span_130](start_span)bifurcation_indicator[i, j]
= abs(hessian_trace) #[span_130](end_span)
        return bifurcation_indicator
```

```
    def _compute_intervention_efficacy(self) -> np.ndarray:
        """Compute expected efficacy of interventions at each point"""
        if self.curvature_field is None:
            [span_131](start_span)return np.ones((self.dimensions,
self.dimensions)) #[span_131](end_span)

        [span_132](start_span)curvature_magnitude =
np.abs(self.curvature_field) #[span_132](end_span)
        [span_133](start_span)efficacy = 1.0 / (1.0 +
curvature_magnitude) #[span_133](end_span)
        return efficacy

    def _estimate_influence_radius(self, tensor: np.ndarray,
                                  curvature: np.ndarray) -> float:
        """Estimate the spatial influence radius of a collapse"""
        [span_134](start_span)tensor_scale = np.linalg.norm(tensor)
#[span_134](end_span)
        [span_135](start_span)curvature_scale =
np.linalg.norm(curvature) #[span_135](end_span)
        [span_136](start_span)radius = np.sqrt(tensor_scale) * (1 +
0.1 * curvature_scale) #[span_136](end_span)
        [span_137](start_span)return radius #[span_137](end_span)


class ManifoldCollapseAnalyzer:
    """Integrates Catastrophe Manifold Folding with FractureMetrics"""

    def __init__(self, manifold_folder: CatastropheManifoldFolder):
        """Initialize with a manifold folder instance"""
        [span_138](start_span)self.manifold_folder = manifold_folder
#[span_138](end_span)

    def _metrics_to_tensor(self, metrics: DomainMetrics,
                           prev_values: np.ndarray = None) ->
Tuple[np.ndarray, np.ndarray]:
        """Convert domain metrics to collapse tensor"""
        values = np.array([
            metrics.institutional_trust,
            metrics.information_integrity,
            metrics.electoral_confidence,
            metrics.alliance_stability,
            metrics.social_cohesion
        [span_139](start_span)]) #[span_139](end_span)

        if prev_values is not None:
            [span_140](start_span)gradient = values - prev_values
#[span_140](end_span)
```

```
        else:
            [span_141](start_span)gradient = np.zeros_like(values)
#[span_141](end_span)

        # Construct tensor
        [span_142](start_span)tensor = np.outer(values, values) + 0.1
* np.outer(gradient, gradient) #[span_142](end_span)
        return tensor, values

# --- 🧬 2. Main Execution Block ---

if __name__ == "__main__":
    print("🚀 Initializing Collapse Engine Stack...")

    # --- Input State Definition (Synthetic Instability) ---
    initial_metrics = DomainMetrics(
        institutional_trust=0.42,
        information_integrity=0.35,
        electoral_confidence=0.60,
        alliance_stability=0.28,
        social_cohesion=0.31,
        timestamp=datetime.datetime.utcnow()
    )
    print(f"▸ Processing Input State at
{initial_metrics.timestamp.isoformat()}")

    # --- Engine Instantiation & Tensor Creation ---
    [span_143](start_span)manifold =
CatastropheManifoldFolder(dimensions=5) #[span_143](end_span)
    [span_144](start_span)analyzer =
ManifoldCollapseAnalyzer(manifold) #[span_144](end_span)

    # In a real scenario, you would have previous values. For this
single run, we assume none.
    [span_145](start_span)[span_146](start_span)collapse_tensor,
current_values = analyzer._metrics_to_tensor(initial_metrics,
prev_values=None) #[span_145](end_span)[span_146](end_span)
    print("    ...Collapse Tensor generated.")

    # --- Manifold Folding ---
    [span_147](start_span)[span_148](start_span)folded_state =
manifold.fold_manifold(collapse_tensor, domain="truth-tech",
coupling_strength=0.2) #[span_147](end_span)[span_148](end_span)
    print("    ...Manifold folding complete.")

    # --- Prediction & Bifurcation Analysis ---
    # Flatten the 5x5 tensor to a 25-element vector for prediction
    current_state_vector = collapse_tensor.flatten()
```

```
    predictions = manifold.predict_collapse_zones(
        current_state=current_state_vector,
        horizon=12,
        trajectories=200
    [span_149](start_span)[span_150](start_span))
#[span_149](end_span)[span_150](end_span)
    print("    ...Collapse and bifurcation zones predicted.")

    # --- Intervention Analysis ---
    intervention_zones = predictions.get('intervention_zones',
np.array([]))
    intervention_efficacy = folded_state.intervention_efficacy
    print("    ...Intervention manifolds calculated.")

    # --- Output Logging ---
    output_log = {
        "run_timestamp": datetime.datetime.utcnow().isoformat(),
        "input_state": initial_metrics.__dict__,
        "analysis_output": {
            "collapse_probability":
predictions.get('collapse_probability', []).tolist(),
            "intervention_zones": intervention_zones.tolist(),
            "cascade_risk": predictions.get('cascade_risk'),
            "distance_to_bifurcation":
predictions.get('bifurcation_distance'),
            "bifurcation_zones_indicator":
folded_state.bifurcation_zones.tolist(),
            "intervention_efficacy_map":
intervention_efficacy.tolist(),
        },
        "manifold_snapshot": {
            "curvature_field": folded_state.curvature_field.tolist(),
            "entanglement_matrix":
folded_state.entanglement_matrix.tolist(),
            "memory_tensor": folded_state.memory_tensor.tolist()
        }
    }

    # Custom JSON encoder for datetime objects
    class DateTimeEncoder(json.JSONEncoder):
        def default(self, obj):
            if isinstance(obj, datetime.datetime):
                return obj.isoformat()
            return json.JSONEncoder.default(self, obj)

    log_filename = "collapse_output_log.json"
    with open(log_filename, "w") as f:
```

```
        json.dump(output_log, f, indent=2, cls=DateTimeEncoder)

    print(f"\n✅ Analysis Complete. Full output logged to:
{log_filename}")
```