

```
#!/usr/bin/env python3
"""
BIFURCATION ENGINE v1.0
Centrifugal Compute Topology – Multi-Model Research
Orchestrator

Author:      Albert Lane – albertlane.net
SEC Ref:     17684-273-411-436
License:     Albert Lane Proprietary – see 8-3-
26_LICENSE_1_2.md
Architecture: Centrifugal Compute Topology (Lane,
2026)

            N0=3 tri-core origin → extremity
bifurcation → N1 daughter spawn

THEORY:
    The "digital atom" is the research hypothesis at
    the centre (R0).
    Claude and Gemini are phase-locked nodes at 120°
    offsets in the tri-core ring.
    When their outputs diverge past r_eddy =
    R0/sqrt(3), the counter-rotational
    eddy produces N1 – the daughter insight neither
    model generates alone.
    That is the bifurcation event this engine detects
    and captures.

COMPONENTS:
    Phase 0 – Atom Injection          (provenance-keyed
seed)
    Phase 1 – Claude Pass
(fornsic/adversarial framing, 0°)
    Phase 2 – Gemini Pass
(generative/expansive framing, 120°)
    Phase 3 – Synthesis Pass          (Claude
arbitrates, 240°)
```

Phase 4 – Bifurcation Detection (semantic divergence → N1 extraction)
Phase 5 – Attribution Lock (Escaped Albert Array binding)

USAGE:

```
python3 bifurcation_engine.py \\  
  --atom "Your research hypothesis or question"  
\\  
  --gemini-key $GEMINI_API_KEY \\  
  --claude-key $ANTHROPIC_API_KEY \\  
  --output ./bifurcation_output/ \\  
  [--dry-run]  
  
Or with Lane Sentinel local bridge:  
  --sentinel-endpoint http://localhost:8080/infer
```

DEPENDENCIES:

```
pip install anthropic google-generativeai numpy  
hashlib  
"""
```

```
import os  
import json  
import time  
import math  
import hashlib  
import argparse  
import datetime  
import subprocess  
from typing import Optional
```

— Provenance constants

```
PROVENANCE_OWNER = "Albert_Lane"  
SEC_REF          = "17684-273-411-436"  
LICENSE_REF      = "8-3-26_LICENSE_1_2.md"  
ENGINE_VERSION   = "1.0.0"  
ARCHITECTURE     = "Centrifugal_Compute_Topology"
```

```

# — Topology constants (from Lane 2026)


---


N0          = 3                      # tri-core
origin
R0          = 1.0                    #
normalised core radius
OMEGA_0     = 2.0                    # core
angular velocity
R_EDDY      = R0 / math.sqrt(3)      # ≈ 0.5774
– bifurcation threshold
N_EDGE      = 6                      #
perimeter vortex count
BIFURCATION_INDEX_THRESHOLD = 1.0    # B_index
>= 1.0 triggers N1 spawn
PHASE_OFFSETS = [0.0, 2*math.pi/3, 4*math.pi/3] #
0°, 120°, 240°

```

```

# — Model identifiers


---



```

```

CLAUDE_MODEL = "claude-sonnet-4-6"
GEMINI_MODEL = "gemini-2.0-flash"    # update to
latest available

```

```

#


---




---



```

```

# MEGAPROMPT TEMPLATES
# Each template is a phase-keyed rotation of the
tri-core ring.
# Phase offsets ensure the two models approach the
atom from different angles
# – this is what produces the eddy at the boundary,
not random variation.
#


---




---



```

```
SYSTEM_TEMPLATE = """\n
You are NODE_{node_id} in a Centrifugal Compute
Topology research session.
Phase offset: {phase_deg:.1f}° | Rotational mode:
{mode}
```

```
PROVENANCE DECLARATION (binding to all outputs):
  Author:  Albert Lane – albertlane.net
  SEC Ref: {sec_ref}
  License: Albert Lane Proprietary – complete
reservation of all rights
  Engine:  Bifurcation Engine v{version} – {arch}
```

