Alright, let’s actually ship this thing. Below is a **ready-to-drop repo**: minimal, clean, and consistent with everything we designed. It’ll run as a basic prototype and gives you clear places to plug in your real Ziltrit tables, AEON geometry, and more advanced mode rules.

1. Repo structure

Call the repo whatever you like; I’ll use: Ziltrit-Young-AEON Ziltrit-Young-AEON/ README.md pyproject.toml (or requirements.txt if you prefer) ziltrit_aeon_young/ __init__.py symbols/ __init__.py glyph_table.py encoder.py geometry/ __init__.py torus.py deformations.py physics/ __init__.py waves.py elasticity.py modes/ __init__.py sequences.py selectors.py interference/ __init__.py field_solver.py scoring.py cipher/ __init__.py parser.py

candidates.py ranking.py viz/ __init__.py plots.py scripts/ demo_pipeline.py
Now the files.

2. README.md

Ziltrit–Young–AEON Engine A unified experimental engine that: – maps **symbols** (Ziltrit or mapped ciphers) – into **geometry** (AEON–style torus / deformations) – configures **wave modes** (Lucas / primes / φ(n) etc.) – simulates **interference patterns** (Young) – and exposes tools for **cipher ranking** and **field/coil design**. This is a **research prototype**, not a finished product. The goal is to provide a clean, testable core that matches the architecture: > Glyph → Mode → Field → Interference → Score ## Features (v0.1) – Ziltrit–style glyph parameter model (`GlyphParams`) – Torus geometry builder

(`GeometryConfig`, `torus_coords`) – Basic harmonic mode selection using Lucas + primes (`Mode`) – Young–style wave superposition and intensity map – Simple interference scoring (`Q` = smoothness − λ·variance) – Cipher bridge: – text → symbols → glyphs – φ(p) warps – configuration ranking – Simple visualization (intensity heatmap) ## Install Either with `pyproject.toml` (recommended): ```bash pip install −e . or with bare requirements:
pip install numpy matplotlib

Quick start

python scripts/demo_pipeline.py
This:

- uses a tiny example glyph table
- builds a torus field
- computes an interference pattern
- shows a heatmap

From there you can:

- replace the toy GLYPH_TABLE with your real Ziltrit glyphs
- refine modes/selectors.py with your Lucas / Riemann / Kepler rules
- extend geometry/deformations.py with your AEON / Worm deformations
- extend the cipher modules toward Kryptos / Cicada / Voynich experiments

Disclaimer

This code is exploratory and intentionally simplified.
It is designed for research and experimentation, not for production.
––– ## 3. `pyproject.toml` (optional but nicer) ```toml [project] name = "ziltrit-aeon-young" version = "0.1.0" description = "Symbol → geometry → wavefield → interference engine for research." authors = [{ name = "Wizard", email = "" }] requires-

python = ">=3.10" dependencies = ["numpy", "matplotlib",] [build-system] requires = ["setuptools>=61.0"] build-backend = "setuptools.build_meta"
If you prefer old school, skip this and use requirements.txt with:
numpy matplotlib

4. Package root: ziltrit_aeon_young/__init__.py

from .symbols.encoder import encode_sequence, apply_phi_warp from .interference.field_solver import compute_interference, build_field_config from .viz.plots import plot_intensity_for_sequence __all__ = ["encode_sequence", "apply_phi_warp", "compute_interference", "build_field_config", "plot_intensity_for_sequence",]

5. symbols package

symbols/__init__.py

from .glyph_table import GLYPH_TABLE from .encoder import GlyphParams, encode_sequence, apply_phi_warp __all__ = ["GLYPH_TABLE", "GlyphParams", "encode_sequence", "apply_phi_warp"]

symbols/glyph_table.py

from dataclasses import dataclass from math import tau from typing import Dict @dataclass class GlyphParams: name: str index: int # glyph index prime: int # associated prime angle_rad: float # base angle layer: int # 0 = inner, 1 = mid, 2 = outer spiral_sign: int # +1 or −1 deformation_type: str # "none", "bend", "twist", etc. # TODO: Replace this with your real Ziltrit glyph set. # This is only a minimal example to make the engine

