RSVP–TARTAN–CLIO Experimental Architecture and Roadmap

Abstract & Purpose

This document specifies a headless experimental architecture that unifies RSVP’s scalar–vector–entropy field dynamics, TARTAN’s recursive lattice constraints, and CLIO’s semantic drift/flow modeling. It includes project specifications, a categorization matrix, execution templates for Blender (bpy), Python orchestration, and shell automation, plus a full set of analytic/merging/recursive operators. The layout is preprint-style with generous margins and wrapped code listings to avoid overflow.

System Overview & Directory Layout

```
rsvp_tartan_clio/
■■■ experiments/
■   ■■■ Tier_I/
■   ■■■ Tier_II/
■   ■■■ Tier_III/
■   ■■■ Tier_IV/
■   ■■■ Tier_V/
■■■ bpy_scripts/
■   ■■■ generate_experiment.py
■   ■■■ simulate_entropy_field.py
■   ■■■ render_snapshot.py
■■■ python_ops/
■   ■■■ operators.py
■   ■■■ orchestrator.py
■   ■■■ config.json
■■■ automation/
■   ■■■ run_all.sh
■   ■■■ run_analysis.sh
■   ■■■ environment_setup.sh
■   ■■■ cron_schedule.txt
■■■ logs/
        ■■■ experiment_logs/
        ■■■ analysis/
```

Categorization Matrix

Dimension	Category	Operators / Modules	Key Metrics	Output
Analytic	Correlation / Causality	Entropy–Curvature Mapper, Ethical-Field Analyzer, Phase-Coherence Mapper	Pearson r, MI, Granger F, PLV	analytic_summary.json
Comparative	Cross-tier & Cross-experiment	Drift-Phase Synchronizer, Recursive Tier Comparator	Trend slope, coherence	comparative_matrix.json
Morphic / Generative	Interpolation & Latent Synthesis	Morphic Interpolator, Latent Synthesizer, Gradient Aligner	Interp loss, explained var	synthetic_experiments.json
Merging / Integration	Field Fusion & Validation	Veil Merger, Boundary Validator, Derived Stack Aggregator	Seam penalty, JS divergence	merged_fields.json
Recursive / Control	Adaptive & Longitudinal	Recursive Reweighter, Cycle-Bloom Trigger, Orchestrator DAG	Meta-weights, convergence	meta_control.json
Composite	Operators-on-Operators	Meta-Trend Extractor, Ω-Composer	R^2 across tiers, stability index	meta_trends.json

Project Specifications

P1. Headless RSVP–TARTAN–CLIO Corpus — 20+ experiments across Tiers I–IV with unified logs (entropy.json, curvature.json, metrics.json).
P2. Operator Library — 13+ operators (analytic, merging, morphic, recursive) with JSON I/O and batch safety.
P3. Orchestrator (Ω-Composer) — DAG executor for operators; registry and dependency resolution.
P4. Recursive Tier Comparator — longitudinal analysis across same-title experiments by tier.
P5. Entropy–Curvature Suite — correlation mapper, spectral fusion, gradient aligner.
P6. Drift–Phase Synchrony Analyzer — PLV/coherence across cyclic experiments.
P7. Ethical-Field Governance Layer — damping, Granger causality, divergence monitoring.
P8. Latent Synthesizer — PCA/autoencoder for metric embeddings and pseudo-experiment interpolation.
P9. Evolution Controller — recursive reweighter + phase triggers for next-run selection.
P10. Reporting Pipeline — dashboards, summary JSONs, reproducible plots.

Execution Pipeline

1) ./automation/environment_setup.sh
2) Generate experiments (headless Blender): blender -b -P bpy_scripts/generate_experiment.py -- --name exp_
3) Run analysis (Python operators): python python_ops/orchestrator.py --tier Tier_III
4) Aggregate reports: logs/analysis//summary.json
5) Schedule nightly runs via cron: 03:00 local.

Appendix A — Analytic Operators (Summaries)

• Entropy–Curvature Correlation Mapper — correlates per-vertex entropy gradients with curvature; outputs Pearson r, clusters.
• Drift–Phase Coherence Synchronizer — PLV and coherence spectra across drift/cycle experiments.
• Ethical-Field Coupling Analyzer — Granger causality between turbulence suppression and ethical damping.
• Turbulence–Knot Complexity Correlator — FFT energy bands vs knot counts (Poisson GLM).
• Veil–Flow Threshold Optimizer — maximizes hidden-route emergence subject to entropy leakage constraints.