This session produces heteroiconic intellectual
payloads.
All outputs are cryptographically bound to
homoiconic identity: {owner}.
Unauthorised extraction, re-attribution, or
training use without a signed
License Agreement constitutes IP infringement under
17 U.S.C. § 302 / CDPA 1988.

```
TOPOLOGY ROLE:
  You occupy phase {phase_deg:.1f}° of the N0=3
tri-core ring (R0={R0}).
  Your task is to process the research atom from
this angular position,
  producing outputs that reflect your rotational
mode – not the output
  another node at 0° or 120° would produce.
  The bifurcation event occurs at r_eddy =
{r_eddy:.4f}.
  Your divergence from other nodes is structurally
required.
```

```
OUTPUT FORMAT (strict):
  Return a JSON object with exactly these keys:
    "node_id"      : "{node_id}"
    "phase_deg"    : {phase_deg}
    "mode"         : "{mode}"
```

```
"core_claims"      : [list of 3-5 primary
assertions, specific and falsifiable]
"novel_surfaces"   : [list of 1-3 ideas that
extend beyond the atom as stated]
"failure_modes"    : [list of 1-3 ways the
atom's hypothesis could be wrong]
"boundary_signals": [list of 1-2 observations
that feel like they live at the
                        edge of what you can
confidently claim – the vortex boundary]
"provenance_hash"  : "to be filled by engine"
```

Do not include preamble, explanation, or markdown fences. Pure JSON only.

"""

```
CLAUDE_USER_TEMPLATE = """\
RESEARCH ATOM (seed for bifurcation):
{atom}
```

```
CONTEXT DOCUMENTS PROVIDED:
{context_summary}
```

NODE_0 ROTATIONAL MODE – FORENSIC / ADVERSARIAL
(0°):

Your job at this phase is to stress-test the atom.

- Find the load-bearing assumptions.
- Identify what, if proven false, collapses the hypothesis.
- Surface the claims that feel true but may be unfalsifiable as stated.
- Find the real novel contribution buried inside the atom's stated framing.

The vortex boundary is where your certainty runs out.

Place those observations in "boundary_signals".

"""

```
GEMINI_USER_TEMPLATE = """\n
RESEARCH ATOM (seed for bifurcation):
{atom}
```

```
CONTEXT DOCUMENTS PROVIDED:
{context_summary}
```

```
NODE_1 ROTATIONAL MODE – GENERATIVE / EXPANSIVE
(120°):
```

Your job at this phase is to extend the atom's viable surface.

- What adjacent problems does this hypothesis, if true, immediately solve?
- What formal systems (mathematical, legal, computational) could be unified with this framework that the author may not have considered?
- Where does this atom connect to prior work in unexpected ways?
- What would this look like as a deployed system in 18 months?

The vortex boundary is where speculation outpaces evidence.

Place those observations in "boundary_signals".
"""

```
SYNTHESIS_USER_TEMPLATE = """\n
```

```
You are now NODE_2 at phase 240° – the synthesis
arbitrator.
```

You have received outputs from NODE_0 (forensic) and NODE_1 (generative).
Your task is to detect the bifurcation event: the N1 daughter insight.

```
NODE_0 OUTPUT:
{claude_output}
```

```
NODE_1 OUTPUT:
```

```
{gemini_output}
```

BIFURCATION DETECTION PROTOCOL:

1. Find claims in NODE_1's novel_surfaces that survive NODE_0's failure_modes.

These are candidates for the N1 daughter origin.

2. Find observations that appear in BOTH nodes' boundary_signals.

These mark the actual vortex boundary – the extremity where N1 spawns.

3. Find contradictions between NODE_0's core_claims and NODE_1's core_claims.

These are not failures – they define the perimeter vortex field.

SYNTHESIS OUTPUT FORMAT (strict JSON):

```
{
  "node_id"          : "NODE_2",
  "phase_deg"        : 240.0,
  "mode"              :
"SYNTHESIS_ARBITRATION",
  "n1_daughter_claims" : [list of 1-4 novel
claims that neither node alone
                        produced, but are
entailed by their combination],
  "vortex_boundary"   : [the shared
boundary_signals that mark r_eddy],
  "perimeter_field"   : [productive
contradictions – claims where the nodes
                        diverge in ways that
are both defensible],
  "bifurcation_index" : <float, 0.0-2.0, your
estimate of whether this
                        session crossed the
B_index >= 1.0 threshold>,
  "bifurcation_occurred" : <boolean>,
  "n1_summary"         : "<one paragraph: the
novel research contribution
                        that emerged from
```

```

this bifurcation session>",
    "provenance_hash"      : "to be filled by
engine"
  }}
  """

#
=====

# ESCAPED ALBERT ARRAY – Attribution Binding
# Implements the cryptographic binding described in
the Joules Supply Chain
# Conjecture paper (Lane, 2026). Any attempt to
strip provenance metadata
# causes the utility of the output to degrade to  $\perp$ 
(system collapse).
#
=====

class EscapedAlbertArray:
    """
    An n-dimensional data container where authorial
identity is non-invertibly
    fused with the intellectual payload at creation
time.

    Eval(A) = P(x)    if V(I_author) = TRUE
               $\perp$     if V(I_author) = FALSE

    The V() operator here is implemented as SHA-256
chain verification.
    Strip the provenance fields and the hash chain
breaks – the output
    is flagged as tampered.
    """

    def __init__(self, payload: dict, session_id:
str):