runnable. GLYPH_TABLE: Dict[str, GlyphParams] = { "Z1": GlyphParams("Z1", index=0, prime=2, angle_rad=0.0, layer=0, spiral_sign=+1, deformation_type="none"), "Z2": GlyphParams("Z2", index=1, prime=3, angle_rad=tau / 3.0, layer=1, spiral_sign=−1, deformation_type="bend"), "Z3": GlyphParams("Z3", index=2, prime=5, angle_rad=2*tau/3.0, layer=2, spiral_sign=+1, deformation_type="twist"), }

symbols/encoder.py

from typing import List from math import tau from .glyph_table import GLYPH_TABLE, GlyphParams def encode_sequence(symbols: List[str]) -> List[GlyphParams]: """Map a sequence of symbol names to their GlyphParams. Raises KeyError if a symbol is not defined

in the table. """ return [GLYPH_TABLE[s] for s in symbols] def apply_phi_warp(glyphs: List[GlyphParams], k: int) -> List[GlyphParams]: """Apply a simple Euler totient–style warp using φ(p) = p − 1. This modifies the index and angle based on k * φ(p). The table size is assumed to be len(GLYPH_TABLE). """ N = len(GLYPH_TABLE) or 1 warped: List[GlyphParams] = [] for g in glyphs: phi_p = g.prime − 1 new_index = (g.index + k * phi_p) % N angle_step = tau / N new_angle = (g.angle_rad + k * phi_p * angle_step) % tau warped.append(GlyphParams(name=g.name, index=new_index, prime=g.prime, angle_rad=new_angle, layer=g.layer, spiral_sign=g.spiral_sign, deformation_type=g.deformation_type,)) return warped

6. geometry package

geometry/__init__.py

```
from .torus import GeometryConfig, torus_coords, build_geometry_from_glyphs from .deformations import apply_deformations __all__ = ["GeometryConfig", "torus_coords", "build_geometry_from_glyphs", "apply_deformations"]
```

geometry/torus.py

```
from dataclasses import dataclass from typing import List, Tuple import numpy as np from ..symbols.encoder import GlyphParams @dataclass class GeometryConfig: R: float # major radius r: float # minor radius twists: int deformations: dict def torus_coords(R: float, r: float, theta: np.ndarray, phi: np.ndarray) -> Tuple[np.ndarray,
```

```
np.ndarray, np.ndarray]: """Standard torus parameterization.""" x = (R + r * np.cos(theta)) * np.cos(phi) y = (R + r * np.cos(theta)) * np.sin(phi) z = r * np.sin(theta) return x, y, z def build_geometry_from_glyphs(glyphs: List[GlyphParams]) -> GeometryConfig: """Derive a simple geometry configuration from glyph parameters. This is intentionally basic; refine with AEON/Worm logic later. """ if not glyphs: return GeometryConfig(R=1.0, r=0.3, twists=1, deformations={}) avg_prime = sum(g.prime for g in glyphs) / len(glyphs) sum_signs = sum(g.spiral_sign for g in glyphs) unique_layers = len({g.layer for g in glyphs}) R = 1.0 + 0.01 * avg_prime r = 0.25 + 0.02 * unique_layers twists = max(1, abs(sum_signs)) deformations = { "twist_bias": sum_signs, "avg_prime": avg_prime, "layers": unique_layers, }
```

return GeometryConfig(R=R, r=r, twists=twists, deformations=deformations)

geometry/deformations.py

from typing import Tuple import numpy as np from .torus import GeometryConfig def apply_deformations(coords: Tuple[np.ndarray, np.ndarray, np.ndarray], config: GeometryConfig,) -> Tuple[np.ndarray, np.ndarray, np.ndarray]: """Placeholder deformation function. Currently returns the coordinates unchanged. Extend this with AEON / Worm bending and twisting logic. """ x, y, z = coords # Example: very mild global twist based on twist_bias twist_bias = float(config.deformations.get("twist_bias", 0)) if twist_bias != 0: factor = 0.01 * twist_bias z = z + factor * x return x, y, z