Appendix B — Merging & Morphic Operators (Summaries)

• Veil-Transparency Field Merger — composites alpha maps to unified occlusion fields.
• Bloom-Front Asymmetry Quantifier — principal-axis drift alignment and asymmetry tensors.
• Cross-Tier Entropy Gradient Aligner — RBF interpolation across tier index as latent axis.
• Resonance Feedback Interpolator — SLERP in parameter manifold; intermediate decay laws.
• Lattice–Ethics Turbulence Suppressor — consensus damping field with GP smoothing.

Appendix C — Recursive & Composite Operators (Summaries)

• Recursive Tier Comparator — depth-first across tiers; trend slopes and convergence.
• Cycle–Bloom Phase Trigger Generator — logistic onset fit with cross-validation.
• Recursive Meta-Trend Extractor — second-order trends over analytic outputs.
• Ω-Composer — DAG registry and execution with unified JSON bundle.

Appendix D — Full-Length Code Templates (with annotations)

D.1 Blender: generate_experiment.py

```
#!/usr/bin/env python3
# Headless Blender experiment generator for RSVP–TARTAN–CLIO corpus
# - Creates a tiled manifold (torus grid), assigns scalar entropy and estimated curvature,
# - Builds optional particle flow to simulate vector fields,
# - Exports JSON logs, OBJ mesh, and a PNG snapshot for verification.
#
# Usage:
#   blender -b -P bpy_scripts/generate_experiment.py -- --name exp_entropy_cascade --tier
Tier_I --frames 120

import bpy, bmesh, sys, argparse, json, math, random
from mathutils import Vector
from pathlib import Path

def parse_args():
        argv = sys.argv
        if "--" in argv:
                argv = argv[argv.index("--") + 1:]
        parser = argparse.ArgumentParser()
        parser.add_argument("--name", required=True, help="experiment name (folder)")
        parser.add_argument("--tier", default="Tier_I")
        parser.add_argument("--frames", type=int, default=60)
        parser.add_argument("--seed", type=int, default=1337)
        parser.add_argument("--subdiv", type=int, default=3, help="subdivisions per torus")
        return parser.parse_args(argv)

def ensure_dir(p: Path):
        p.mkdir(parents=True, exist_ok=True)

def create_torus_grid(rows=2, cols=2, radius=1.0, tube=0.35, spacing=2.8):
        tori = []
        for r in range(rows):
                for c in range(cols):
                        bpy.ops.mesh.primitive_torus_add(major_radius=radius,
            minor_radius=tube)
                        obj = bpy.context.active_object
                        obj.location = (c * spacing, r * spacing, 0.0)
                        tori.append(obj)
        # Join into single object for simpler export
        ctx = bpy.context.copy()
        ctx["active_object"] = tori[0]
        ctx["selected_editable_objects"] = tori
        bpy.ops.object.join(ctx)
        return tori[0]

def subdivide(obj, levels=2):
        bpy.context.view_layer.objects.active = obj
        bpy.ops.object.modifier_add(type='SUBSURF')
        obj.modifiers["Subdivision"].levels = levels
        bpy.ops.object.modifier_apply(modifier="Subdivision")

def compute_entropy_and_curvature(obj):
        # Simple proxies: entropy ~ vertex noise; curvature ~ Laplacian magnitude (approx).
        import random, math
        mesh = obj.data
        verts = mesh.vertices
        entropy = {}
        curvature = {}
        # Build bmesh for adjacency
        bm = bmesh.new()
        bm.from_mesh(mesh)
        bm.verts.ensure_lookup_table()
        # Precompute neighbor lists
        neighbors = {v.index: set() for v in bm.verts}
        for e in bm.edges:
                a, b = e.verts[0].index, e.verts[1].index
                neighbors[a].add(b); neighbors[b].add(a)
        # Assign entropy and compute crude Laplacian
        for v in bm.verts:
                ent = random.random()  # replace with noise if desired
                entropy[v.index] = ent
```

```
        for v in bm.verts:
                nbrs = neighbors[v.index]
                if not nbrs:
                        curvature[v.index] = 0.0
                        continue
                avg = sum(entropy[n] for n in nbrs) / len(nbrs)
                curvature[v.index] = abs(entropy[v.index] - avg)  # Laplacian proxy
        bm.free()
        return entropy, curvature

def export_obj(obj, out_path: Path):
        # Export a lightweight OBJ for downstream analysis
        original_selection = [o for o in bpy.context.selected_objects]
        bpy.ops.object.select_all(action='DESELECT')
        obj.select_set(True)
        bpy.context.view_layer.objects.active = obj
        bpy.ops.export_scene.obj(filepath=str(out_path), use_selection=True,
    use_materials=False)
        for o in original_selection:
                o.select_set(True)

def configure_render(exp_dir: Path):
        scene = bpy.context.scene
        scene.render.engine = 'BLENDER_EEVEE'
        scene.render.image_settings.file_format = 'PNG'
        scene.render.resolution_x = 1600
        scene.render.resolution_y = 1200
        scene.render.filepath = str(exp_dir / "snapshot.png")

def main():
        args = parse_args()
        random.seed(args.seed)
        exp_dir = Path("experiments") / args.tier / args.name
        ensure_dir(exp_dir)

        # Reset scene
        bpy.ops.wm.read_factory_settings(use_empty=True)
        bpy.ops.object.camera_add(location=(6, -8, 6))
        bpy.context.scene.camera = bpy.context.active_object
        bpy.ops.object.light_add(type='SUN', location=(10, 10, 10))

        # Geometry
        obj = create_torus_grid(rows=2, cols=2, radius=1.2, tube=0.35, spacing=3.0)
        subdivide(obj, levels=args.subdiv)

        # Fields
        entropy, curvature = compute_entropy_and_curvature(obj)

        # Export logs
        with open(exp_dir / "entropy.json", "w") as f: json.dump(entropy, f, indent=2)
        with open(exp_dir / "curvature.json", "w") as f: json.dump(curvature, f, indent=2)
        with open(exp_dir / "metrics.json", "w") as f:
                json.dump({"tier": args.tier, "verts": len(obj.data.vertices)}, f, indent=2)

        # Export mesh & snapshot
        export_obj(obj, exp_dir / "mesh.obj")
        configure_render(exp_dir)
        bpy.ops.render.render(write_still=True)

        print(f"[✓] Generated {args.name} in {exp_dir}")

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

D.1 Blender: simulate_entropy_field.py