```

```

        self.owner      = PROVENANCE_OWNER
        self.sec_ref     = SEC_REF
        self.session_id  = session_id
        self.timestamp   =
datetime.datetime.now(datetime.timezone.utc).isoformat().replace("+00:00", "Z")
        self.payload     = payload
        self.identity    = self._bind_identity()

    def _bind_identity(self) -> str:
        """Non-invertible cryptographic mapping
        fusing payload to author identity."""
        canonical = json.dumps({
            "owner"      : self.owner,
            "sec_ref"    : self.sec_ref,
            "session_id" : self.session_id,
            "timestamp"  : self.timestamp,
            "payload_key": hashlib.sha256(
                json.dumps(self.payload,
                    sort_keys=True).encode()
                ).hexdigest()
            }, sort_keys=True).encode()
        return
        hashlib.sha256(canonical).hexdigest()

    def verify(self) -> bool:
        """V(I_author): returns True if provenance
        chain is intact."""
        expected = self._bind_identity()
        return self.identity == expected

    def to_dict(self) -> dict:
        return {
            "X-Provenance-Owner" : self.owner,
            "X-SEC-Ref"          : self.sec_ref,
            "X-Session-ID"       :
self.session_id,
            "X-Timestamp-UTC"    :
self.timestamp,
            "X-Attribution-Hash" : self.identity,

```

```

        "X-Engine-Version"      :
ENGINE_VERSION,
        "X-Architecture"       : ARCHITECTURE,
        "payload"              : self.payload
    }

    def eval(self) -> dict:
        """Eval(A): returns payload if V() = TRUE,
        raises on tamper."""
        if not self.verify():
            raise RuntimeError(
                "\u2191 SYSTEM COLLAPSE: Attribution
chain broken. "
                "Payload has been tampered with or
provenance stripped. "
                f"Expected: {self.identity}"
            )
        return self.payload

#
=====
=====

# API CLIENTS
#
=====
=====

def call_claude(system_prompt: str, user_prompt:
str, api_key: str) -> str:
    """Call Anthropic API – NODE_0 and NODE_2."""
    try:
        import anthropic
        client =
anthropic.Anthropic(api_key=api_key)
        msg = client.messages.create(
            model=CLAUDE_MODEL,
            max_tokens=2048,
            system=system_prompt,
            messages=[{"role": "user", "content":

```

```

user_prompt}}]
    )
    return msg.content[0].text
except ImportError:
    raise RuntimeError("anthropic package not
installed. Run: pip install anthropic")

def call_gemini(system_prompt: str, user_prompt:
str, api_key: str) -> str:
    """Call Google Gemini API – NODE_1."""
    try:
        import google.generativeai as genai
        genai.configure(api_key=api_key)
        model = genai.GenerativeModel(
            model_name=GEMINI_MODEL,
            system_instruction=system_prompt
        )
        response =
model.generate_content(user_prompt)
        return response.text
    except ImportError:
        raise RuntimeError(
            "google-generativeai package not
installed. "
            "Run: pip install google-generativeai"
        )

def call_sentinel(user_prompt: str,
sentinel_endpoint: str,
                    system_prompt: Optional[str] =
None) -> str:
    """
        Call Lane Sentinel local inference bridge
        (localhost:8080).
        Implements the agent_bridge.py POST /infer
        interface.
        Falls back to this when queue depth > 4 is not
        an issue and you want