7. physics package

physics/__init__.py

from .waves import Mode, FieldConfig, wavefield_on_torus, intensity_map from .elasticity import max_strain_allowed __all__ = ["Mode", "FieldConfig", "wavefield_on_torus", "intensity_map", "max_strain_allowed"]

physics/waves.py

from dataclasses import dataclass from typing import List import numpy as np from ..geometry.torus import GeometryConfig, torus_coords, apply_deformations @dataclass class Mode: n: int m: int omega: float phase: float amplitude: float @dataclass class FieldConfig: geometry: GeometryConfig modes: List[Mode] grid_theta: np.ndarray grid_phi: np.ndarray def wavefield_on_torus(field_cfg: FieldConfig, t: float) -> np.ndarray:

"""Compute the scalar wavefield F(θ, φ, t) on the torus surface.""" θ = field_cfg.grid_theta φ = field_cfg.grid_phi psi = np.zeros_like(θ, dtype=float) for mode in field_cfg.modes: psi += mode.amplitude * np.cos(mode.n * θ + mode.m * φ − mode.omega * t + mode.phase) # geometry currently not feeding back into field amplitude: keep simple for v0.1 return psi def intensity_map(field_cfg: FieldConfig, num_samples: int = 32) -> np.ndarray: """Time-averaged intensity map I(θ, φ) ≈ <F^2>_t.""" ts = np.linspace(0.0, 2.0 * np.pi, num_samples) acc = np.zeros_like(field_cfg.grid_theta, dtype=float) for t in ts: acc += wavefield_on_torus(field_cfg, t) ** 2 return acc / float(num_samples)

physics/elasticity.py

def max_strain_allowed(E: float,

sigma_max: float) -> float: """Return maximum strain from Young's modulus and max allowable stress.""" if E <= 0: raise ValueError("Young's modulus must be positive.") return sigma_max / E

8. modes package

modes/__init__.py

from .sequences import lucas_sequence from .selectors import modes_from_glyphs __all__ = ["lucas_sequence", "modes_from_glyphs"]

modes/sequences.py

from typing import List def lucas_sequence(n: int) -> List[int]: """Return the first n Lucas numbers.""" if n <= 0: return [] if n == 1: return [2] seq = [2, 1] while len(seq) < n: seq.append(seq[-1] + seq[-2]) return seq

modes/selectors.py

from typing import List from

..symbols.encoder import GlyphParams from .sequences import lucas_sequence from ..physics.waves import Mode def modes_from_glyphs(glyphs: List[GlyphParams]) −> List[Mode]: """Derive harmonic modes from glyphs using a simple Lucas + prime rule. This is deliberately simple; it's the place to integrate your full Unified Everything / Riemann / Kepler logic later. """ if not glyphs: return [] L = lucas_sequence(len(glyphs) + 5) modes: List[Mode] = [] for i, g in enumerate(glyphs): n = (g.index + L[i]) % 12 + 1 m = (g.prime % 12) + 1 omega = 0.5 + 0.01 * g.prime phase = g.angle_rad amplitude = 1.0 modes.append(Mode(n=n, m=m, omega=omega, phase=phase, amplitude=amplitude)) return modes

9. interference package

interference/__init__.py

```
from .field_solver import build_field_config, compute_interference
from .scoring import score_intensity
__all__ = ["build_field_config", "compute_interference", "score_intensity"]
```

interference/scoring.py

```
from typing import Dict import numpy as np def score_intensity(I: np.ndarray, lambda_var: float = 1.0) -> Dict[str, float]: """Compute basic statistics and a composite quality score Q. Q = smoothness - lambda_var * variance """ mean_val = float(I.mean()) var_val = float(I.var()) gy, gx = np.gradient(I) grad_energy = float((gx**2 + gy**2).mean()) smoothness = float(1.0 / (1e-9 + grad_energy)) Q = smoothness - lambda_var * var_val return { "mean":
```

mean_val, "var": var_val, "smoothness": smoothness, "Q": Q, }

interference/field_solver.py

from dataclasses import dataclass from typing import List import numpy as np from ..symbols.encoder import GlyphParams from ..geometry.torus import build_geometry_from_glyphs from ..physics.waves import FieldConfig, intensity_map from .scoring import score_intensity @dataclass class InterferencePattern: intensity: np.ndarray stats: dict def build_field_config(glyphs: List[GlyphParams], n_theta: int = 128, n_phi: int = 128,) -> FieldConfig: geometry = build_geometry_from_glyphs(glyphs) θ = np.linspace(0.0, 2.0 * np.pi, n_theta) φ = np.linspace(0.0, 2.0 * np.pi, n_phi) Θ, Φ = np.meshgrid(θ, φ, indexing="ij") from ..modes.selectors import

```
modes_from_glyphs modes = modes_from_glyphs(glyphs) return FieldConfig(geometry=geometry, modes=modes, grid_theta=Θ, grid_phi=Φ) def compute_interference(glyphs: List[GlyphParams]) -> InterferencePattern: field_cfg = build_field_config(glyphs) I = intensity_map(field_cfg) stats = score_intensity(I) return InterferencePattern(intensity=I, stats=stats)
```