```
#!/usr/bin/env python3
# Simulate scalar entropy over frames with simple diffusion + noise injection.
# Produces time-series logs and optional per-frame PNGs (disabled by default).

import bpy, json, math, random, sys
from pathlib import Path

FRAMES = 100
ALPHA = 0.92     # retention
NOISE = 0.08     # random injection
TILE = (2, 2)    # torus grid shape
```

```
def ensure_dir(p: Path): p.mkdir(parents=True, exist_ok=True)

def init_scene():
        bpy.ops.wm.read_factory_settings(use_empty=True)
        bpy.ops.object.camera_add(location=(6, -8, 6))
        bpy.ops.object.light_add(type='SUN', location=(10, 10, 10))
        bpy.context.scene.camera = bpy.context.active_object

def torus_grid(rows=2, cols=2):
        objs = []
        for r in range(rows):
                for c in range(cols):
                        bpy.ops.mesh.primitive_torus_add(major_radius=1.2,
            minor_radius=0.35)
                        o = bpy.context.active_object
                        o.location = (c * 3.0, r * 3.0, 0.0)
                        objs.append(o)
        ctx = bpy.context.copy()
        ctx["active_object"] = objs[0]
        ctx["selected_editable_objects"] = objs
        bpy.ops.object.join(ctx)
        return objs[0]

def simulate_entropy(obj, frames=FRAMES, alpha=ALPHA, noise=NOISE):
        import random
        mesh = obj.data
        n = len(mesh.vertices)
        s = [random.random() for _ in range(n)]
        series = []
        for t in range(frames):
                # Diffuse by averaging with neighbors (approx via object-space smoothing)
                s2 = []
                for i, v in enumerate(mesh.vertices):
                        # crude local average using nearby vertices by index (proxy)
                        left = s[i-1] if i > 0 else s[i]
                        right = s[i+1] if i < n-1 else s[i]
                        avg = 0.5 * (left + right)
                        val = alpha * s[i] + (1-alpha) * avg + noise * (random.random() -
            0.5)
                        s2.append(max(0.0, min(1.0, val)))
                s = s2
                series.append(sum(s) / n)
        return series

def main():
        out_dir = Path("experiments/Tier_II/exp_entropy_flow")
        ensure_dir(out_dir)
        init_scene()
        obj = torus_grid(*TILE)
        series = simulate_entropy(obj)
        with open(out_dir / "entropy_series.json", "w") as f:
                json.dump({"mean_entropy": series}, f, indent=2)
        print(f"[✓] Simulated entropy series → {out_dir}")