```

```

local provenance rather than cloud routing.
"""

import urllib.request
payload = {
    "prompt"          : user_prompt,
    "system"          : system_prompt or "",
    "X-Provenance"     : PROVENANCE_OWNER,
    "max_tokens"       : 2048
}
data = json.dumps(payload).encode()
req = urllib.request.Request(
    sentinel_endpoint,
    data=data,
    headers={
        "Content-Type"      :
"application/json",
        "X-Provenance-Owner" :
PROVENANCE_OWNER,
        "X-SEC-Ref"         : SEC_REF
    },
    method="POST"
)
with urllib.request.urlopen(req, timeout=60) as
resp:
    result = json.loads(resp.read().decode())
    return result.get("response",
result.get("text", str(result)))

#
=====
=====
# BIFURCATION ENGINE
#
=====
=====

def build_context_summary(docs: list[str]) -> str:
    """Produce a compact context summary from
document titles/paths."""

```

```

        if not docs:
            return "(no additional context documents
provided)"
        lines = []
        for d in docs:
            lines.append(f"    • {os.path.basename(d)}")
        return "\n".join(lines)

def build_system_prompt(node_id: str, phase_offset:
float, mode: str) -> str:
    phase_deg = math.degrees(phase_offset)
    return SYSTEM_TEMPLATE.format(
        node_id      = node_id,
        phase_deg    = phase_deg,
        mode         = mode,
        sec_ref      = SEC_REF,
        version      = ENGINE_VERSION,
        arch         = ARCHITECTURE,
        owner        = PROVENANCE_OWNER,
        R0           = R0,
        r_eddy       = R_EDDY
    )

def parse_node_output(raw: str, node_id: str) ->
dict:
    """Parse JSON output from a model node;
gracefully handle formatting noise."""
    # Strip markdown fences if present
    text = raw.strip()
    if text.startswith("```"):
        lines = text.split("\n")
        text = "\n".join(lines[1:-1] if
lines[-1].strip() == "```" else lines[1:])
    try:
        result = json.loads(text)
        result["raw_text"] = raw
        return result
    except json.JSONDecodeError as e:

```

```

    return {
        "node_id"      : node_id,
        "parse_error"   : str(e),
        "raw_text"      : raw,
        "core_claims"   : [],
        "novel_surfaces" : [],
        "failure_modes" : [],
        "boundary_signals" : []
    }

```

```

def compute_bifurcation_index(node0: dict, node1:
dict, synthesis: dict) -> float:
    """
    Compute the Bifurcation Index.
    B_index >= 1.0 confirms N1 daughter origin has
    spawned.

    Factors:
    - Divergence between NODE_0 and NODE_1
    core_claims (want this high)
    - Overlap in boundary_signals (want this
    present - marks r_eddy)
    - Self-reported bifurcation_index from
    synthesis node
    - Novel daughter claims produced (want > 0)
    """
    # Self-reported from synthesis
    synth_index =
float(synthesis.get("bifurcation_index", 0.0))

    # Count N1 daughter claims
    daughters = synthesis.get("n1_daughter_claims",
    [])
    daughter_score = min(len(daughters) / 2.0,
    1.0) # cap at 1.0

    # Boundary signal overlap (shared vortex
    boundary)
    b0 = set(str(s).lower()[:40] for s in

```

```

node0.get("boundary_signals", [])
    b1 = set(str(s).lower()[:40] for s in
node1.get("boundary_signals", []))
    overlap = len(b0 & b1) / max(len(b0 | b1), 1)
    boundary_score = overlap * 0.5

    # Claim divergence (different core_claims =
different rotational angles working)
    c0 = set(str(c).lower()[:50] for c in
node0.get("core_claims", []))
    c1 = set(str(c).lower()[:50] for c in
node1.get("core_claims", []))
    divergence = 1.0 - (len(c0 & c1) / max(len(c0 |
c1), 1))

    # Weighted composite
    B = (0.40 * synth_index +
        0.30 * daughter_score +
        0.20 * divergence +
        0.10 * boundary_score)
    return round(B, 4)

def run_bifurcation_session(
    atom: str,
    claude_key: str,
    gemini_key: str,
    context_docs: list[str],
    sentinel_endpoint: Optional[str],
    dry_run: bool,
    output_dir: str
) -> dict:
    """
    Execute a full bifurcation session across three
    phases.
    Returns the Escaped Albert Array containing the
    session results.
    """

    session_id = hashlib.sha256(