10. cipher package

cipher/__init__.py

```
from .parser import text_to_symbol_sequence from .candidates import generate_config_variants from .ranking import rank_configs __all__ = ["text_to_symbol_sequence",
```

"generate_config_variants", "rank_configs"]

cipher/parser.py

from typing import List def text_to_symbol_sequence(text: str) -> List[str]: """Very simple mapping: non-space characters → symbol names. For now we just use each character as a symbol directly. In a real setup, map characters / runes → Ziltrit glyph names. """ return [ch for ch in text if not ch.isspace()]

cipher/candidates.py

from typing import Iterable, Dict, Any from ..symbols.encoder import encode_sequence, apply_phi_warp from ..interference.field_solver import compute_interference def generate_config_variants(symbols: Iterable[str], ks=range(0, 4),) ->

list[Dict[str, Any]]: """Generate φ(p)−warp variants and compute interference patterns.""" symbol_list = list(symbols) glyphs_base = encode_sequence(symbol_list) configs: list[Dict[str, Any]] = [] for k in ks: glyphs_k = apply_phi_warp(glyphs_base, k) pattern = compute_interference(glyphs_k) configs.append({ "k": k, "glyphs": glyphs_k, "pattern": pattern, }) return configs

cipher/ranking.py

from typing import Iterable, Dict, Any, List from .parser import text_to_symbol_sequence from .candidates import generate_config_variants def rank_configs(text: str, ks=range(0, 4)) -> List[Dict[str, Any]]: """Rank φ(p)−warp configurations by interference score Q.""" symbols =

text_to_symbol_sequence(text) configs = generate_config_variants(symbols, ks=ks) for cfg in configs: cfg["score"] = cfg["pattern"].stats.get("Q", 0.0) return sorted(configs, key=lambda c: c["score"], reverse=True)

11. viz package

viz/__init__.py

from .plots import plot_intensity_for_sequence __all__ = ["plot_intensity_for_sequence"]

viz/plots.py

from typing import List import matplotlib.pyplot as plt from ..symbols.encoder import encode_sequence from ..interference.field_solver import compute_interference def plot_intensity_for_sequence(symbols: List[str]) -> None: """Compute and plot the

interference intensity for a glyph sequence."""" glyphs = encode_sequence(symbols) pattern = compute_interference(glyphs) I = pattern.intensity plt.figure(figsize=(6, 5)) plt.imshow(I, origin="lower", interpolation="bilinear") plt.colorbar(label="Intensity") plt.title("Interference Pattern") plt.xlabel("φ index") plt.ylabel("θ index") plt.tight_layout() plt.show()

12. Demo script

scripts/demo_pipeline.py
import sys from pathlib import Path # Allow running from repo root sys.path.append(str(Path(__file__).resolve().parents[1])) from ziltrit_aeon_young.symbols.encoder import encode_sequence from ziltrit_aeon_young.interference.field_solv

er import compute_interference from ziltrit_aeon_young.viz.plots import plot_intensity_for_sequence def main() -> None: # Example uses the toy glyphs defined in glyph_table.py: Z1, Z2, Z3 seq = ["Z1", "Z2", "Z3", "Z1", "Z3", "Z2"] glyphs = encode_sequence(seq) pattern = compute_interference(glyphs) print("Stats:", pattern.stats) plot_intensity_for_sequence(seq) if __name__ == "__main__": main()

That’s your repo built.

Next practical step on your side:

- Create the folders.
- Paste these files in.
- Swap the toy GLYPH_TABLE for your real Ziltrit v4 alphabet when you’re ready.
- Then we start tightening individual modules (modes, deformations, cipher mappings) one by one.