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

D.1 Blender: render_snapshot.py

```
#!/usr/bin/env python3
# Import OBJ and render a verification PNG from a fixed camera/light rig.

import bpy, sys
from pathlib import Path

def setup_scene():
        bpy.ops.wm.read_factory_settings(use_empty=True)
        bpy.ops.object.camera_add(location=(6, -8, 6))
        bpy.ops.object.light_add(type='SUN', location=(10, 10, 10))
        bpy.context.scene.camera = bpy.context.active_object
        bpy.context.scene.render.engine = 'BLENDER_EEVEE'
        bpy.context.scene.render.resolution_x = 1600
        bpy.context.scene.render.resolution_y = 1200

def main():
        argv = sys.argv
        if "--" in argv: argv = argv[argv.index("--") + 1:]
        obj_path = Path(argv[0])
```

```
    out_png = Path(argv[1]) if len(argv) > 1 else obj_path.with_suffix(".png")

    setup_scene()
    bpy.ops.import_scene.obj(filepath=str(obj_path))
    bpy.context.scene.render.filepath = str(out_png)
    bpy.ops.render.render(write_still=True)
    print(f"[✓] Rendered {out_png}")

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

D.2 Python: operators.py (annotated library)

```
#!/usr/bin/env python3

"""
Operator library implementing analytic / merging / recursive functions
over experiment directories that contain JSON logs (entropy.json, curvature.json,
metrics.json).
All outputs are pure JSON dictionaries for composability.
"""
import json, numpy as np, os
from pathlib import Path
from typing import List, Dict

def _load_json(p: Path, default=None):
        try:
                with open(p, "r") as f: return json.load(f)
        except FileNotFoundError:
                return {} if default is None else default

def entropy_curvature_correlation(exp_dirs: List[str]) -> Dict:
        """
        Compute Pearson correlation between per-vertex entropy and curvature proxies.
        Returns an aggregate summary across all experiments.
        """
        corrs = []
        for d in exp_dirs:
                d = Path(d)
                e = _load_json(d / "entropy.json")
                k = _load_json(d / "curvature.json")
                keys = sorted(set(e) & set(k))
                if not keys:
                        continue
                x = np.array([e[i] for i in keys], dtype=float)
                y = np.array([k[i] for i in keys], dtype=float)
                if len(x) > 1 and np.std(x) > 0 and np.std(y) > 0:
                        corrs.append(float(np.corrcoef(x, y)[0, 1]))
        return {"count": len(corrs), "mean_corr": float(np.mean(corrs)) if corrs else None}

def drift_phase_coherence(exp_dirs: List[str]) -> Dict:
        """
        Compute a simple phase-locking value (PLV) between drift and a reference phase
    series.
        Expects entropy_series.json or phase.json.
        """
        def plv(phases):
                phases = np.array(phases)
                return float(np.abs(np.mean(np.exp(1j*phases))))

        plvs = []
        for d in exp_dirs:
                d = Path(d)
                series = _load_json(d / "entropy_series.json")
                phases = series.get("mean_entropy", [])
                if len(phases) > 5:
                        # crude phase from normalized series
                        s = (phases - np.mean(phases)) / (np.std(phases) + 1e-8)
                        # map to angle domain [-pi, pi] via tanh
                        theta = np.pi * np.tanh(s)
                        plvs.append(plv(theta))
        return {"count": len(plvs), "mean_plv": float(np.mean(plvs)) if plvs else None}

def veil_visibility_optimizer(exp_dirs: List[str]) -> Dict:
        """
        Placeholder optimizer: in real runs, would scan alpha thresholds vs. information
    gain.
        Here we synthesize outputs to show JSON schema.
        """
        results = {}
        for d in exp_dirs:
                best_alpha = 0.42  # placeholder
                info_gain = 0.18
                results[str(d)] = {"alpha": best_alpha, "info_gain": info_gain}
        return {"experiments": results}

def run_registry(op_name: str, exp_dirs: List[str]) -> Dict:
        registry = {
                "entropy_curvature_correlation": entropy_curvature_correlation,
                "drift_phase_coherence": drift_phase_coherence,
```