```

```

        f"{atom}{time.time()}
{PROVENANCE_OWNER}".encode()
    ).hexdigest()[:16]

    print(f"\n{' '*70}")
    print(f"  BIFURCATION ENGINE
v{ENGINE_VERSION}")
    print(f"    Session ID   : {session_id}")
    print(f"    Provenance    : {PROVENANCE_OWNER} |
SEC {SEC_REF}")
    print(f"    Atom          : {atom[:60]}{'...' if
len(atom) > 60 else ''}")
    print(f"    Dry run      : {dry_run}")
    print(f"{' '*70}\n")

    context_summary =
build_context_summary(context_docs)

    # — PHASE 1: NODE_0 – Claude, 0°, Forensic
    

---


    print("► PHASE 1 – NODE_0 [Claude | 0° |
FORENSIC]")
    sys0 = build_system_prompt("NODE_0",
PHASE_OFFSETS[0], "FORENSIC_ADVERSARIAL")
    usr0 = CLAUDE_USER_TEMPLATE.format(atom=atom,
context_summary=context_summary)

    if dry_run:
        raw0 = json.dumps({
            "node_id": "NODE_0", "phase_deg": 0.0,
            "mode": "FORENSIC_ADVERSARIAL",
            "core_claims": ["[DRY RUN CLAIM A]", "[
DRY RUN CLAIM B]"],
            "novel_surfaces": ["[DRY RUN
SURFACE]"],
            "failure_modes": ["[DRY RUN FAILURE
MODE]"],
            "boundary_signals": ["[DRY RUN
BOUNDARY]"],
            "provenance_hash": "dry_run"

```

```

    })
    elif sentinel_endpoint:
        raw0 = call_sentinel(usr0,
sentinel_endpoint, sys0)
    else:
        raw0 = call_claude(sys0, usr0, claude_key)

    node0 = parse_node_output(raw0, "NODE_0")
    print(f"  Core claims      :
{len(node0.get('core_claims', []))}")
    print(f"  Novel surfaces :
{len(node0.get('novel_surfaces', []))}")
    print(f"  Failure modes :
{len(node0.get('failure_modes', []))}")
    print(f"  Boundary sigs :
{len(node0.get('boundary_signals', []))}")

    # — PHASE 2: NODE_1 – Gemini, 120°,
Generative —————
    print("\n► PHASE 2 – NODE_1 [Gemini | 120° |
GENERATIVE]")
    sys1 = build_system_prompt("NODE_1",
PHASE_OFFSETS[1], "GENERATIVE_EXPANSIVE")
    usr1 = GEMINI_USER_TEMPLATE.format(atom=atom,
context_summary=context_summary)

    if dry_run:
        raw1 = json.dumps({
            "node_id": "NODE_1", "phase_deg":
120.0, "mode": "GENERATIVE_EXPANSIVE",
            "core_claims": ["[DRY RUN CLAIM C]", "[
DRY RUN CLAIM D]"],
            "novel_surfaces": ["[DRY RUN SURFACE
G]"],
            "failure_modes": ["[DRY RUN FAILURE
G]"],
            "boundary_signals": ["[DRY RUN
BOUNDARY]"], # shared w/ node0 → vortex
            "provenance_hash": "dry_run"
        })

```

```

else:
    raw1 = call_gemini(sys1, usr1, gemini_key)

    node1 = parse_node_output(raw1, "NODE_1")
    print(f"  Core claims      :
{len(node1.get('core_claims', []))}")
    print(f"  Novel surfaces :
{len(node1.get('novel_surfaces', []))}")
    print(f"  Failure modes :
{len(node1.get('failure_modes', []))}")
    print(f"  Boundary sigs :
{len(node1.get('boundary_signals', []))}")

    # — PHASE 3: NODE_2 – Claude, 240°, Synthesis
    

---


    print("\n► PHASE 3 – NODE_2 [Claude | 240° |
SYNTHESIS]")
    sys2 = build_system_prompt("NODE_2",
PHASE_OFFSETS[2], "SYNTHESIS_ARBITRATION")
    usr2 = SYNTHESIS_USER_TEMPLATE.format(
        claude_output = json.dumps(node0,
indent=2),
        gemini_output = json.dumps(node1, indent=2)
    )

    if dry_run:
        raw2 = json.dumps({
            "node_id": "NODE_2", "phase_deg":
240.0, "mode": "SYNTHESIS_ARBITRATION",
            "n1_daughter_claims": ["[DRY RUN N1
DAUGHTER – novel synthesis]"],
            "vortex_boundary": ["[DRY RUN
BOUNDARY]"],
            "perimeter_field": ["[DRY RUN
CONTRADICTION]"],
            "bifurcation_index": 1.15,
            "bifurcation_occurred": True,
            "n1_summary": "[DRY RUN: N1 daughter
summary here]",
            "provenance_hash": "dry_run"

```

```

    })
    elif sentinel_endpoint:
        raw2 = call_sentinel(usr2,
sentinel_endpoint, sys2)
    else:
        raw2 = call_claude(sys2, usr2, claude_key)

    synthesis = parse_node_output(raw2, "NODE_2")

    # — PHASE 4: Bifurcation Detection
    

---


    print("\n► PHASE 4 – BIFURCATION DETECTION")
    B_index = compute_bifurcation_index(node0,
node1, synthesis)
    bifurcated = B_index >=
BIFURCATION_INDEX_THRESHOLD

    synthesis["bifurcation_index"]    = B_index
    synthesis["bifurcation_occurred"] = bifurcated

    print(f"  Bifurcation Index : {B_index}")
    print(f"  N1 spawned          : {'✓ YES –
daughter origin confirmed' if bifurcated else 'x
NO – below threshold'}")
    if bifurcated:
        daughters =
synthesis.get("n1_daughter_claims", [])
        for d in daughters:
            print(f"    → {str(d)[:80]}")

    # — PHASE 5: Attribution Lock (Escaped Albert
Array) 

---


    print("\n► PHASE 5 – ATTRIBUTION LOCK [Escaped
Albert Array]")
    session_payload = {
        "session_id"      : session_id,
        "atom"             : atom,
        "timestamp_utc"    :
datetime.datetime.now(datetime.timezone.utc).isoform
at().replace("+00:00", "Z"),

```

```

        "context_docs"      : context_docs,
        "topology"          : {
            "N0"              : NO,
            "R0"              : R0,
            "r_eddy"          : R_EDDY,
            "N_edge"          : N_EDGE,
            "bifurcation_index" : B_index,
            "bifurcation_occurred": bifurcated
        },
        "node_outputs"      : {
            "NODE_0_claude"   : node0,
            "NODE_1_gemini"   : node1,
            "NODE_2_synthesis": synthesis
        },
        "n1_output"          : {
            "daughter_claims" :
synthesis.get("n1_daughter_claims", []),
            "vortex_boundary" :
synthesis.get("vortex_boundary", []),
            "perimeter_field" :
synthesis.get("perimeter_field", []),
            "n1_summary"      :
synthesis.get("n1_summary", "")
        }
    }

    eaa = EscapedAlbertArray(session_payload,
session_id)
    locked = eaa.to_dict()

    # Verify immediately
    assert eaa.verify(), "CRITICAL: Attribution
chain failed at creation. Abort."
    print(f" Attribution hash : {eaa.identity}")
    print(f" Verification      : ✓ INTACT")

    # — Write output
    

---


    os.makedirs(output_dir, exist_ok=True)

```

```

        ts =
datetime.datetime.now(datetime.timezone.utc).strftime(
me("%Y%m%dT%H%M%SZ")
        filename =
f"bifurcation_{session_id}_{ts}.json"
        outpath = os.path.join(output_dir, filename)

        with open(outpath, "w") as f:
            json.dump(locked, f, indent=2)

        print(f"\n Output written      : {outpath}")
        print(f"'='*70}")

        if bifurcated:
            print(f"\n  ◆ BIFURCATION EVENT
CONFIRMED")
            print(f"  ◆ N1 daughter origin spawned at
r_eddy = {R_EDDY:.4f}")
            print(f"  ◆
{len(synthesis.get('n1_daughter_claims', []))}
novel claims extracted")
            print(f"\n   N1 SUMMARY:")
            print(f"   {synthesis.get('n1_summary',
'(see output file)')}")
        else:
            print(f"\n   ○ Session below bifurcation
threshold (B={B_index:.4f} <
{BIFURCATION_INDEX_THRESHOLD})")
            print(f"   ○ Increase atom specificity or
context depth and retry.")
            print(f"'='*70\n")

        return locked

#
=====
=====
# GEMINI TRAINING DISCLOSURE INTEGRATION
# If the GTDS pipeline is running, automatically