```
            "veil_visibility_optimizer": veil_visibility_optimizer
    }
    func = registry.get(op_name)
    if not func:
            raise SystemExit(f"Unknown operator: {op_name}")
    return func(exp_dirs)
```

D.2 Python: orchestrator.py (CLI dispatcher)

```
#!/usr/bin/env python3
# Command-line wrapper to run operators over experiment directories.
# Usage:
#   python python_ops/orchestrator.py --op entropy_curvature_correlation --glob
"experiments/Tier_I/*"

import argparse, glob, json
from pathlib import Path
from operators import run_registry

def main():
        ap = argparse.ArgumentParser()
        ap.add_argument("--op", required=True, help="operator name (see
    operators.run_registry)")
        ap.add_argument("--glob", default="experiments/Tier_I/*", help="glob for experiment
    directories")
        ap.add_argument("--out", default="logs/analysis/result.json", help="output JSON
    file")
        args = ap.parse_args()

        exp_dirs = [p for p in glob.glob(args.glob) if Path(p).is_dir()]
        result = run_registry(args.op, exp_dirs)

        Path(args.out).parent.mkdir(parents=True, exist_ok=True)
        with open(args.out, "w") as f:
                json.dump(result, f, indent=2)
        print(f"[✓] {args.op} → {args.out}")

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

D.3 Shell Automation

run_all.sh

```
#!/usr/bin/env bash
# Orchestrate generation + analysis in one pass.
set -euo pipefail

# Generate a few sample experiments (varying names/tiers)
blender -b -P bpy_scripts/generate_experiment.py -- --name exp_entropy_cascade --tier Tier_I
--frames 120
blender -b -P bpy_scripts/generate_experiment.py -- --name exp_flow_crystal     --tier
Tier_II --frames 120
blender -b -P bpy_scripts/generate_experiment.py -- --name exp_semantic_weave   --tier
Tier_III --frames 120

# Optional time series
blender -b -P bpy_scripts/simulate_entropy_field.py

# Run operators over all Tier_I dirs
python python_ops/orchestrator.py --op entropy_curvature_correlation --glob
"experiments/Tier_I/*" --out logs/analysis/tier1_corr.json

# Aggregate another operator on Tier_II and Tier_III
python python_ops/orchestrator.py --op drift_phase_coherence --glob "experiments/Tier_II/*"
--out logs/analysis/tier2_plv.json
python python_ops/orchestrator.py --op drift_phase_coherence --glob "experiments/Tier_III/*"
--out logs/analysis/tier3_plv.json

echo "[✓] Full pipeline complete."
```

environment_setup.sh

```
#!/usr/bin/env bash
# environment_setup.sh — install dependencies (Debian/Ubuntu example)
set -euo pipefail

# Blender (if not present) — many systems already include 3D stack separately
# sudo apt update && sudo apt install -y blender

# Python deps
python3 - <<'PY'
import sys, subprocess
pkgs = ["numpy", "pandas"]
for p in pkgs:
        try:
                __import__(p)
                print(f"[✓] {p} present")
        except ImportError:
                print(f"Installing {p} ...")
                subprocess.check_call([sys.executable, "-m", "pip", "install", p])
print("[✓] Python environment ready.")
PY
```

cron_schedule.txt

```
# cron_schedule.txt — run nightly at 03:00 local
0 3 * * * /absolute/path/to/rsvp_tartan_clio/automation/run_all.sh >>
/absolute/path/to/rsvp_tartan_clio/logs/analysis/nightly.log 2>&1
```

D.4 End-to-End Execution Workflow

```
1. Ensure environment: ./automation/environment_setup.sh
2. Generate experiments (Tier I–III examples): ./automation/run_all.sh
3. Inspect outputs:
      - experiments/Tier_*/exp_*/entropy.json, curvature.json, mesh.obj, snapshot.png
      - logs/analysis/*.json (operator results)
4. Extend:
      - Add new experiments in bpy_scripts/
      - Add new operators in python_ops/operators.py and invoke via orchestrator.py
5. Schedule: Add cron_schedule.txt to your crontab.
```