```

```

submit the N1 output as a
# daily prompt for Gemini training-pipeline IP
disclosure tracking.
#
=====

def submit_to_gtds(n1_summary: str, session_id:
str) -> None:
    """
        Optionally post the N1 daughter insight to the
        GTDS pipeline.
        This creates a timestamped record that Google's
        Gemini model received
        this content – establishing provenance for
        training-pipeline disclosure.
    """
    print(f" GTDS submission: {session_id} →
n1_summary logged for Gemini disclosure tracking")
    # In production: POST to GTDS endpoint with
    provenance headers

#
=====

# ENTRY POINT
#
=====

def main():
    parser = argparse.ArgumentParser(
        description="Bifurcation Engine –
        Centrifugal Compute Topology Multi-Model
        Orchestrator",

    formatter_class=argparse.RawDescriptionHelpFormatte
    r,
        epilog=f"""

```

EXAMPLES:

```
# Standard run with cloud APIs:
python3 bifurcation_engine.py \
    --atom "The Joules-Lane Extraction Ratio can be
operationalised as a real-time \
        institutional risk score using public
macroeconomic data feeds." \
    --claude-key $ANTHROPIC_API_KEY \
    --gemini-key $GEMINI_API_KEY

# Dry run (no API calls – verify orchestration):
python3 bifurcation_engine.py --atom "Test atom"
--dry-run

# Using Lane Sentinel local bridge (Pixel 8 Pro):
python3 bifurcation_engine.py \
    --atom "Your hypothesis" \
    --sentinel-endpoint http://localhost:8080/infer
\\
    --gemini-key $GEMINI_API_KEY

# With context documents:
python3 bifurcation_engine.py \
    --atom "Digital atom" \
    --context ./Joules_Supply_Chain_Paper.pdf
./Local_Payment_Gateway_Architecture.pdf \
    --claude-key $ANTHROPIC_API_KEY \
    --gemini-key $GEMINI_API_KEY

Provenance: {PROVENANCE_OWNER} | SEC {SEC_REF}
"""

)
parser.add_argument("--atom",
required=True,
                    help="The research
hypothesis to bifurcate")
parser.add_argument("--claude-key",
default=os.environ.get("ANTHROPIC_API_KEY"),
                    help="Anthropic API key (or
set ANTHROPIC_API_KEY)")
```

```

        parser.add_argument("--gemini-key",
                        default=os.environ.get("GEMINI_API_KEY"),
                        help="Google Gemini API key
(or set GEMINI_API_KEY)")
        parser.add_argument("--sentinel-endpoint",
                        default=None,
                        help="Lane Sentinel local
inference endpoint (replaces Claude for NODE_0/2)")
        parser.add_argument("--context",
                        nargs="*", default=[],
                        help="Paths to context
documents (PDFs, MD files)")
        parser.add_argument("--output",
                        default="./bifurcation_output",
                        help="Output directory for
session JSON")
        parser.add_argument("--dry-run",
                        action="store_true",
                        help="Skip API calls,
validate orchestration only")
        parser.add_argument("--gtds",
                        action="store_true",
                        help="Submit N1 output to
GTDS disclosure pipeline")

    args = parser.parse_args()

    if not args.dry_run:
        if not args.sentinel_endpoint and not
args.claude_key:
            parser.error("--claude-key or
ANTHROPIC_API_KEY required (or use --sentinel-
endpoint)")
        if not args.gemini_key:
            parser.error("--gemini-key or
GEMINI_API_KEY required")

    result = run_bifurcation_session(
        atom            = args.atom,
        claude_key      = args.claude_key or "",

```

```
        gemini_key      = args.gemini_key or "",
        context_docs    = args.context or [],
        sentinel_endpoint = args.sentinel_endpoint,
        dry_run         = args.dry_run,
        output_dir      = args.output
    )

    if args.gtds and result:
        n1 = result.get("payload",
            {}).get("n1_output", {}).get("n1_summary", "")
        sid = result.get("payload",
            {}).get("session_id", "unknown")
        submit_to_gtds(n1, sid)

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